

Developing Apama Applications

Version 10.7.1

April 2021

This document applies to Apama 10.7.1 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 2013-2021 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at <https://softwareag.com/licenses/>.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <https://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at <https://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

Document ID: PAM-DEV-1071-20210415

Table of Contents

About this Guide.....	11
Documentation roadmap.....	12
Online Information and Support.....	13
Data Protection.....	14
 I Developing Apama Applications in EPL.....	15
1 Getting Started with Apama EPL.....	17
Introduction to Apama Event Processing Language.....	18
How EPL applications compare to applications in other languages.....	19
About dynamic compilation in the correlator.....	20
About the Apama development environment in Software AG Designer.....	20
Terminology.....	21
Defining event types.....	26
Working with events.....	29
2 Defining Monitors.....	33
About monitor contents.....	34
Example of a simple monitor.....	37
Spawning monitor instances.....	39
Communication among monitor instances.....	45
About service monitors.....	52
Adding predefined annotations.....	52
Subscribing to channels.....	54
Adding service monitor bundles to your project.....	57
Utilities for operating on monitors.....	57
3 Defining Queries.....	59
Introduction to queries.....	60
Format of query definitions.....	68
Defining metadata in a query.....	70
Partitioning queries.....	71
Defining query input.....	77
Finding and acting on event patterns.....	103
Implementing parameterized queries.....	129
Restrictions in queries.....	134
Best practices for defining queries.....	135
Testing query execution.....	138
Communication between monitors and queries.....	140
4 Defining Event Listeners.....	143
About event expressions and event templates.....	144
Specifying the on statement.....	147
Using a stream source template to find events of interest.....	148
Defining event expressions with one event template.....	148
Terminating and changing event listeners.....	153
Specifying multiple event listeners.....	155
Listening for events that do not match.....	156
Specifying completion event listeners.....	157

Improving performance by ignoring some fields in matching events.....	159
Defining event listeners for patterns of events.....	160
Specifying and/or/not logic in event listeners.....	162
How the correlator executes event listeners.....	167
Defining event listeners with temporal constraints.....	174
Understanding time in the correlator.....	179
Out of band connection notifications.....	185
5 Working with Streams and Stream Queries.....	187
Introduction to streams and stream networks.....	188
Defining streams.....	189
Using output from streams.....	190
Defining stream queries.....	193
Defining custom aggregate functions.....	221
Working with lots that contain multiple items.....	225
Stream network lifetime.....	229
Using dynamic expressions in stream queries.....	232
Troubleshooting and stream query coding guidelines.....	239
6 Defining What Happens When Matching Events Are Found.....	245
Using variables.....	246
Defining actions.....	250
Defining static actions.....	262
Getting the current time.....	263
Generating events.....	264
Handling the any type.....	269
Handling any values of different types with the switch statement.....	272
Assigning values.....	273
Defining conditional logic with the if statement.....	273
Defining conditional logic with the ifpresent statement.....	274
Defining loops.....	275
Exception handling.....	276
Logging and printing.....	279
Sample financial application.....	283
7 Implementing Parallel Processing.....	285
Introduction to contexts.....	286
Creating contexts.....	288
How many contexts can you create?.....	289
Using channels to communicate between contexts.....	289
Obtaining context references.....	290
Spawning to contexts.....	291
Channels and contexts.....	292
Sending an event to a channel.....	293
Sending an event to a particular context.....	294
Sending an event to a sequence of contexts.....	295
Common use cases for contexts.....	297
Samples for implementing contexts.....	297
Contexts and correlator determinism.....	304
How contexts affect other parts of your Apama application.....	304
8 Using Correlator Persistence.....	307
Description of state that can be persistent.....	308
When persistence is useful.....	309

When non-persistent monitors are useful.....	309
How the correlator persists state.....	310
Enabling correlator persistence.....	311
How the correlator recovers state.....	314
Designing applications for persistence-enabled correlators.....	316
Upgrading monitors in a persistence-enabled correlator.....	317
Sample code for persistence applications.....	318
Requesting snapshots from EPL.....	320
Developing persistence applications.....	320
Backing up the persistence database while the correlator is running.....	321
Using EPL plug-ins when persistence is enabled.....	323
Using the MemoryStore when persistence is enabled.....	323
Comparison of correlator persistence with other persistence mechanisms.....	325
Restrictions on correlator persistence.....	326
9 Common EPL Patterns in Monitors.....	327
Contrasting using a dictionary with spawning.....	328
Factory pattern.....	329
Using quit() to terminate event listeners.....	330
Combining the dictionary and factory patterns.....	331
Testing uniqueness.....	332
Reference counting.....	332
Inline request-response pattern.....	334
Writing echo monitors for debugging.....	335
Versioning and upgrading monitors.....	336
10 Using EPL Plug-ins.....	339
Overhead of using plug-ins.....	340
When to use plug-ins.....	340
When not to use plug-ins.....	341
Using the TimeFormat Event Library.....	341
Using the MemoryStore.....	350
Using the distributed MemoryStore.....	365
Using the Management interface.....	391
Using the JSON plug-in.....	395
Using MATLAB® products in an application.....	396
Using the R plug-in.....	404
Interfacing with user-defined EPL plug-ins.....	406
About the chunk type.....	406
11 Making Application Data Available to Clients.....	409
Adding the DataView Service bundle to your project.....	411
Creating DataView definitions.....	411
Deleting DataView definitions.....	412
Creating DataView items.....	412
Deleting DataView items.....	413
Updating DataView items.....	414
Looking up field positions.....	415
Using multiple correlators.....	415
12 Testing and Tuning EPL monitors.....	417
Optimizing EPL programs.....	418
Best practices for writing EPL.....	418
Structure of a basic test framework.....	420

Using event files.....	420
Handling runtime errors.....	421
Capturing test output.....	423
Avoiding listeners and monitor instances that never terminate.....	423
Handling slow or blocked receivers.....	424
Diagnosing infinite loops in the correlator.....	424
Tuning contexts.....	425
13 Generating Documentation for Your EPL Code.....	429
Code constructs that are documented.....	430
Steps for using ApamaDoc.....	430
Inserting ApamaDoc comments.....	431
Inserting ApamaDoc tags.....	432
Inserting ApamaDoc references.....	435
Inserting EPL source code examples.....	436
Generating ApamaDoc from the command line.....	437
Generating ApamaDoc from an Ant script.....	439
II Developing Apama Applications in Java.....	441
14 Overview of Apama JMon Applications.....	443
Introducing JMon API concepts.....	445
About event types.....	446
About monitors.....	452
About event listeners and match listeners.....	453
Description of the flow of execution in JMon applications.....	455
Parallel processing in JMon applications.....	456
Identifying external events.....	459
Optimizing event types.....	459
Logging in JMon applications.....	461
Using EPL keywords as identifiers in JMon applications.....	462
15 Defining Event Expressions.....	463
About event templates.....	464
Specifying parameter constraints in event templates.....	466
Obtaining matching events.....	468
Emitting, routing, and enqueueing events.....	469
Specifying temporal sequencing.....	471
Defining advanced event expressions.....	473
Optimizing event expressions.....	485
Validation of event expressions.....	486
16 Concept of Time in the Correlator.....	487
Getting the current time.....	488
About timers and their trigger times.....	489
17 Developing and Deploying JMon Applications.....	493
Steps for developing JMon applications in Software AG Designer.....	494
Java prerequisites for using Apama's JMon API.....	495
Steps for developing JMon applications manually.....	496
Deploying JMon applications.....	496
Removing JMon applications from the correlator.....	497
Creating deployment descriptor files.....	497
Package names and namespaces in JMon applications.....	505

Sample JMon applications.....	505
III Developing EPL Plug-ins.....	507
18 Introduction to EPL Plug-ins.....	509
19 Providing an EPL Event Wrapper for a Plug-in.....	511
20 Writing EPL Plug-ins in C++.....	513
Creating a plug-in using C++.....	514
Using plug-ins written in C++.....	525
21 Writing EPL Plug-ins in Java.....	527
Creating a plug-in using Java.....	528
Using EPL plug-ins written in Java.....	530
Sample plug-ins in Java.....	536
22 Writing EPL Plug-ins in Python.....	539
Creating a plug-in using Python.....	540
Using Python plug-ins.....	544
Installing Python modules.....	546
Sample plug-ins written in Python.....	547
IV Protecting Personal Data in Apama Applications.....	549
23 Introduction.....	551
24 Where personal data is held within the Apama platform.....	553
25 Documenting personal data flows within an Apama application.....	557
26 Handling personal data in the "in-memory" state of the correlator.....	559
27 Handling personal data "at rest" in the correlator persistence and JMS datastores.....	563
28 Handling personal data "in motion" from dashboards.....	565
29 Handling personal data "at rest" in log files.....	567
Example log messages containing personal data.....	568
Protecting and erasing data from Apama log files.....	569
Recommended log levels.....	570
Recommendations for logging by Apama application code.....	570
30 Handling personal data "at rest" in the correlator input log file.....	573
31 Handling personal data "at rest" in containerization environments.....	575
V EPL Reference.....	577
32 Introduction.....	579
Hello World example.....	580
33 Types.....	583
Primitive and string types.....	584
Reference types.....	584
Default values for types.....	586
Type properties summary.....	587
Timestamps, dates, and times.....	590
Type methods and instance methods.....	591
Type conversion.....	592
Comparable types.....	593
Cloneable types.....	594
Potentially cyclic types.....	595
Support for IEEE 754 special values.....	598

34 Events and Event Listeners.....	601
Event definitions.....	602
Event templates.....	604
Event listener definitions.....	609
Event lifecycle.....	609
Event listener lifecycle.....	610
Event processing order for monitors.....	611
Event processing order for queries.....	612
Event expressions.....	613
Event channels.....	618
35 Monitors.....	619
Monitor lifecycle.....	620
Monitor files.....	621
Packages.....	621
The using declaration.....	622
Monitor declarations.....	622
The import declaration.....	622
Monitor actions.....	623
Contexts.....	624
Plug-ins.....	625
Garbage collection.....	625
36 Queries.....	627
Query lifetime.....	628
Query definition.....	630
Metadata section.....	631
Parameters section.....	632
Inputs section.....	632
Query input definition.....	632
Find statement.....	634
37 Aggregate Functions.....	641
Built-in aggregate functions.....	642
Custom aggregates.....	642
38 Statements.....	645
Simple statements.....	646
Compound statements.....	651
Transfer of control statements.....	656
39 Expressions.....	659
Introduction to expressions.....	660
Using an expression as a statement.....	660
Primary expressions.....	661
Bitwise logical operators.....	661
Logical operators.....	663
Shift operators.....	664
Comparison operators.....	665
Additive operators.....	666
Multiplicative operators.....	667
Unary additive operators.....	668
Expression operators.....	668
Expression operator precedence.....	669
Postfix expressions.....	670

Stream queries.....	671
Stream source templates.....	675
40 Variables.....	677
Variable declarations.....	678
Variable scope.....	678
Provided variables.....	679
Specifying named constant values.....	681
41 Lexical Elements.....	683
Program text.....	684
Comments.....	684
White space.....	685
Line terminators.....	686
Symbols.....	687
Identifiers.....	687
Keywords.....	687
Operators.....	688
Separators.....	689
Literals.....	689
Names.....	693
Annotations.....	694
42 Limits.....	695
 A EPL Naming Conventions.....	 697
 B Testing Apama Applications Using PySys.....	 701

About this Guide

■ Documentation roadmap	12
■ Online Information and Support	13
■ Data Protection	14

Developing Apama Applications describes different technologies for developing Apama applications: EPL monitors, Apama queries, and Java. You can use one or several of these technologies to implement a single Apama application. In addition, there are C++ and Java APIs for developing components that plug in to a correlator. You can use these components from EPL.

Documentation roadmap

Apama provides documentation in the following formats:

- HTML (available from both the documentation website and the doc folder of the Apama installation)
- PDF (available from the documentation website)
- Eclipse help (accessible from Software AG Designer)

You can access the HTML documentation on your machine after Apama has been installed:

- **Windows.** Select **Start > All Programs > Software AG > Tools > Apama *n.n* > Apama Documentation *n.n***. Note that **Software AG** is the default group name that can be changed during the installation.
- **UNIX.** Display the `index.html` file, which is in the `doc/apama-onlinehelp` directory of your Apama installation directory.

The following guides are available:

Title	Description
<i>Release Notes</i>	Describes new features and changes introduced with the current Apama release as well as earlier releases.
<i>Installing Apama</i>	Summarizes all important installation information and is intended for use with other Software AG installation guides such as <i>Using Software AG Installer</i> .
<i>Introduction to Apama</i>	Provides a high-level overview of Apama, describes the Apama architecture, discusses Apama concepts and introduces Software AG Designer, which is the main development tool for Apama.
<i>Using Apama with Software AG Designer</i>	Explains how to develop Apama applications in Software AG Designer, which is an Eclipse-based integrated development environment.
<i>Developing Apama Applications</i>	Describes the different technologies for developing Apama applications: EPL monitors, Apama queries, and Java. You can use one or several of these technologies to implement a single Apama application. In addition, there are C++ and Java APIs for developing components that plug in to a correlator. You can use these components from EPL.

Title	Description
<i>Connecting Apama Applications to External Components</i>	Describes how to connect Apama applications to any event data source, database, messaging infrastructure, or application.
<i>Building and Using Apama Dashboards</i>	Describes how to build and use an Apama dashboard, which provides the ability to view and interact with DataViews. An Apama project typically uses one or more dashboards, which are created in the Dashboard Builder. The Dashboard Viewer provides the ability to use dashboards created in the Dashboard Builder. Dashboards can also be deployed as simple web pages. Deployed dashboards connect to one or more correlators by means of a dashboard data server or display server.
<i>Deploying and Managing Apama Applications</i>	Describes how to deploy components with Software AG Command Central, how to deploy and manage queries, and how to deploy Apama applications using Docker and Kubernetes. It also provides information for improving Apama application performance by using multiple correlators, for managing and monitoring Apama components over REST (Representational State Transfer), and for using correlator utilities and configuration files.

In addition to the above guides, Apama also provides the following API reference information:

- API Reference for EPL (ApamaDoc)
- API Reference for Java (Javadoc)
- API Reference for C++ (Doxygen)
- API Reference for .NET
- API Reference for Python
- API Reference for Component Management REST APIs

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at <http://documentation.softwareag.com>.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at <https://empower.softwareag.com/>.

You can find product information on the Software AG Empower Product Support website at <https://empower.softwareag.com>.

To submit feature/enhancement requests, get information about product availability, and download products, go to [Products](#).

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the [Knowledge Center](#).

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at <http://techcommunity.softwareag.com>. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

Developing Apama Applications in EPL

1	Getting Started with Apama EPL	17
2	Defining Monitors	33
3	Defining Queries	59
4	Defining Event Listeners	143
5	Working with Streams and Stream Queries	187
6	Defining What Happens When Matching Events Are Found	245
7	Implementing Parallel Processing	285
8	Using Correlator Persistence	307
9	Common EPL Patterns in Monitors	327
10	Using EPL Plug-ins	339
11	Making Application Data Available to Clients	409
12	Testing and Tuning EPL monitors	417
13	Generating Documentation for Your EPL Code	429

1 Getting Started with Apama EPL

■ Introduction to Apama Event Processing Language	18
■ How EPL applications compare to applications in other languages	19
■ About dynamic compilation in the correlator	20
■ About the Apama development environment in Software AG Designer	20
■ Terminology	21
■ Defining event types	26
■ Working with events	29

The correlator is Apama's core event processing and correlation engine. The interface to the correlator lets you inject events that the correlator analyzes. You can configure the correlator to watch for particular events or patterns of interest. In addition, you specify the actions to undertake when the correlator identifies such patterns. Identification of events of interest plus what to do when such events are found constitute an Apama application's logic.

To deploy an application on the correlator, you can use either the correlator's native Apama Event Processing Language (EPL) or the Apama in-process API for Java (JMon). The information presented in this part focuses exclusively on EPL.

This part teaches you how to write EPL programs. While some programming experience is assumed, no prior knowledge of EPL is assumed.

Apama EPL is an event-driven programming language. It lets you write applications that:

- Monitor streams of events to find particular events or patterns of events of interest.
- Analyze events (or patterns of events) of interest to determine whether some action is appropriate.
- Perform actions based on particular events or patterns of events.

This section discusses the main concepts you must understand to write applications in EPL.

Software AG Designer provides tutorials that can help you get started with EPL. On the Welcome page of Software AG Designer, click **Tutorials** under the **Apama** heading.

Note:

MonitorScript is the old name for EPL. You might still see the old name in the product documentation.

Introduction to Apama Event Processing Language

EPL is a flexible and powerful "curly-brace", domain-specific language designed for writing programs that process events. In EPL, an event is a data object that contains a notification of something that has happened, such as a customer order was shipped, a shipment was delivered, a sensor state change occurred, a stock trade took place, or myriad other things. Each kind of event has a type name and one or more data elements (called fields) associated with it. External events are received by one or more adapters, which receive events from the event source and translate them from a source-specific format into Apama's internal canonical format. Derived events can be created as needed by EPL programs.

Though it contains many of the familiar constructs and features found in general-purpose programming languages like Python or Java, EPL also has special features to make it easy to aggregate, filter, correlate, transform, act on, and create events in a concise manner. Here is the canonical "Hello world" example written in EPL:

```
monitor HelloWorld
{
    action onload()
    {
        print "Hello world!";
    }
}
```

```
}  
}
```

The Apama event processor, called the correlator, receives events of various types from external sources. The EPL programs that process these events are monitors or queries.

Monitors have registered event handlers, called listeners, for events of particular types with specific combinations of data values or ranges of values. When a listener detects an event of interest, it triggers a particular action. If there are no listeners for an event, the correlator either discards it or passes it to a listener specifically for events that have no handler. A monitor instance processes events on one correlator and can send events to communicate with other monitors on the same correlator or remote correlators.

Queries are scalable across multiple correlators. An Apama query operates on only the input event types you specify and you can filter which instances of those events should be processed. Apama partitions these incoming events according to a key field that you specify, for example, there might be a partition for each credit card number. The query processes the events in each partition independently of the events in every other partition. As events are added to partitions, the query checks for a set of events that matches the event pattern you specified, which can optionally specify complex conditions for there to be a match. When a match is found the query executes procedural code that you have defined, which can include sending events.

Event handlers in EPL are conceptually similar to methods or functions used for handling user-interface events in other languages, such as Java Swing or SWT applications. In EPL, the correlator executes code only in response to events.

The correlator is capable of looking for hundreds of thousands of different events or different event patterns concurrently. When you write an EPL application, you write a set of monitors and/or one or more queries and then you inject or load them into a running correlator. As streams of events pass into a correlator, the monitors and their listeners and/or the queries watch for the events or patterns of events that you have specified as being of interest. There are a variety of actions that you can specify that you want the correlator to perform when a listener or query detects an event or event pattern of interest. For example, the most common action for a monitor is to generate and dispatch a message to an external receiver.

EPL is case-sensitive.

How EPL applications compare to applications in other languages

EPL is an event-oriented programming language, as opposed to an object-oriented language. Because EPL is part of an event-processing framework, it requires a different approach to decomposing the problem you want to solve.

EPL syntax is similar to other scripting languages. EPL has variables, data structures, conditions, and procedures (called actions in EPL). But EPL supports a paradigm that is different from that supported by other scripting languages:

- A monitor or a query is the basic module in EPL programs.
- All communication is by means of message passing.

- All processing is triggered in response to events.
- Monitors spawn instances of themselves to generate multiple units of execution and/or to initiate parallel processing. A query uses a key to partition incoming events and can share the same data across multiple correlators.

EPL requires a different way of developing applications.

About dynamic compilation in the correlator

EPL is dynamically compiled. You inject (load) EPL source files into a running correlator. The correlator compiles the files into optimized byte-code representations.

The EPL compiler is strict. There is no implicit type conversion. You cannot discard return values. To minimize the chance of runtime errors, your code must be explicit and not make assumptions. The correlator terminates execution of a program at the first runtime error.

The dynamic compilation approach removes the need for a byte code interpreter that supports older versions of byte code. Also, the correlator can apply new optimization techniques during byte code generation.

About the Apama development environment in Software AG Designer

Software AG Designer provides an integrated environment for developing Apama applications. The process of developing an Apama application is centered around an Apama project. In Software AG Designer, you create a project and then you use Software AG Designer to:

- Add and manage the component files that make up the application.
- Write the EPL for your application.
- Specify the adapters and dashboards that are necessary for the application.
- Specify the configuration properties necessary for launching the application.
- Run and monitor the application.
- Export the initialization information necessary for deploying the application.
- Export your EPL files to a correlator deployment package (CDP).

See *Using Apama with Software AG Designer* for detailed information.

In addition to using Software AG Designer to create Apama projects, you can also do this using the `apama_project` command-line tool. See "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for more information.

As you add components to your application, Software AG Designer automatically generates the boilerplate EPL code for the application's standard features and launches the appropriate editor where you add the code to implement the component's behavior.

A central Apama feature in Software AG Designer is the EPL editor. The EPL editor provides support for writing EPL, for example:

- Automatic EPL validation
- Content assistance
- Auto-completion
- Hovering over an event declaration displays the event's type definition
- Automatic indenting and bracketing
- A separate panel shows the hierarchy of the EPL that appears in the editor
- Ability to define templates for frequently-used fragments of EPL

In Software AG Designer, you can examine the EPL files that are part of the Apama demo applications.

See "Overview of Developing Apama Applications" in *Using Apama with Software AG Designer* for more information.

Terminology

This topic provides a definition of each important EPL term. The definitions are organized into several groups.

Basic modules

EPL Term	Definition
Application	An Apama application consists of one or more collaborating monitors and/or one or more queries.
Package	A mechanism for qualifying monitor, query and event names. Monitors, queries and global events in the same package must each have a unique name within the package.
Context	Contexts allow EPL applications to organize work into threads that the correlator can concurrently execute.
Monitor	<p>A monitor is a basic unit of program execution. Monitors have both data and logic. Monitors communicate by sending and receiving events. A monitor is defined in a <code>.mon</code> file.</p> <p>In a monitor, you can create multiple contexts and divide processing among multiple contexts.</p> <p>A monitor cannot contain an Apama query.</p>

EPL Term	Definition
Query	<p>An Apama query is a basic unit of program execution. It partitions incoming events according to a key and then independently processes the events in each partition. Processing involves watching for an event pattern and then executing a block of procedural code when that pattern is found. A query is defined in a <code>.qry</code> file.</p> <p>In a query, you do not create contexts. Apama automatically uses multiple contexts as needed to process your query.</p> <p>An Apama query cannot contain a monitor.</p>
Channel	<p>A string name that monitor instances and receivers can subscribe to in order to receive particular events. Adapter and client configurations can specify the channel to deliver events to. In EPL, you can send an event to a specified channel.</p> <p>Queries do not subscribe to channels.</p>
Event (<i>type</i>)	<p>An event is a data object. All events have an event type and an ordered set of event fields. An event type might also have zero or more defined event actions that operate on the event fields.</p>
Field	<p>A data element of an event.</p>
Method	<p>A method is a predefined action. A given EPL type has a given set of methods that it supports.</p>

Data types

EPL Term	Definition
Data type	<p>Usually referred to as simply <i>type</i>. EPL supports the following value types: <code>boolean</code>, <code>decimal</code>, <code>float</code>, <code>integer</code>, and the following reference types: <code>action</code>, <code>Channel</code>, <code>chunk</code>, <code>context</code>, <code>dictionary</code>, <code>event</code>, <code>Exception</code>, <code>listener</code>, <code>location</code>, <code>optional</code>, <code>sequence</code>, <code>StackTraceElement</code>, <code>stream</code>, <code>string</code>. Also, <code>monitor</code> is a very limited pseudo-type.</p>
sequence	<p>An EPL type used to hold an ordered set of objects (referenced by position).</p>
dictionary	<p>An EPL type used to hold a keyed set of objects (referenced by key).</p>
optional	<p>An EPL type used to hold either zero elements or one element.</p>
location	<p>An EPL type that represents a rectangular area in a two-dimensional unitless Cartesian coordinate plane.</p>
chunk	<p>An EPL type that references an opaque data set, the data items of which are manipulated only in an EPL plug-in.</p>

EPL Term	Definition
listener	You can assign an event listener or a stream listener to a variable of this type and then subsequently call <code>quit()</code> on the listener to remove the listener from the correlator.
action	An EPL type that references an action. Actions in EPL are the equivalent of methods in object-oriented languages. Actions are user-defined methods that you can define in monitor and query definitions, event type definitions, and custom aggregate function definitions.
context	An EPL type that provides a reference to a context. A context lets the correlator concurrently process events.
stream	An EPL type that refers to a stream object. Each stream is a conduit through which items flow. A stream transports items of only one type, which can be any Apama type. Streams are internal to a monitor.
Channel	An EPL type that contains a string or a context. A contained string is the name of a channel. A contained context lets you send an event to that context. Defined in the <code>com.apama</code> namespace.
Exception	Values of <code>Exception</code> type are objects that contain information about runtime errors. Defined in the <code>com.apama</code> namespace.
StackTrace Element	A <code>StackTraceElement</code> type value is an object that contains information about one entry in the stack trace.

Monitors

EPL Term	Definition
Monitor name	Each monitor has a name that can be used to delete the monitor from the correlator.
Monitor definition	The set of source statements that define a monitor.
Monitor instance	A monitor instance is created whenever a monitor is loaded into the correlator. Subsequent monitor instances are created whenever a monitor instance spawns. As one time, a monitor instance was referred to as a sub-monitor.
Sub-monitor	A monitor instance was previously referred to as a sub-monitor.

Queries

See also [“Query terminology” on page 63](#).

EPL Term	Definition
Query name	Each Apama query has a name that can be used to delete the query from the correlator.
Query definition	The set of source statements that define an Apama query.
Query instance	A query instance is created whenever a non-parameterized query is loaded into the correlator. When a parameterized query is loaded, no instances are created until parameter values are provided. After specification of parameter values, Apama creates an instance of the query, which is referred to as a parameterization. A query definition supports multiple parameterizations.
Query key	A query key identifies one or more fields in the event types that the query specifies as input event types. Each query input event type must specify the same key.
Query partition	A partition contains a set of events that all have the same key value. One or more windows contain the events added to each partition.

Events

EPL Term	Definition
Event name	Every event must identify its event type. Event types are identified by a unique event name. The event name can also be used to remove the event definition from the correlator.
Event definition	The set of source statements that define an event type.
Event type	All events of a given event type have the same structure. An event type defines the event name, the ordered set of event fields and the set of event actions that can be called on the event fields.
Event field	A data element of an event.
Event action	An action defined within an event definition. The action can operate only on the fields of the event and any arguments passed into the action call.

Listeners

EPL Term	Definition
Event listener	A construct that monitors the events passed to, or routed within, a correlator context. When the event pattern matches the event pattern specified in an event listener, the correlator invokes the event listener's code block.

EPL Term	Definition
	In monitors, it is up to you to define event listeners. In queries, Apama defines event listeners for you.
on statement	EPL statement that defines an event listener. An on statement specifies an event expression and a listener action.
Stream listener	A construct that continuously watches for items from a stream and invokes the listener code block each time new items are available.
from statement	EPL statement that defines a stream listener. A from statement specifies a source stream, a variable, and a code block. The from statement coassigns each stream output item to the specified variable and executes the statement or block once for each output item.
Listener action	The action, statement or block part of a listener.
Listener handle	It is possible to assign the handle (reference) to a listener to a listener variable. This variable can then be used to quit the listener.
Event template	Specifies an event type and the set of (or set of ranges of) event field values to match.
Event operator	Relational, logical, or temporal operator that applies to an event template and that you specify in an event expression.
Event expression	An expression, constructed using event operators and event templates, that identifies an event or pattern of events to match.

Streams

See also the above definitions for the `stream` data type, stream listener, and the `from` statement.

EPL Term	Definition
Stream query	<p>A stream query is defined in a monitor. A stream query is a query that the correlator applies continuously to one or two streams. The output of a stream query is one continuous stream of derived items.</p> <p>A stream query is a completely different construct than an Apama query.</p>
Stream source template	An event template preceded by the <code>all</code> keyword. It uses no other event operators. A stream source template creates a stream that contains events that match the event template.
Stream network	Network of stream source templates, streams, stream queries, and stream listeners. Upstream elements feed into downstream elements to generate derived, added-value items.

EPL Term	Definition
Activation	When the passage of time or the arrival of an item causes a stream network or an element in a stream network to process items.

Defining event types

Conceptually, an event is an occurrence of a particular item of interest at a specific time. Examples of events include:

- A price of \$100 for a share of IBM stock at noon on November 7, 2014
- Purchase of 1000 shares of IBM stock at \$80 per share at 12:01 PM on December 12, 2014
- RFID tag 123-456-789 was scanned at 10:05 AM at loading dock 3
- Purchase order 55555 for 10,000 widgets sent to Acme Motor Supply
- TCP/IP address 123.4.56.789 just accessed server 5
- Container X was overfilled greater than 0.2 grams more than standard amount

An event usually corresponds to a message of some form. The correlator is designed to take in huge numbers of messages per second, and sift them for the events or patterns of events of interest. When the correlator detects interesting events or patterns it can undertake a variety of actions.

A correlator can receive events in several ways:

- You use Software AG Designer to send events from a file.
- From an adapter that receives an event from an external source. Apama adapters translate events from non-Apama format to Apama format.
- You run the Apama `engine_send` utility to manually send events into the correlator.
- A monitor or query generates an event within the correlator.
- You can write an application in C, C++, Java, or .NET that uses the Apama client API to send events into the correlator.

The correlator propagates information by sending events.

In EPL, each event is of a specific type. An event type has a name and a particular set of fields. Each field has a name and is one of a selection of types. Every event instance of a given event type has the same set and order of fields. For the correlator to process an event of a specific event type, it needs to have the event type definition for that type. Having the definition for an event type, lets the correlator

- Operate on the messages of that event type
- Create optimal indexing structures for finding events of that type that are of interest

An event type definition specifies the event type's name and the name and type of each of its fields.

See also [“Specifying named constant values” on page 249](#).

Allowable event field types

A field in an event can be any Apama type. See also [“Types” on page 583](#).

Certain field types are valid only within a certain scope and you cannot pass events with such field types outside that scope. The details are as follows:

- **context** — When an event contains a context type field, you can send the event to other monitors within the same correlator but you cannot send the event outside the correlator. In other words, you can send or route the event. See [“Generating events” on page 264](#).
- **chunk, listener and stream** — An event that contains one or more of these types of fields is valid only within the monitor that creates it. You cannot send, route, or enqueue an event that contains a field of type chunk, listener or stream.

If an event contains a chunk, listener, or stream field you cannot listen for that event.

For more information, see the description of event in the *API Reference for EPL (ApamaDoc)*.

Format for defining event types

In EPL, the format for an event type definition is as follows:

```
event event_type {
    [
        [ wildcard] field_type field_name; |
        constant field_type field_name := literal; |
        action_definition
    ] ...
}
```

Syntax description

Syntax Element	Description
event	This EPL keyword is required. It indicates an event type definition.
event_type	Replace <i>event_type</i> with a name that you choose for this event type. An EPL best practices convention is to specify an initial capital in event type names, and to capitalize subsequent words in the name. For example: <code>StockTick</code> .
{ }	Enclose the field definitions in curly braces.
wildcard	Specify the <code>wildcard</code> keyword in front of a field definition when you are certain that you will never specify that field in the match criteria for this event type. In other words,

Syntax Element	Description
	<p>when the correlator watches for certain events of this type, the value of a wildcard field is always irrelevant.</p> <p>For more details, see “Improving performance by ignoring some fields in matching events” on page 159.</p>
<i>field_type</i>	<p>Replace <i>field_type</i> with the name of a type. If you specify action, sequence, stream or dictionary, you must also specify the type of the action's argument(s) and return value if there are any, the type of the values in the sequence or stream, or the type of the dictionary's key as well as the type of the values in the dictionary. For example: <code>dictionary<integer,string></code>. For more details, see the descriptions of the dictionary and sequence types in the <i>API Reference for EPL (ApamaDoc)</i>.</p>
<i>field_name</i>	<p>Replace <i>field_name</i> with a name that you choose for this field.</p> <p>An event can have zero or more fields. You might define an event with no fields in a situation where only detection of the event is needed to start some process.</p> <p>While there is no limit to the number of fields in an event, the correlator can index up to 32 fields per event. This means that the correlator can match on up to 32 fields per event. If an event type has more than 32 fields, you must specify the <code>wildcard</code> keyword for the additional fields. Note that if the type of an event field is <code>location</code>, that field counts as 2. For example, if you have 28 non-<code>location</code> type fields and 2 <code>location</code> fields, then you have reached the limit of 32 indexed fields. If you try to inject an event definition that specifies more than 32 fields and you do not specify the <code>wildcard</code> keyword for additional fields, the correlator rejects the file. You must add the <code>wildcard</code> keywords to be able to inject the file.</p>
<i>constant</i>	<p>Specify the constant keyword in front of a field definition whose type is boolean, decimal, float, integer, or string and whose value never changes.</p>
<i>literal</i>	<p>If you specify the constant keyword, you must assign a literal to that field. The type of the literal must be the same as the <i>field_type</i> you specified for this field.</p>
<i>action_definition</i>	<p>When you specify an action in an event type definition you can call that action on an instance of the event (see “Specifying actions in event definitions” on page 253), unless it is a static action, in which case you can instead</p>

Syntax Element	Description
	call it on the event type itself (see “Defining static actions” on page 262).

Example event type definition

For example, the EPL definition of an event type for simple financial stock price ticks might include the stock's name and its price:

```
event StockTick {
    string name;
    float price;
}
```

To represent a specific instance of an event, use the following form:

```
event_type (field1_value, field2_value ...)
```

For example, a `StockTick` event describing Acme's new price of 55.20 looks like this:

```
StockTick("ACME", 55.20)
```

The reading order of fields in an event type definition and in instances of that event type must always match and is always left-to-right and then top-to-bottom. That is, "ACME" is the value of the name field and 55.20 is the value of the price field.

Working with events

After you define an event type, there are built-in methods you can call on it, and there are various ways that you can make that event available to monitors and queries.

You can call a number of methods on any event type. For an overview of these methods, see the description of `event` in the *API Reference for EPL (ApamaDoc)*.

Making event type definitions available to monitors and queries

A monitor or query must have information about the type definitions of the events that it processes. You can provide this information as follows:

- Define the event type in a separate file that contains only event definitions. An event type definition file has a `.mon` extension. It is still an EPL file even though it contains only event type declarations.

You can define any number of event types in a single file. A common practice is to define the event interface to a service in a file that is separate from the implementation of that service. You might have a single event interface file and multiple implementations of services that process those event types.

An event type definition file is the only way to make event type definitions available to queries.

- Define the event type in the monitor. Only instances of that monitor can process events of that type. Also, events of that type cannot be sent into the correlator from outside. When you define an event type inside a monitor it has a fully qualified name. For example:

```
monitor Test
{
    event Example{}
}
```

The fully qualified name for the `Example` event type is `Test.Example` and the `toString()` output for the event name is `"Test.Example()"`.

- After the optional package specification, define the event type at the beginning of an EPL file that also defines monitors. All event type declarations must be before the monitor declarations. After you inject this file into the correlator, the following monitors can process events of that type:
 - All monitors that you define in the same file
 - All monitors that you inject after you inject the file that contains the event definition.

You might have a need for different event type definitions to have the same event type name. In this situation, define each event type in a different package. Remember that event types to be used by queries must be defined in event type definition files. Then, in your monitor or query, use one of the following ways to make the appropriate event type definition available. In the monitor or query:

- Specify the fully qualified name of the event type, for example:

```
com.apamax.test.Status
```

- After any package declaration and before any other declarations, specify a `using` declaration. For example:

```
using com.apamax.test.Status;
```

In your code, you can then simply refer to the `Status` event type.

Do not create EPL structures in the `com.apama` namespace. This namespace is reserved for future Apama features. If you inadvertently create an EPL structure in the `com.apama` namespace, the correlator might not flag it as an error in this release, but it might flag it as an error in a future release.

See also [“Name Precedence” on page 693](#).

An event type definition must be injected into the correlator before a monitor that processes events of that type. After you inject an event type definition into the correlator, any monitor that you inject after that can process events of that type.

During development, when you use Software AG Designer to launch a project, it ensures that files are injected in the right order. When more than one project requires the same event definition file, do one of the following:

- In each project, declare an external dependency on the common event definition file. See *"Specifying dependencies for a single-user project"* in *Using Apama with Software AG Designer*.

- Create a project that contains the common event definition file. In each project that requires these event definitions, declare a dependency on the project that contains the common event definition file. See "Specifying projects" in *Using Apama with Software AG Designer*.

Channels and input events

Adapters, Apama client applications, and tools such as the `engine_send` correlator utility send events into the correlator. Each incoming event is associated with a channel either explicitly or implicitly. An event that has a channel explicitly set is delivered on the specified channel. An event that does not have a channel explicitly set is delivered on the default channel. The default channel's name is the empty string.

An incoming event that is sent on the default channel goes to each public context. In addition, contexts can subscribe to channels of interest (see [“Subscribing to channels” on page 54](#)). An incoming event for which a channel is explicitly set goes to each context that is subscribed to its associated channel. If there are no contexts subscribed to the specified channel the event is discarded.

Any running Apama queries receive events that come in on the default channel. In addition, Apama queries run in contexts that are subscribed to receive events sent on the `com.apama.queries` channel. So queries also receive events sent on that channel.

Events sent into the correlator from, for example, clients and adapters, are not normally delivered to external receivers. However, external receivers can specify the `com.apama.input` channel in their configuration. This is a wildcard for all events coming into the correlator. Also, an external receiver can specify `com.apama.input.channel_name` to receive correlator input events that are associated with that particular channel.

When two events are sent to different channels there is no ordering guarantee. The only guarantee is that events going from the same source to the same destination on the same channel will be delivered in order. Also, if there is an external connection with, for example, an adapter or client, then the events must use the same connection for them to be delivered in the same order.

All routable event types can be sent to channels, including event types defined in monitors.

An Apama application can use Software AG's Universal Messaging message bus to deliver events on specified channels. If a correlator is configured to connect to Universal Messaging, then a channel might have a corresponding Universal Messaging channel.

See "Choosing when to use Universal Messaging channels and when to use Apama channels" in *Connecting Apama Applications to External Components*.

2 Defining Monitors

■ About monitor contents	34
■ Example of a simple monitor	37
■ Spawning monitor instances	39
■ Communication among monitor instances	45
■ About service monitors	52
■ Adding predefined annotations	52
■ Subscribing to channels	54
■ Adding service monitor bundles to your project	57
■ Utilities for operating on monitors	57

A monitor is one of the basic units of EPL program execution.

Note:

The other basic unit is a query. A monitor cannot contain a query. A query cannot contain a monitor. For information about writing queries, see [“Defining Queries” on page 59](#). For a comparison of queries and monitors, see "Architectural comparison of queries and monitors" in *Introduction to Apama*.

Monitors have both data and logic. Monitors communicate by sending and receiving events. You define a monitor in a .mon source file. When you load the .mon file into the correlator, the correlator creates an instance of the defined monitor.

A monitor instance can operate like a factory and spawn additional monitor instances. A spawned monitor instance is a duplicate of the monitor instance that spawned it except that the correlator does not clone any active listeners or stream queries. Spawning lets a single monitor instance generate multiple instances of itself. While generally, the spawned monitor instances all listen for the same event type, each one can listen for events that have different values in particular fields.

It is good practice to define monitors and events in separate files. An advantage of doing this is that queries, as well as monitors, can use the same event definitions. When you inject files into the correlator, be sure to load event type definitions before you load the monitors and/or queries that process events of those types.

The topics below provide information and instructions for defining monitors. For reference information, see [“Monitors” on page 619](#). Apama provides several sample monitor applications, which you can find in the `samples\ep1` directory of your Apama installation directory.

See also: "Overview of Developing Apama Applications" in *Using Apama with Software AG Designer* and "Overview of Deploying Apama Applications" in *Deploying and Managing Apama Applications*.

About monitor contents

A file that defines a monitor has the following form:

1. An optional package declaration
2. Followed by
 - a. Zero or more `using` declarations
 - b. Zero or more custom aggregate function definitions
 - c. Zero or more event type definitions
3. One or more monitor definitions

When you define monitors that are closely related, it is your choice whether to define them in the same file or different files.

A monitor must have information about any event types it processes. Hence, the correlator must receive and parse all of the event types used by the monitor before it is able to correctly parse the monitor itself.

A monitor can contain one or more *global variables*. A global variable declaration appears inside a monitor but outside any actions. The variable is global within the scope of the monitor.

A monitor can also contain a number of *actions*. Actions are similar to procedures. Finding an event, or pattern of events, of interest can trigger an action. You can also trigger an action by invoking it from inside another action.

Any construct that you declare inside a monitor is available only from within that monitor. In other words, its use is restricted to the scope of the monitor.

Below is a minimal monitor:

```
monitor EmptyMonitor {
    action onload() {
    }
}
```

The monitor above does not do anything; it does not register interest in any event or event pattern, it does not have variables, and it does not do anything in its single action statement. However, it does show the minimum structure of a monitor:

- It specifies the `monitor` keyword followed by the name of the monitor. In the example, the name of the monitor is `EmptyMonitor`. The name of the monitor and the name of the file that contains the monitor do not need to be the same. A single file can contain multiple monitors.
- It declares the `onload()` action. When you inject a monitor into the correlator, the correlator executes the monitor's `onload()` action. Every monitor must contain an `onload()` action. The `onload()` action is similar to the `main()` function in C/C++.

If you define two or more monitors in the same file, the correlator executes the `onload()` actions of the monitors in the order in which you define the monitors. If there is an `onload()` action whose execution is dependent on the results of the execution of the `onload()` action of another monitor, but sure you define that other monitor earlier in the same file. If you define that other monitor in a separate file, be sure you inject that file first. Tip: it is better to avoid these dependencies as much as possible by using initialization events. See [“Using events to control processing” on page 51](#).

EPL provides a number of actions, such as `onload()`, `onunload()`, and `ondie()`. You can define additional actions, and assign a name of your choice that is not an EPL keyword. See also [“Keywords” on page 687](#).

Do not create EPL structures in the `com.apama` namespace. This namespace is reserved for future Apama features. If you do inadvertently create an EPL structure in the `com.apama` namespace, the correlator might not flag it as an error in this release, but it might flag it as an error in a future release.

Loading monitors into the correlator

During development, you use Software AG Designer to load your project, including monitors, into the correlator. Software AG Designer ensures that files are loaded in the required order.

At any time, you can use the `engine_inject` correlator tool to load EPL files into the correlator. See “Injecting code into a correlator” in *Deploying and Managing Apama Applications*.

In a deployment environment, you can load monitors into the correlator in any of the following ways:

- Use the `engine_inject` tool.
- Write a program in C++, Java, or .NET and use the corresponding Apama client API.

If you try to inject a monitor whose name is the same as a monitor that was already injected, the correlator rejects the monitor. You can inject two monitors with the same name into the correlator only if they exist in different packages. To specify the package for a monitor or event type, add a package statement as the first line in the EPL file that contains the monitor/event definition. For example:

```
package com.mycompany.mypackage;
monitor Foo {
    ...
}
```

Terminating monitors

A monitor instance terminates when one of the following events occurs:

- The monitor instance executes a `die` statement in one of its actions.
- A runtime error condition is raised.
- The monitor is terminated externally (for example, with the `engine_delete` utility). When the correlator deletes a monitor it terminates all instances of that monitor.
- The monitor instance has executed all its code and there are no active event or stream listeners. This will occur rapidly if the monitor's `onload()` action does not create any listeners. See also [“Beware of accidental stream leaks” on page 243](#).

When a monitor instance terminates, the correlator invokes the monitor's `ondie()` action, if it is defined. You cannot spawn in an `ondie()` action.

Unloading monitors from the correlator

The correlator unloads a monitor in the following situations:

- All of the monitor's instances have terminated.
- An external request kills the monitor. This kills any instances of the monitor.

If the monitor defines an `onunload()` action, the correlator executes it just before it unloads the monitor. You cannot spawn in an `onunload()` action.

If an *owning* monitor has an internal event type, it is possible for another *dependent* monitor to hold an instance of that internal event type in a variable of the any type (see the description of the any type in the *API Reference for EPL (ApamaDoc)*) if the owning monitor has sent or routed an instance of the monitor-internal event. In this case, a monitor is not completely unloaded, even if all of its monitors have terminated, because another monitor still depends on one of the monitor-internal types. The monitor name will stay in the correlator, but there will be no monitor instances running.

The `onunload()` action, if defined, still executes when the last monitor instance is terminated. The monitor is not automatically deleted in this case. The monitor name needs to be explicitly deleted with the `engine_delete` tool or by using the client API, which can only be done if the monitors that are dependent on the internal type are either no longer dependent or have been deleted themselves. See also "Deleting code from a correlator" in *Deploying and Managing Apama Applications*.

Example of a simple monitor

The empty monitor discussed in [“About monitor contents” on page 34](#) does not do anything. To write a useful monitor, add the following:

- An event type definition
- A global variable declaration
- An event expression that indicates the pattern to monitor for
- An action that operates on an event that matches the specified pattern

For example, the EPL below

- Defines the `StockTick` event type, which is the event type that the monitor is interested in.
- Defines the `newTick` global variable, which is accessible by all actions within this monitor. The `newTick` variable can hold a `StockTick` event.
- Registers an interest in all `StockTick` events.
- Invokes the `processTick()` action when it finds a `StockTick` event. The `processTick()` action uses the `log` statement to output the name and price of all `StockTick` events received by the correlator.

Lines starting with `//` are comments. EPL also supports the standard C++/Java `/* ... */` multi-line comment syntax.

```
// Definition of the event type that the correlator will receive.
// These events represent stock ticks from a market data feed.
event StockTick {
    string name;
    float price;
}

// A simple monitor follows.
monitor SimpleShareSearch {
    // The following is a global variable for storing the latest
    // StockTick event.
    StockTick newTick;
    // The correlator executes the onload() action when you inject the
    // monitor.
    action onload() {
        on all StockTick(*,*) : newTick processTick();
    }
    // The processTick() action logs the received StockTick event.
    action processTick() {
        log "StockTick event received" +
```

```
        " name = " + newTick.name +  
        " Price = " + newTick.price.toString() at INFO;  
    }  
}
```

About the variable in the example

The single global variable is of the event type `StockTick`. A variable can be of any primitive type (boolean, decimal, float, integer, string), or any reference type (action, context, dictionary, event, listener, location, sequence or stream).

About the `onload()` action

In this example, the `onload()` action contains only one line of code:

```
on all StockTick(*,*) : newTick processTick();
```

This line specifies the following:

- `on all StockTick(*,*)` indicates the event to look for.

The `on` statement begins the definition of an *event listener*. It means, “when the following event (or a pattern of events) is received ...”. This event listener is looking for all `StockTick` events. The asterisks indicate that the values of the `StockTick` event fields do not matter.

- `:newTick processTick();` indicates what to do when a `StockTick` event is found.

If the event listener finds a `StockTick` event, the coassignment (`:`) operator indicates that you want to copy the found event into the `newTick` global variable. The `onload()` action then invokes the `processTick()` action.

About event listeners

The `on` statement must be followed by an event expression. An event expression specifies the pattern you want to match. It can specify multiple events, but this simple example specifies a single event in its event expression. For details, see [“About event expressions and event templates” on page 144](#).

The `all` keyword extends the `on` statement to listen for all events that match the specified pattern. Without the `all` keyword, the event listener would listen for only the first matching event. In this example, without the `all` keyword, the event listener would terminate after it finds one `StockTick` event.

In the sample code, the event expression is `StockTick(*,*)`. Each event expression specifies one or more *event templates*. Each event template specifies one event that you want to listen for. The `StockTick(*,*)` event expression contains one event template.

The first part of an event template defines the type of event the event listener is looking for (in this case `StockTick`). The section in parentheses specifies filtering criteria for contents of events of the desired type. In this example, the event template sets both fields to wildcards (`*`). This declares an event listener that is interested in all `StockTick` events, regardless of content.

When an event listener finds a matching event, the listener can use the `as` operator to place the event into an implicitly declared variable only available in the scope of the listener processing block or the `:` assignment operator to place that event in a *global* or *local* variable. For example:

```
on StockTick(*,*) as newTick {  
    processTick(newTick);  
}
```

This copies a `StockTick` event into the `newTick` variable which is only in scope of the processing block. This is known as implicit coassignment.

Or:

```
on all StockTick(*,*):newTick processTick();
```

This copies a `StockTick` event into the `newTick` global variable. This is known as a variable coassignment.

Finally, the `on` statement invokes the `processTick()` action. For all received `StockTick` events, regardless of content, the sample monitor copies the matching event into the `newTick` global variable, and then invokes the `processTick()` action. For details, see [“Using global variables” on page 246](#).

About the `processTick()` action

The `processTick()` action executes the `log` statement to output some data on the registered logging device, which by default is standard output. This `log` statement is used to report some of the fields from the received event. For details, see [“Logging and printing” on page 279](#).

Accessing fields in events

EPL uses the `.` operator to access the fields of an event. You can see that the `processTick()` action uses the `.` operator to retrieve both the name (`newTick.name`) and price (`newTick.price`) fields of each event.

The `log` statement requires strings as fields, so the `processTick()` action specifies the built-in `.toString()` operation on the non-string value:

```
newTick.price.toString()
```

Spawning monitor instances

It is frequently necessary to enable a single monitor to concurrently listen for multiple kinds of the same event type. For example, you might want one monitor to listen for and process stock ticks that each have a different stock name. You accomplish this by spawning monitor instances as described in the topics below.

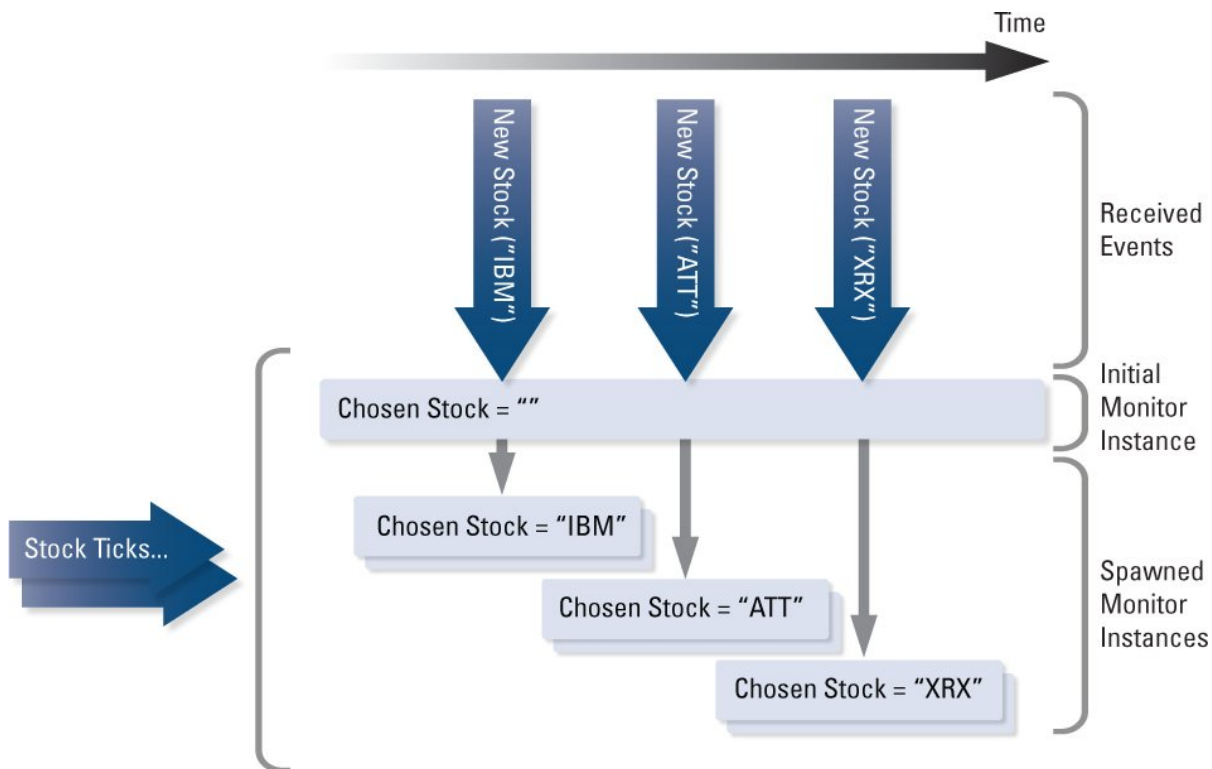
See also [“Spawning to contexts” on page 291](#).

How spawning works

In a monitor, you spawn a monitor instance by specifying the `spawn` keyword followed by an action. When the correlator spawns a monitor instance, it does the following:

1. Creates a new instance of the monitor that is spawning.
2. Copies the following, if there are any, to the new monitor instance:
 - Current values of the spawning monitor instance's global variables
 - Any arguments declared in the action that is specified in the `spawn` statement
 - Anything referred to indirectly by means of the copied variables and arguments
3. Executes the named action with the specified arguments in the new monitor instance.

The new monitor instance does not contain any active event listeners, stream listeners, streams or stream queries that were in the spawning monitor instance. For example, data held in local variables that are bound to a listener are not copied from the spawning monitor instance to the new monitor instance. The figure below illustrates this process:



The figure shows a monitor that spawns when it receives a `NewStock` event. Initially, the monitor has one active event listener. When the event listener finds the first `NewStock` event, the monitor

1. Copies the name `IBM` to the `chosenStock` variable.
2. Spawns a monitor instance.

The spawned monitor instance duplicates the initial monitor instance's state. In this example, this means that the value of the `chosenStock` variable in the spawned monitor instance is `IBM`. When the initial monitor instance receives another `NewStock` event (the value of the `name` field is `ATT`), it again copies the stock's name to the `chosenStock` variable and spawns. The same occurs for the `XRX` event, resulting in three spawned monitor instances.

Each new monitor instance starts with no active event listeners. It then creates a new event listener for `StockTick` events of the chosen stock (see the sample code in the next topic). The initial monitor instance's event listener for `NewTick` events remains active after spawning. However, because the action to create a new `StockTick` event listener is executed only in the spawned monitor instances, the initial monitor instance continues to listen for only `NewTick` events.

Sample code for spawning

EPL that implements the example described in [“How spawning works” on page 40](#) is as follows:

```
// The following event type defines a stock that a user is interested
// in. The event type includes the name of the stock (name) and the
// user's personal name (owner).
//
event NewStock {
    string name;
    string owner;
}

event StockTick {
    string name;
    float price;
}

monitor SimpleShareSearch {
    NewStock chosenStock;
    integer numberTicks;
    StockTick newTick;

    // Listen for all NewStock events. When a NewStock event is found
    // assign it to the chosenStock variable and spawn with a call to
    // the matchTicks() action. This clones the state of the monitor
    // and launches a monitor instance that executes matchTicks().
    action onload() {
        numberTicks := 0;
        on all NewStock (*, *):chosenStock spawn matchTicks();
    }

    // In the spawned monitor instance, listen for only those StockTick
    // events whose name matches the name in the chosenStock variable.
    action matchTicks() {
        on all StockTick(chosenStock.name, *):newTick processTick();
    }

    action processTick() {
        numberTicks := numberTicks + 1;
        log "A StockTick regarding the stock "
            + newTick.name + "has been received "
            + numberTicks.toString() + " times. This is relevant for "
            + " Trader name: " + chosenStock.owner
    }
}
```

```
        + " and the price is " + newTick.price.toString()  
        + "." at INFO;  
    }  
}
```

This example defines a new event type named `NewStock`. Traders dispatch this event when they want to look for a specific kind of stock event. The code example spawns a monitor instance when the monitor finds a `NewStock` event. For example, if three `NewStock` events are received by the initial monitor instance, there will be three spawned monitor instances. Other than spawning, the difference between this code sample and the sample in [“Example of a simple monitor” on page 37](#) is that this one specifies an owner in each `NewStock` event and the monitor's state now includes a counter.

In this example, after spawning, all processing is within a spawned monitor instance. Processing begins with execution of the `matchTicks` action. This action starts by defining an event listener for `StockTick` events whose `name` field matches the `name` field in the spawned monitor instance's `chosenStock` variable. When there are multiple, spawned monitor instances, each spawned monitor instance listens for only the `StockTick` events that match their `chosenStock` name.

The `numberTicks` counter variable and the `chosenStock` event variable, which contains the stock name and the owner's name, are available in the cloned state of the spawned monitor instance. This lets the `processTick()` action in each spawned monitor instance

- Customize output to include the originating trader's name
- Maintain a counter of how many `StockTicks` for a particular stock have been detected for a trader

The really important aspect that distinguishes spawning is that the entire variable space is cloned at the moment of spawning. In the example, every spawned monitor instance has a copy of the `chosenStock` variable that contains the `NewStock` event that triggered spawning. Also, every spawned monitor instance has a copy of the `numberTicks` variable, which is always set to 0 when the initial monitor instance spawns. This ensures that each spawned monitor instance can maintain an accurate count of how many matching `StockTick` events have been found.

The initial monitor instance listens for `NewStock` events. Remember that spawning does not clone active listeners, so the spawned monitor instances do not have listeners that watch for `NewStock` events. Each spawned monitor instance listens for only those `StockTick` events that contain `name` fields that match that spawned monitor instance's value for the `chosenStock` variable.

Typically, spawning is not an expensive operation. However, its overhead does increase as the size of the monitor being spawned increases. When writing an EPL application avoid repeated spawning of monitors that contain a large number of variables.

Spawned monitor instances contain copies of all global state from the spawning monitor instance. It does not matter whether the spawned monitor instance is going to use that state or not. To avoid wasting memory, a typical practice is to hold state in events that are referred to by local variables, which are not copied during spawning. This ensures that you do not have a lot of state information in global variables when the monitor instance spawns. Alternatively, you can insert code so that the new monitor instance clears unneeded state immediately after it starts running.

For information about spawning to actions that are members of events, see [“Spawning” on page 254](#).

Terminating monitor instances

The example discussed in [“Sample code for spawning” on page 41](#) spawns a monitor instance for each `NewStock` event that the initial monitor instance receives. This is not always desirable. For example, if two identical `NewStock` events are received, two identical monitor instances are spawned. To prevent this, you can use the `die` statement to delete a monitor instance if a more recent one (with the same spawning properties) has been created. For example:

```
action onload() {
    on all NewStock(*, *):chosenStock spawn matchTicks();
}
action matchTicks() {
    on NewStock (chosenStock.name, chosenStock.owner) die;
    // ...
}
```

In this fragment, the monitor spawns when it receives a `NewStock` event. In the spawned monitor instance, the initial `on` statement activates an event listener for a `NewStock` event that is identical to the one that caused the spawning. In other words, the spawned monitor instance is listening for a `NewStock` event where the fields are the same as that held by the `chosenStock` variable. If such an event arrives, the monitor instance terminates. This structure ensures that only one monitor instance for each stock name and owner exists at any one time. The same `NewStock` event kills the existing monitor instance and causes spawning of a new monitor instance. That is, the same event triggers the concurrent event listeners of the initial monitor and the spawned monitor instance.

In this solution, when a `NewStock` event kills an existing monitor instance and spawns a new monitor instance, the value of the `numberTicks` variable in the new instance is zero. Often, this kind of behavior is required. You want to ignore the state of the old monitor instance and start afresh.

Note that the event that triggers the initial monitor instance's event listener and causes the spawning of a monitor instance does not get processed by the spawned monitor instance's new event listener. An event is available to only those event listeners that are active when the correlator receives the event.

You can also use the `die` statement to kill a monitor instance at will. For example, consider the following fragments:

```
event StopStock {
    string name;
    string owner;
}

action onload() {
    on all newStock(*, *):chosenStock spawn matchTicks();
}

action matchTicks() {
    on StopStock (chosenStock.name, chosenStock.owner) die;
    // . . .
}
```

Traders would send `StopStock` events when they are no longer interested in a particular stock. Receiving a matching `StopStock` event kills the monitor instance that is listening for that stock. You can use this technique to explicitly kill any monitor instance.

About executing `ondie()` actions

A monitor instance can terminate for any of the following reasons:

- It executes all its code and has no active listeners or streaming elements.
- It executes a `die` statement in one of its actions.
- The `engine_delete` utility or an Apama client API removes the monitor from the correlator.
- A runtime error is detected in the monitor's code, which causes that instance of the monitor to die.

In all of these situations, if the monitor defines an `ondie()` action, the correlator invokes it. Like the `onload()` and `onunload()` actions, `ondie()` is a special action because the correlator invokes it automatically in certain situations.

Suppose that a monitor that defines the `ondie()` action spawns ten times, and each monitor instance dies. The correlator invokes `ondie()` eleven times: once for each spawned monitor instance, and once for the initial monitor instance. Then, just before the monitor's EPL is unloaded from the correlator, the correlator invokes the `onunload()` action only once, and it does so in the context of the last remaining monitor instance.

The correlator executes each `ondie()` operation in the context of its monitor instance. Therefore, the `ondie()` operation can access the variables in the monitor instance being terminated.

You cannot spawn in an `ondie()` or an `onunload()` action.

There are two forms of the `ondie` action: one form can have no argument and the other form can have optional `<com.apama.exceptions.Exception>` and optional `<string>` arguments. If the monitor instance terminates due to an uncaught exception, this exception is passed as a first argument to the `ondie` action. The second argument of the `ondie` action is populated with the reason if monitor instance terminated without an exception. Constants for the reason string are defined on the monitor type. See the *API Reference for EPL (ApamaDoc)* for more information.

Example:

```
action ondie(optional<com.apama.exceptions.Exception> exc, optional<string>
  reasonCode) {
    ifpresent exc {
      // do something
    }
    ifpresent reasonCode {
      if reasonCode = monitor.DIE or reasonCode = monitor.NO_LISTENERS
        or reasonCode = monitor.ENGINE_DELETE {
          //do something
        }
    }
}
```

Specifying parameters when spawning

When spawning a monitor instance, you can pass parameters to an action. For example:

```
monitor m {
  action onload() {
    spawn forward("a", "channelA");
    spawn forward("b", "channelB");
  }

  action forward(string arg, string channel) {
    on all Event(arg) as e {
      send e to channel;
    }
    on StopForwarding(arg) {
      die;
    }
  }
}
```

Communication among monitor instances

In EPL applications, everything in a monitor instance is private. There is no direct way for a monitor instance to invoke an action or access the state of another monitor instance. Instead, messages, in the form of events, are the mechanism for communication among monitor instances. All events are visible to all interested monitor instances.

Consequently, how you divide your application operations into monitors and what events the monitor instances use to communicate are crucial design decisions. An understanding of the order in which the correlator processes events for monitors helps you determine where and when to allocate events.

The topics below provide information for making these decisions.

You can use the `MemoryStore` to share state between monitors, see [“Using the MemoryStore” on page 350](#). If you are mixing monitors and queries in your application, see [“Communication between monitors and queries” on page 140](#).

Organizing behavior into monitors

Typically, an Apama application consists of several monitors each doing a specific task. For example, a simple algorithmic trading system might consist of the following monitors:

- A monitor that manages order processing by spawning a monitor instance for each order.
- One or more market data monitors. Each monitor listens for a different type of market data (such as tick data, market depth) required to process orders. Each of these monitors typically spawns a monitor instance for each stock you want to observe.

A more complex application might organize its orders into portfolios or split sets of orders into smaller orders for wave trading or some other purpose.

In an Apama application, each monitor can usually be categorized as a core processing monitor or a service monitor. A core processing monitor performs the tasks you want to accomplish. A service monitor provides data needed by the core processing monitors. Typically, the core processing monitors spawn multiple monitor instances. These monitor instances will consume data from the same service monitors. For example, all monitor instances that manage the individual orders for a given stock would obtain tick data from the same instance of a service monitor. The ordinality of the solution elements (for example, N order processors that require data from 1 tick data provider) often dictates how the solution code should be organized into separate monitors. See also [“About service monitors” on page 52](#).

The ordinality of the solution elements often dictates how the solution code should be organized into separate monitors. For example, there is an $N:1$ relationship between the N order processor monitor instances that require market data for a given stock and the 1 market data service monitor instance that supplies it.

Event processing order for monitors

As mentioned earlier, contexts allow EPL applications to organize work into threads that the correlator can execute concurrently. When you start a correlator it has a main context. In a monitor, you can create additional contexts to enable the correlator to concurrently process events.

Note:

In a query, you do not create contexts. Instead, Apama automatically creates contexts as needed to process the incoming events.

Each context, including the main context, has its own input queue, which receives

- Events sent specifically to that context from other contexts.
- Events sent to a channel that a monitor in the context is subscribed to. See [“Channels and input events” on page 31](#).

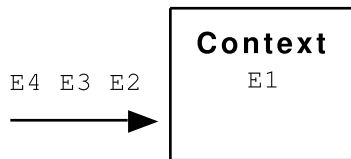
Concurrently, in each context, the correlator

- Processes events in the order in which they arrive on the context's input queue
- Completely processes one event before it moves on to process the next event

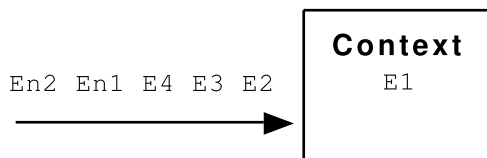
When the correlator processes an event within a given context, it is possible for that processing to route an event. A routed event goes to the front of that context's input queue. The correlator processes the routed event before it processes the other events in that input queue.

If the processing of a routed event routes one or more additional events, those additional routed events go to the front of that context's input queue. The correlator processes them before it processes any events that are already on that context's input queue.

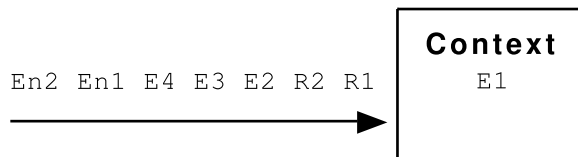
For example, suppose the correlator is processing the E1 event and events E2, E3, and E4 are on the input queue in that order.



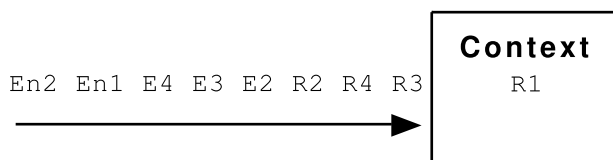
While processing E1, suppose that events En1 and En2 are created in that order or sent to the context. Assuming that there is room on the input queue of that context, those events go to the end of the input queue of that context:



While still processing E1, suppose that events R1 and R2 are created in that order and routed. These events go to the front of the queue:

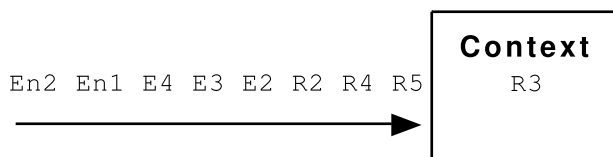


When the correlator finishes processing E1, it processes R1. While processing R1, suppose that two event listeners trigger and each event listener action routes an event. This puts event R3 and event R4 at the front of the context's input queue. The input queue now looks like this:



It is important to note that R3 and R4 are on the input queue in front of R2. The correlator processes all routed events, and any events routed from those events, and so on, before it processes the next routed or non-routed event already on that queue.

Now suppose that the correlator is done processing R1 and it begins processing R3. This processing causes R5 to be routed to the front of that context's input queue. The context's queue now looks like the following:



See also [“Understanding time in the correlator” on page 179](#).

Allocating events in monitors

Note:

The principles described here apply to variables of any type, not just to any event type or any reference type.

When writing monitors consider when and where to declare and populate event variables. You can declare event variables at the monitor level or inside an action. Event variables that you declare at the monitor level are similar to global variables.

Events are reference types. This means that, for example, a variable of event type `Foo` is not an instance of `Foo`. The variable is a reference to an instance of `Foo`.

You cannot initialize the fields of a monitor-level variable. You can, however, initialize a monitor-level instance of the event that the variable refers to. For example:

```
Foo a := Foo(1, 2.3);
```

This instantiates a `Foo` event and specifies that `a` refers to that event. Now suppose you declare the following:

```
Foo b := a;
```

This does not instantiate a new `Foo` event. It only initializes `b` as an alias for `a`.

When you declare an event at the monitor level, the correlator can automatically use default values for the event's fields. You can, but you do not have to, initialize field values. This is because the correlator implicitly transforms a statement such as this:

```
Foo a;
```

into this:

```
Foo a := new Foo;
```

Before you use a locally declared event variable in an action, you must either assign it to an existing event of the same type, or you must specify the `new` operator to create a new event to assign to the variable. Note that each event field of an event created using `new` initially has the default value for that event field type.

The following code illustrates these points:

```
event Foo
{
    integer i,
    float x;
}

monitor Bar
    Foo a; // Global (monitor-level) declaration.
           // The correlator creates a Foo event with default
           // values for fields.

    action onload() {
        a.i := 10; // Assign non-default value.
```



```

a.x := 20.0;    // Assign non-default value.
Foo b;         // Local (in an action) declaration.
               // The correlator does not create an event yet.

b := new Foo;  // Create a default Foo event and assign
               // it to local event.

b.i := 10;     // Assign a non-default value.
b.x := 20.0;   // Assign a non-default value.
Foo c := a;    // You can assign a locally declared event to
               // reference an existing event.
               // Variables a and c alias the same event.

c.i := 123     // The value of a.i is now also 123.
Foo d := Foo(15,30.0);
               // Create an event and also initialize it.
}

```

Sending events to other monitors

After you inject a monitor into the correlator, it can communicate with other injected monitors under the following conditions:

- If the source monitor instance and the target monitor instance are in the same context, the source monitor instance can route an event that the target monitor instance is listening for. A routed event goes to the front of the context's input queue. The correlator processes all routed events before it processes the next non-routed event on the context's input queue. If the processing of a routed event routes another event, that event goes to the front of the input queue and the correlator processes it before it processes any other routed events on the queue. See [“Event processing order for monitors” on page 46](#).
- If the source monitor instance and the target monitor instance are in different contexts, the source monitor instance must have a reference to the context that contains the target monitor instance. The source monitor instance can then send an event to the context that contains the target monitor instance. The target monitor instance must be listening for the sent event or the context that contains the target monitor instance must be subscribed to the channel that the event is sent on. See [“Sending an event to a particular context” on page 294](#) and [“Subscribing to channels” on page 54](#).

Within a context, an application can use routed events and completion event listeners to initiate and complete a service request inline, that is, prior to processing any subsequent events on the input queue. See [“Specifying completion event listeners” on page 157](#).

In the following example, the event listeners trigger in the order in which they are numbered.

```

monitor Client {
...
  listener_1:= on EventA() { route RequestB(...) }
  listener_5:= on ResponseForB () { doWork(); }
  listener_6:= on completed EventA() { doMoreWork(); }
...
}

monitor Service1{
...
  listener_2:= on RequestB(...)
  route RequestC();
  listener_4:= on ResponseForC{

```

```
    route ResponseForB ();
  }
  ...
}

monitor Service1a{
  ...
  listener_3:= on RequestC (...)
  route ResponseForC();
}
```

Best practices for working with routed events include:

- Keep them small; preferably zero, one, or two fields.
- Specify wildcards wherever appropriate in definitions of events that will be routed.

See also [“Generating events with the route statement” on page 264](#).

Defining your application's message exchange protocol

Monitors use events to communicate with each other. Consequently, an EPL application will have a well-defined message exchange protocol. A message exchange protocol defines the following:

- Types and structure of events that function as messages between monitor instances.
- Relationships among these events.
- Sequence and flow of events — which events are sent in response to receiving particular events.
- Which monitors need to be able to handle which events, and conversely, which monitors should not receive which events.
- Which channels these events are sent to, or whether they are sent directly between contexts.

When you define your application's message exchange protocol, keep in mind that any event that the correlator processes is potentially available to all loaded monitors. Consequently, you want to follow conventions that prevent the inadvertent matching of events with event listeners. These conventions are:

- Use packages to restrict the scope of event names (for example, `MyPackage`, `YourPackage`).
- Use duplicate event definitions with different event names (for example, `MyStartEvent`, `YourStartEvent`).
- Use discriminating/addressing information in the event (for example, `Request{integer senderId;...}`, `Response { integer toSender;...}`).

While event definitions provide partial support for a robust message exchange protocol, they lack the ability to specify event patterns, request-response associations, and so on. You should insert structured comments in your event definition files to define this part of the message exchange protocol. The comments that describe the relationships among the events define the contract that the participating monitors must adhere to. It is up to you to document the expected flows and patterns and to ensure that your monitors comply with the contract.

Some common message exchange patterns are:

- Request/response
- Publish/subscribe/unsubscribe
- Start/stop

To identify the event types that a core monitor needs to support, consider the following:

- What actions do you want to perform on the object that the monitor represents? You might want to define an event that is dedicated to each action. For example, for an order processing monitor, you might define an event type for each of the following actions:
 - Place an order
 - Change an order
 - Cancel an order
 - Suspend trading
 - Resume trading
- What initialization and termination events are needed? Keep in mind that a core monitor is typically a factory that creates monitor instances that each represent a single entity. You probably want to define at least one event type for initialization and one event type for termination.
- Do you need other control events? For example, in the order processing example, do you need a control event that suspends all trading and applies to all orders? See [“Using events to control processing” on page 51](#).
- Do you need to add any events to observe what is happening in the monitor? For example, each order processing monitor could support a request/response protocol to inquire of its state or it could simply send an `OrderProcessingState` event each time there is a significant state change.

Using events to control processing

In addition to using events to share data, you can use events to control processing. Control events are like switches. You use them to move a monitor from one state to another. Control events typically contain little or no data; that is, they have one or no fields.

A common use for control events is to initialize or terminate a process. For example, rather than use an `onload()` statement to set things up, it is good practice to use a monitor's `onload()` statement to create an event listener for a start event. This practice defers initialization until the start event is received. Similarly, you can use a stop event to signal to a monitor that it should perform shutdown actions such as deallocating resources before you terminate the correlator.

For example, consider the following action:

```
action initialize() {
    on EndAuction() and not BeginAuction() startNormalProcessing();
}
```

```
on BeginAuction() and not EndAuction() startAuctionProcessing();
route RequestAuctionState(); //A service monitor will respond with
                             //an EndAuction or BeginAuction event
}
```

In this code, `EndAuction` and `BeginAuction` can be viewed as control events. Receipt of one of these events determines whether the monitor executes the logic associated with being in an auction or out of an auction.

About service monitors

Of course, all monitors can be considered to be providing some kind of service. However, as mentioned earlier, it can be helpful to view the monitors that make up your application as either core processing monitors or service monitors. It is common for a single instance of a service monitor to provide data to a set of monitor instances spawned from a core processing monitor instance.

Apama provides a number of service monitors that fit this pattern. These service monitors provide support for the following:

Name	Description
DataView Service	Exposes read-only data to dashboards. This data comes from EPL and Java applications. See “Making Application Data Available to Clients” on page 409 .
Password Service	Supports retrieval of passwords from implementation-specific providers.
Scenario Service	Provides support for interacting with the application using the Scenario Service API. See "Scenario Service API" in <i>Connecting Apama Applications to External Components</i> .

In addition, there are a number of service monitors for use by adapters:

Name	Description
ADBC Adapter	Provides event capture and playback in conjunction with Apama's Data Player in Software AG Designer. Also monitors Java database connectivity (JDBC) and open database connectivity (ODBC).
IAF Status Manager	Monitors connectivity with an adapter.

Adding predefined annotations

Some EPL language elements can take predefined annotations. They provide the runtime and Software AG Designer with extra information about these language elements. These predefined annotations can appear immediately before the following:

- Event declarations
- Actions in monitors or event definitions

Annotations have packaged names like events. Thus, either their full name, or (preferably) a `using` declaration should be added to the file to allow the name to be used without having to specify its full name. Annotations are written as an at symbol (@) followed by the name of the annotation, followed by parameters in parentheses. The values used in annotation parameters must be literals. If both annotations and `ApamaDoc` are specified, the order should be: `ApamaDoc`, followed by annotations, followed by the language element that they apply to.

The following annotations are available:

Annotation	Parameters	Description
<code>SideEffectFree</code>	None	This annotation is part of the <code>com.apama.ep1</code> package and applies to action definitions. It tells the EPL compiler that this action has no side effects. When called from a log statement, the compiler is free to not call an action if it has no side effects and the log level is such that the log statement would not print anything to the log file. See “Logging and printing” on page 279 .
<code>OutOfOrder</code>	None	This annotation is part of the <code>com.apama.queries</code> package and applies to event definitions. It tells the query runtime that these events may occur out of order. See “Out of order events” on page 97 .
<code>TimeFrom</code>	string	This annotation is part of the <code>com.apama.queries</code> package and applies to event definitions. It tells the query runtime the default action name on the event definition to obtain source time from. See “Using source timestamps of events” on page 90 .
<code>Heartbeat</code>	string	This annotation is part of the <code>com.apama.queries</code> package and applies to event definitions. It tells the query runtime the default heartbeat event type to use. See “Using heartbeat events with source timestamps” on page 96 .
<code>DefaultWait</code>	string	This annotation is part of the <code>com.apama.queries</code> package and applies to event definitions. It tells the query editor in Software AG Designer the default time to wait to use. See “Using source timestamps of events” on page 90 .
<code>ExtraFieldsDict</code>	string	This annotation is part of the <code>com.softwareag.connectivity</code> package and

Annotation	Parameters	Description
		applies to event definitions. It names a field of type <code>dictionary<string,string></code> where the <code>apama.eventMap</code> connectivity host plug-in will place unmapped entries. See "Translating EPL events using the <code>apama.eventMap</code> host plug-in" in <i>Connecting Apama Applications to External Components</i> .
<code>MessageId</code>	<code>string</code>	<p>This annotation is part of the <code>com.softwareag.connectivity</code> package and applies to event definitions. It names a field of an event type that should contain the unique identifier of the connectivity plug-in message that the event came from.</p> <p>The field must be of type <code>string</code> and it can refer to a field nested in another event, for example:</p> <pre>MessageId("nestedEvent.fieldOfEvent")</pre> <p>You should not name a field that you expect to have a real value. See "Using reliable transports" and "Reliable messaging with Digital Event Services" in <i>Connecting Apama Applications to External Components</i>.</p>

Example:

```
using com.apama.epl.SideEffectFree;

monitor SomeMonitor {
    action onload() {
        on all Event() as e {
            log prettyPrint(e) at DEBUG;
        }
    }

    @SideEffectFree()
    action prettyPrint(Event e) returns string {
        return e.field1 + " : "+e.field2.toString();
    }
}
```

Subscribing to channels

Adapters and clients can specify the channel to deliver events to. In EPL, you can send an event to a specified channel. To obtain the events delivered to particular channels, monitor instances and external receivers can subscribe to those channels.

In a monitor instance, to receive events sent to a particular channel, call the `subscribe()` method on the monitor pseudo-type by using the following format:

```
monitor.subscribe(channel_name);
```

Replace `channel_name` with a string expression that indicates the name of the channel you want to subscribe to. You cannot specify a `com.apama.Channel` object that contains a string.

Call the `subscribe()` method from inside an action. Any monitor instance in any context can call `monitor.subscribe()`.

The `subscribe()` method subscribes the calling context to the specified channel. When a context is subscribed to a channel events delivered to that channel are processed by the context, and can match against any listeners in that context. This includes listeners from monitor instances other than the instance that called `subscribe()`. However, the subscription is owned by the monitor instance that called `monitor.subscribe()`. If that monitor instance terminates, then any subscriptions it owned also terminate.

A subscription ends when the monitor instance that subscribed to the channel terminates or executes `monitor.unsubscribe()`.

Whether an event is coming into the correlator or is generated inside the correlator, it is delivered to everything that is subscribed to the channel. If the target channel has no subscriptions from monitor instances nor external receivers then the event is discarded.

For example:

```
monitor pairtrade
{
    action onload()
    {
        on all PairTrade() as pt {
            spawn start_trade(pt.left, pt.right) to context(pt.toString());
        }
    }

    action start_trade(string sym1, string sym2)
    {
        monitor.subscribe("ticks-"+sym1);
        monitor.subscribe("ticks-"+sym2);
        // Next, set up listeners for sym1 and sym2.
        . . .
    }
}
```

This code spawns a monitor for each trade pair. The spawned monitor subscribes to just the ticks for the symbols passed to it. If a symbol in one pair is slow to process, any unrelated pairs of symbols are unaffected. See "Event association with a channel" in *Deploying and Managing Apama Applications*..

In a context, any number of monitor instances can subscribe to the same channel. When multiple monitors in a context require data from a channel the recommendation is for each monitor to subscribe to that channel. This ensures that the termination of one monitor does not affect the events received by other monitors. Subscriptions are reference counted. The result of multiple

subscriptions to the same channel from the same context is that each event is delivered once as long as any of the subscriptions are active. An event is not delivered once for each subscription.

Suppose that in one monitor instance you unsubscribe from a channel but another monitor instance in the same context is subscribed to that channel. In the monitor instance that unsubscribed, be sure to terminate any listeners for the events from the unsubscribed channel. Events from the unsubscribed channel continue to come in because of the subscription from the other monitor instance.

To explicitly terminate a subscription, call `monitor.unsubscribe(channel_name)`. In a given context, if you terminate the last subscription to a particular channel then the context no longer receives events from that channel. If events from the previously subscribed channel were delivered but not yet processed (they are waiting on the input queue) those events will be processed. This could include the processing of any listener matches. It is an error to unsubscribe from a channel that the calling monitor instance does not have a subscription to, and this will throw an exception.

If a monitor is going to terminate anyway there is neither requirement nor advantage to calling `unsubscribe()`. Calling `unsubscribe()` can be useful when a monitor listens to configuration data during startup but does not need to listen to it during normal processing.

Note:

The `subscribe()` and `unsubscribe()` methods are static methods on the `monitor` type. However, it is not possible to use instances of the `monitor` type. For example, there cannot be variables or event members of type `monitor`.

See also [“Channels and contexts” on page 292](#).

Apama queries cannot subscribe to channels. However, events sent on the default channel as well as events sent on the `com.apama.queries` channel are received by all running Apama queries. See [“Defining Queries” on page 59](#).

If a correlator is configured to connect to Universal Messaging, then a channel might have a corresponding Universal Messaging channel. If there is a corresponding Universal Messaging channel, the monitor is subscribed to the Universal Messaging channel. See “Choosing when to use Universal Messaging channels and when to use Apama channels” in *Connecting Apama Applications to External Components*.

About the default channel

The name of the default channel is the empty string.

Public contexts, including the main context, are always subscribed to the default channel. Contexts that Apama queries run in are also always subscribed to the default channel.

When an adapter or client that is sending events to the correlator does not specify a target channel the event goes to the default channel. There is no need for a public context to subscribe to the default channel.

Events generated by the `route` statement are not delivered to the default channel.

An adapter that is using Universal Messaging to send events cannot use the default channel. See "Configuring IAF adapters to use Universal Messaging" in *Connecting Apama Applications to External Components*.

About wildcard channels

An external receiver can be configured to listen on the `com.apama.input` channel, which is a wildcard channel for all events that come into the correlator. This can be useful for diagnostics, testing, or auditing, but it is not recommended for production. In a production environment, the recommendation is to explicitly specify the channels that the receiver should listen on.

A monitor instance cannot subscribe to `com.apama.input`.

To configure an external receiver to process all events generated in the correlator, specify that the receiver listens on the default channel (`""`). With this specification, a receiver would get all events generated by the `send...to channel` and `emit` statements regardless of the channel the event was directed to. Events generated by the `route` statement are not delivered to the default channel.

Adding service monitor bundles to your project

Depending on what your Apama application does, it might require one or more provided service monitors. Apama organizes service monitors into bundles. To use the service, you add the bundle to your Apama project in Software AG Designer. See also "Specifying the project bundles properties" in *Using Apama with Software AG Designer*.

When the bundle has been added in Software AG Designer, expand the bundle directory to see the contents. To understand exactly what each service monitor provides, open the service's EPL file. The comments in the EPL file explain the purpose of each service monitor and how to use it.

You can also write your own service monitors. Best practices for doing this include:

- Follow good engineering practices for defining message exchange protocols.
- Copy the conventions used in the Apama-provided service monitors as these monitors implement common patterns.

Utilities for operating on monitors

Apama provides the following command-line utilities for operating on monitors. For details about using these utilities, see "Correlator Utilities Reference" in *Deploying and Managing Apama Applications*.

- `engine_inject` — injects files into the correlator.
- `engine_delete` — removes items from the correlator.
- `engine_send` — sends Apama-format events to the correlator.
- `engine_receive` — lets you connect to a running correlator and receive events from that correlator.

- `engine_watch` — lets you monitor the runtime operational status of a running correlator.
- `engine_inspect` — lets you inspect the state of a running correlator.
- `engine_management` — lets you shut down a running correlator or obtain information about a running correlator. You can also use this utility to manage other types of components, such as adapters and continuous availability processes.

3 Defining Queries

■ Introduction to queries	60
■ Format of query definitions	68
■ Defining metadata in a query	70
■ Partitioning queries	71
■ Defining query input	77
■ Finding and acting on event patterns	103
■ Implementing parameterized queries	129
■ Restrictions in queries	134
■ Best practices for defining queries	135
■ Testing query execution	138
■ Communication between monitors and queries	140

A query is one of the basic units of EPL program execution.

Note:

The other basic unit is a monitor. A monitor cannot contain a query. A query cannot contain a monitor. For information about writing monitors, see [“Defining Monitors” on page 33](#). For a comparison of queries and monitors, see "Architectural comparison of queries and monitors" in *Introduction to Apama*.

Apama queries are suitable for applications where the incoming events provide information updates about a very large set of real-world entities. Apama provides several sample query applications, which you can find in the `samples\queries` directory of your Apama installation directory.

The topics below provide information and instructions for defining queries.

For reference information, see [“Queries” on page 627](#).

See also: "Using Query Designer" in *Using Apama with Software AG Designer* and "Deploying and Managing Queries" in *Deploying and Managing Apama Applications*.

Introduction to queries

An Apama query is a self-contained processing element that communicates with other queries, and with its environment, by sending and receiving events. Queries are designed to be multithreaded and to scale across machines.

You use Apama queries to find patterns within, or perform aggregations over, defined sets of events. For each pattern that is found, an associated block of procedural code is executed. Typically this results in one or more events being transmitted to other parts of the system.

Note:

If a license file cannot be found while the correlator is running, several restrictions are enforced on queries. See "Running Apama without a license file" in *Introduction to Apama*.

Example of a query

The following code provides an example of a query. This query monitors credit card transactions for a large set of credit card holders. The goal is to identify any fraudulent transactions. While this example illustrates query operation, it is not intended to be a realistic application.

```
query ImprobableWithdrawalLocations {
  parameters {
    float period;
  }
  inputs {
    Withdrawal(value>500) key cardNumber within period;
  }
  find Withdrawal as w1 -> Withdrawal as w2
  where w2.country != w1.country {
    log "Suspicious withdrawal: " + w2.toString() at INFO;
  }
}
```

Each query definition is in a separate file that has a `.qry` file name extension. The example shows several query features:

- **Parameters section**

Queries can be parameterized. When a query has no parameters, a single instance of the query is automatically created when the query is loaded into a correlator. If one or more parameters are defined for a query, when the query is loaded into a correlator, no instances are created until you define an instance and specify a set of parameter values for that instance.

- **Inputs section**

The `inputs` section identifies the events that the query will operate on, that is, the event inputs. This section contains one or more definitions. Each definition identifies the type of input event (`Withdrawal` in the example) together with details identifying which `Withdrawal` events are input, how those events are distributed, and what state, or event history, is to be held.

The query key is a fundamental concept. If a key is defined, then the incoming events are partitioned into different sets based on the value of the key. Query processing operates independently for each set/partition. In the example query, events for each `cardNumber` will be independently processed.

For each event input, the definition identifies the set of events that are current. When looking for pattern matches or evaluating aggregates, only current events are used. For each event input, the set of events that is current is referred to as the event window.

- **Find statement**

The `find` statement identifies an event pattern to be matched and defines what event processing actions are taken when a match is found. A `find` statement consists of an event pattern followed by a `find` block.

The event pattern can specify conditions that determine whether there is a match. A `where` condition specifies a Boolean expression that must evaluate to `true` for there to be a match. A `within` condition specifies that certain elements within the pattern must occur within a given time period. A `without` condition specifies an event whose presence can prevent a match.

Statements in a `find` block can send events to communicate with other queries, with monitor instances, and with external system elements in a deployment, such as adapters, correlators, or other deployed processes. Some EPL statements, such as `on`, `spawn`, `from`, and `die` are not allowed in queries.

Use cases for queries

Apama queries are useful when you want to monitor incoming events that provide information updates about a very large set of real-world entities such as credit cards, bank accounts, cell phones. Typically, you want to independently examine the set of events associated with each entity, that is, all events related to a particular credit card account, bank account, or cell phone. A query application operates on a huge number of independent sets with a relatively small number of events in each set.

One use case for Apama queries is to detect subsequent withdrawals from the same bank account but from locations that make it improbable that the withdrawals are legitimate. Very large numbers of withdrawal events would stream into your application. A query can segregate the transactions for each bank account from the transactions of any other bank account. Your query application can then check the transaction events for a particular account to determine if there have been withdrawals within, for example, a two-hour period from locations that are more than two hours apart. You can write a query application so that if it finds this situation the response is to contact the credit card holder.

Another use case is to detect repeated maximum withdrawals from the same automatic teller machine (ATM) within a short period of time. This might be due to a criminal with a stack of copied cards and identification numbers. In this case, a query can segregate events by ATMs. That is, the transactions conducted at a particular ATM would be in their own partition, separate from transactions conducted at any other ATM. Your query application can check the events in each partition to determine if, for example, there are repeated withdrawals of \$500 within one hour. If such a situation is found your query can be written to send an alert message to the local police.

Another use case for Apama queries is to offer a better data plan to new smartphone users. Large numbers of events related to cell phone customers would come into the system. Your query application can create sets of events where each set, or partition, contains the events related to one cell phone customer. When your query detects an upgrade from a flip phone to a smart phone, your application can automatically send a message to that customer that outlines a better data plan.

In summary, the characteristics of an Apama query application include:

- You want to monitor a very large number of real-world entities.
- You want to process events on a per-entity basis, for example, all events related to one credit card account.
- The data you need to retain in order to run Apama queries is either too large to fit on to a single machine or there is a requirement to place it in shared, fast-access storage (a cache) to support resilience/availability requirements.

More information about the use cases for queries can be found in "Understanding queries" in *Introduction to Apama*.

Delayed and out of order events

In many of the typical applications envisaged for Apama queries, the input events may be either delayed or out of order. For example, cars and other mobile sources of events such as smart phones and tablet computers might normally send regular streams of events, but when such devices are out of network coverage, these events will have to be batched and sent when back in range. Many older generation factory robots store events and only send periodic batches by design. And in other cases, events may be sent out of order. Television set top boxes, for example, often employ distinct channels for tuning information and diagnostics. This means that a "channel changed" event may be received before a "set top box crashed" event, and so may be thought to have caused it, even though the event in fact happened after it, and was causally unconnected.

Delayed or out of order events can create problems for the query runtime because it assumes that events should be treated as being in the order in which they are processed, and the time of each event is the correlator's time at the point the event is processed. However, provided that the input events contain a timestamp recording the time that the event was created at the source, these problems can be overcome by using the Apama queries source timestamp functionality. This allows the queries runtime to wait for specified periods before processing events, and then to process those events on the basis of their source timestamps rather than the time they were received by the correlator. (For out of order events, the Apama event definitions must have the appropriate annotation; for more information, see [“Out of order events” on page 97](#)).

Events can also be supplemented by heartbeat events with timestamps from data sources to inform the query runtime when communication with the data source is working correctly, which avoids long delays waiting for events to occur in case they are delayed.

See [“Using source timestamps of events” on page 90](#) for details on how to configure Apama queries to use source timestamps.

Query terminology

The following table defines important query terms.

Term	Description
query	A self-contained processing unit. It partitions incoming events according to a key and then independently processes the events in each partition. Processing involves watching for an event pattern and then executing a block of procedural code when that pattern is found.
input	An event type that a query operates on. An input definition specifies an event type plus details that indicate how to partition incoming events and what state, or event history, is to be held.
key	A query key identifies one or more fields in the events being operated on. Each input definition must specify the same key.
partition	A partition contains a set of events that all have the same key value. One or more windows contain the events added to each partition.
window	For each input, a window contains the events that are current. The query operates on only current events.
latest event	The latest event is the event that was most recently added to a partition.
set of current events	The events that are in the window(s) of a partition.
pattern	Specification of the event or sequence of events or aggregation that you are interested in. A pattern can include conditions and operators.
match set	A match set is the set of events that matches the specified pattern. A match set always includes the latest event.

Term	Description
parameterization	A query definition that specifies parameters is a parameterized query. An instance of a parameterized query is referred to as a parameterization.
source timestamp	The time an event occurred at its source. This may be before it is processed if there is some delay or disruption in delivering the event from the source to the query runtime. This will be data in one or more fields of an input event. Queries can use the source timestamp if an action is provided to obtain the source timestamp in the correct form. See “Using source timestamps of events” on page 90 .
heartbeat event	An event that a query uses to determine when communication with a data source is working correctly, and it has not missed any events that are delayed. With heartbeat events, received input events can be processed as they are considered definitive. Without these, the query runtime needs to wait for the input's wait time specified in the query definition to ensure it avoids missing delayed events.
definitive time	The point in time for which the query runtime has been told that it can assume it has received all the events it is going to receive. All events at or before this point in time are considered definitive and can be used to evaluate the query. This applies when using the source timestamp functionality.

Overview of query processing

When Apama executes queries, it does so in parallel, making use of multiple CPU cores as available. This is good for performance, but uses more resources on the hosts running the correlator and can, in edge cases, cause events to be processed in an order that is different from the order in which they were delivered to the correlator. To simplify testing, a serial mode is supported where events are processed in order, no matter how quickly they are sent.

Apama processes queries as follows:

1. Based on the `inputs` section of a query, the query subsystem creates listeners for the required events.
2. Running Apama queries receive events sent on the default channel and on the `com.apama.queries` channel.
3. Events matching those listeners are forwarded to the query subsystem that processes the events.
4. The events are processed in parallel. That is, multiple threads of execution are employed, thereby achieving vertical scaling on machines that have multiple cores.
5. The query subsystem must locate the relevant events for the query partition. That is, the previously encountered events that are still current according to the defined event windows

for that query. The information in the incoming event, that is, the key, is all that is required to locate these events.

6. The window contents are updated, adding the new event and discarding any events that are no longer current.
7. The system then checks the updated window contents to determine if there are any new pattern matches.
8. For each new pattern match the associated `find` block statements are executed.

In a single correlator solution, events in a particular partition are held in one or more Apama MemoryStore records. The key from the incoming event is used to locate these records. In a multi-correlator solution, the records are held in a distributed cache, accessed by means of the MemoryStore API. All of this is internal, however, you should consider timing constraints when deciding whether a query-based solution is appropriate for a given problem. See "Understanding queries" in *Introduction to Apama*.

After injecting a query into a correlator, events may be immediately sent to that query. If necessary, Apama stores these events until the query is prepared. That is, the query might be opening local/remote stores. Events are delivered when the query is ready to process them. There is no guarantee that the order in which the events arrived in the correlator is the same order in which the query processes them. See ["Event ordering in queries" on page 138](#).

When testing, either send events at a realistic event rate, with pauses in between each set of events, or use single context mode. To send events with pauses, you can place BATCH entries in the `.evt` file. See "Event timing" in *Deploying and Managing Apama Applications*.

By default, the query subsystem determines the size of the machine it is running on (the number of cores) and scales accordingly. If other services are affected by the load on the host machine, or for testing, then send one of the following events to the correlator (for example, by creating an `.evt` file in Software AG Designer and sending it as part of the Run Configuration) to configure how the correlator executes queries:

- `com.apama.queries.SetSingleContext()`
- `com.apama.queries.SetMultiContext()`

Overview of query application components

While queries can make up the central logic of an Apama deployment, deploying an Apama query application also requires event definitions, and connections to event sinks and event sources. Optionally, an Apama query application can make use of EPL plug-ins, EPL actions, and interactions with EPL monitors.

In addition to queries, the following components are required to implement a query application.

- **Event definitions.** This includes event types used by adapters or mapped from message busses (see below) or used internally within application components. Typically, event types specific to an adapter or to existing messages on a message bus would be written by those creating or configuring the adapter.

- Connections between event sources and queries and also between queries and event sinks. This is typically handled by adapters or by mapping to messages on a message bus by means of JMS. For testing, it is possible to use Software AG Designer or command line tools to send and receive messages.
- A correlator process. Several queries can share the same correlator process. Queries can be started by Ant scripts, which can be exported from an Apama project. For testing, Software AG Designer can start the queries.
- Optionally, queries can use a library of functions that you provide. These would be written in EPL and can call EPL plug-ins written in C++ or Java. Functions in such a library can be invoked from different points in a query.
- Optionally, a query can interact with monitors. See [“Communication between monitors and queries” on page 140](#).

For additional information, see "Query application architecture" in *Deploying and Managing Apama Applications*.

Writing event definitions

Event definitions are defined in Apama .mon files. When writing event type definitions be sure to consider the following:

- An inputs block in a query can specify filters on event fields of type boolean, decimal, float, integer, string or location.
- An event field to be specified as a query key must be of type boolean, decimal, float, integer, string or location.
- An event field to be specified in an inputs block, whether as a filter or a key, cannot be marked with the wildcard modifier in the event type definition.
- A where condition in a query can make use of all actions and fields of events, including members of reference types such as sequence, dictionary and other events.
- Specifying an event filter in an inputs block is very efficient because it prevents any part of the query from executing if the filter condition does not match. However, a filter in an inputs block can operate on only contiguous ranges and can compare only a single field to a constant or parameter value.

Specifying an event filter in a where condition is more expensive than specifying an event filter in an inputs block. However, a filter in a where clause can be more powerful because it can specify any EPL expression.

- A query cannot use an event that contains an action variable or fields of type chunk or listener.
- If you want to take advantage of the source timestamp functionality, be sure to add an event field that records the time of the creation of the data encapsulated in the event, and an action that returns this time in the form of a float representing the number of seconds since the epoch (midnight, 1 Jan 1970 UTC). If the time data is not in this format, you can use the TimeFormat event library to perform the relevant conversions (for further information, see [“Using the TimeFormat Event Library” on page 341](#)).

For example, consider the following event definitions:

```
event Slice {
    integer quantity;
    float price;
}
event UsableEvent {
    integer quantity;
    string username;
    wildcard string auxData;
    sequence<Slice> slices;
    action averagePrice() returns float {
        float t:=0;
        Slice s;
        for s in slices {
            t:=t+s.price;
        }
        return t/(slices.length().toFloat());
    }
}
event InternalEvent {
    action<> returns float averager;
}
```

`UsableEvent.quantity` and `UsableEvent.username` can be used in a query inputs block or in a query where condition.

`UsableEvent.auxData`, `UsableEvent.slices` and `UsableEvent.averagePrice()` can be used in where conditions but not in inputs blocks.

`InternalEvent` cannot be an input to a query because it has an action variable. However, an instance of `InternalEvent` could be used in a where condition or in triggered EPL code in a find block.

For example, the find statement in a query can be written as follows:

```
find UsableEvent as e1 and UsableEvent as e2
where e1.averagePrice() > e2.averagePrice()
and
e1.slices[0].price < e2.slices[0].price
```

Action definitions can supply helper actions such as the `averagePrice()` action above. This can be useful in both event types used by adapters and in internal event types. For example, some event types may have no members but simply be a named container for useful library actions.

To make use of EPL plug-ins written in C, C++ or Java, it is recommended to write an EPL event type or set of event types that wrap the plug-in. This provides a more consistent interface and can add type safety to the use of chunks, which are opaquely-typed C, C++ or Java objects. These EPL actions can then be called from queries, as can any EPL action.

Event sinks and sources

A typical deployment includes adapters that connect the Apama system to external sources of data or provide the means to send events out of Apama. This can include:

- Adapters hosted in the Apama IAF. See "Using the IAF" in *Connecting Apama Applications to External Components*.
- Connections to a JMS message bus with mapping of JMS messages to Apama event types. See "Using the Java Message Service (JMS)" in *Connecting Apama Applications to External Components*.
- Connections to a database by means of ADBC. See "The Database Connector IAF Adapter (ADBC)" in *Connecting Apama Applications to External Components*.
- Connections to other components using the Apama `engine_client` library. See "Developing Custom Clients" in *Connecting Apama Applications to External Components*.

For testing purposes, Software AG Designer can send / receive events from / to files, and command line tools are provided as well.

Correlator process

When developing queries in Software AG Designer, launching a configuration starts a correlator and injects queries into it by default. It is also possible to export the Apama launch configuration to an Ant script, which can be copied onto another machine such as a server to run your project on that machine.

It is possible to run multiple correlators that are configured to use the same distributed cache store. These correlators share query state. In such deployments, the recommendation is to use a JMS message queue. Typically, these deployments would use correlators on separate physical machines so a failure of one does not affect others. For testing, it is possible to run several correlators on a single machine provided a separate port number is allocated to each correlator. Take care to use the correct port number when interacting with the correlators.

Format of query definitions

A query searches for an event pattern that you specify. You define a query in a file with the extension `.qry`. Each `.qry` file contains the definition of only one query. The following sample shows the definition of a simple query that will search for a `Withdrawal` event pattern:

```
query ImprobableWithdrawalLocations {
  metadata {
    "author": "Apama",
    "version": "1"
  }
  parameters {
    float period;
  }
  inputs {
    Withdrawal() key cardNumber within (period);
  }
  find
    Withdrawal as w1 -> Withdrawal as w2
    where w2.country != w1.country {
      log "Suspicious withdrawal: " + w2.toString() at INFO;
    }
}
```

The format for a query definition is as follows:

```
query name {
  [ metadata { metadata_block } ]
  [ parameters { parameters_block } ]
  inputs { inputs_block }
  find pattern block
  [ action_definition ... ]
}
```

Syntax Element	Description
<code>query name</code>	Specify the query keyword followed by a name for your query. Like monitors and event types, the identifier you specify as the name of a query must be unique within your application.
<code>metadata</code>	The metadata section is optional. If you specify a metadata section, it must be the first section in the query. Metadata are specified as a list of key-value pairs. Both key and value must be string literals. For more information, see “Defining metadata in a query” on page 70 .
<code>parameters</code>	<p>The parameters section is optional. If you specify a parameters section, it must follow the metadata section, if there is one, and precede the inputs section. Parameters must be of the following types:</p> <ul style="list-style-type: none"> ■ integer ■ decimal ■ float ■ string ■ boolean <p>Specify one or more <code>data_type parameter_name</code> pairs. The parameter name can use any of the characters allowed for EPL identifiers (see “Identifiers” on page 687). Any parameters you specify are available throughout the rest of the query. For more information about parameters and how parameters get their values, see “Implementing parameterized queries” on page 129.</p>
<code>inputs</code>	<p>The inputs section is required and it must follow the parameters section, if there is one, and precede the <code>find</code> statement. In the inputs section, you must define at least one input. If you specify more than one input each input must be a different event type.</p> <p>The inputs section specifies the events that the query operates on. An input definition can include the keyword, <code>key</code>, followed by one or more fields in the specified event. This is the query key. The correlator uses the key to partition incoming events into separate windows. For example, the <code>cardNumber</code> key indicates that there is a separate window for the <code>Withdrawal</code> events for each card number. In other words, each</p>

Syntax Element	Description
	window can contain <code>Withdrawal</code> events associated with only one account. For details, see “Defining query input” on page 77 .
<code>find statement</code>	After the <code>inputs</code> section, you must specify a <code>find</code> statement. A <code>find</code> statement specifies the event pattern of interest and a block that contains procedural code. This code can define EPL actions you want to perform when there is a match. For more information, see “Finding and acting on event patterns” on page 103 .
<code>action_definition</code>	After the <code>find</code> statement, you can optionally specify one or more actions in the same form as in EPL monitors. An expression in a <code>find</code> statement can reference an action defined in that query. See “Defining actions in queries” on page 128 .

Defining metadata in a query

You can record information about a query in the `metadata` section. This can be, for example, the recording author, the version number, or the last modified date of a query. Once defined, metadata information about a query can be viewed in the Scenario Browser. See also "Using the Scenario Browser view" in *Using Apama with Software AG Designer*.

Format for defining query metadata

You define query metadata in the `metadata` section of a query definition. The `metadata` section is optional. If you specify a `metadata` section, it must be the first section in the query. The format for specifying the `metadata` section is as follows:

```
metadata {  
    key:value  
    [ , key:value ]...  
}
```

`key` and `value` must be string literals. Both are case-sensitive.

`value` can be a multi-line string.

`key` must be a valid EPL identifier (see [“Identifiers” on page 687](#)). Therefore, `key` must not include spaces, hyphens, dots or any other characters that are not allowed in EPL identifiers.

All `key` definitions that are contained in a single `metadata` section of a query must be unique.

It is recommended to use lowerCamelCase style for the key. The prefix “apama” should not be used for the key as it is reserved for future use.

Partitioning queries

Based on the values of selected fields in incoming events, the correlator segregates events into many separate partitions. Partitions typically relate to real-world entities that you are monitoring such as bank accounts, cell phones, or subscriptions. For example, you can specify a query that partitions `Withdrawal` events based on their account number. Each partition could contain the `Withdrawal` events for one account. Typically, a query application operates on a huge number of partitions with a relatively small number of events in each partition.

Each partition is identified by a unique key value. You specify a key definition in each input definition in the query's `inputs` block. The key definition specifies one or more fields or actions in the event type you want to monitor. The number, order and type of the key fields must be the same in each input definition in a query.

A query operates on the events in the windows in each partition independently of the other partitions.

Note:

Several restrictions are enforced on queries if a license file cannot be found while the correlator is running. See "Running Apama without a license file" in *Introduction to Apama*.

Defining query keys

At runtime, each partition is identified by a unique key value, which is the value of one or more fields or actions in the events that the query operates on.

Note:

Using a key is optional. If you do not specify a key, all events the query operates on are in one partition. Since this is an unusual use case for queries, the documentation assumes that you always choose to specify a key.

An event member that is declared as a constant cannot be used as a query key.

In a query, each input definition in the `inputs` section specifies the query key in the key definition. The key definition specifies one or more fields or actions in the event that the window will contain. For example:

```
query ImprobableWithdrawalLocations {
  inputs {
    Withdrawal() key cardNumber within (600.0);
  }
  find (Withdrawal as w1 -> Withdrawal as w2)
    where (w1.country != w2.country) {
    getAccountInfo();
    sendEmail();
  }
}
```

In this example, the definition for `Withdrawal` events specifies that the `cardNumber` field is the key. When the correlator processes a `Withdrawal` event, it adds the event to the partition identified by the `Withdrawal` event's `cardNumber` value.

Suppose the input definition in this example specifies two key fields:

```
inputs {
  Withdrawal() key cardNumber, cardType within (600.0);
}
```

Each partition is now identified by a combination of the `cardNumber` value and the `cardType` value. When you specify two or more key fields, insert a comma after each field except the last one. It is allowable to specify key fields in an order that is different than the order of the fields in the event.

When you specify more than one input in a query, each input definition must specify the same number and data type order of key fields. For example:

```
inputs {
  Withdrawal() key cardNumber within (600.0);
  AddressChange() key cardNumber retain 1;
}
```

For each input in this example, the key is the `cardNumber` field. The data type of the `cardNumber` field in the `Withdrawal` event must be the same as the data type of the `cardNumber` field in the `AddressChange` event.

Sometimes, a field in one event contains the same information as a field in another event but the two fields have different names. For example, information about the type of a card could be in the `cardType` field in `Withdrawal` events and the `typeOfCard` field in `AddressChange` events. In this situation, you must specify an alias for one of the event field names. You do this in the input definition's key definition. In the following example, `as cardType` in the second input definition specifies the alias:

```
inputs {
  Withdrawal() key cardNumber, cardType within (600.0);
  AddressChange() key cardNumber, typeOfCard as cardType retain 1;
}
```

When you specify more than one input, the key definition in each input definition must specify the same number of fields in the same order. Also, the data type of a field in one key definition must be the same as the data type of its corresponding field in every other key definition in the same `inputs` block. If the names of corresponding key fields are not the same, you must use the `as` keyword to specify an alias.

While specification of an alias for a key field name is sometimes required, it is always an option you can choose to use. For example:

```
inputs {
  Withdrawal() key number as cardNumber, cardType within (600.0);
  AddressChange() key number as cardNumber, typeOfCard as cardType retain 1;
}
```

An alias maps a field in an event to a key field. You cannot use an alias as a field of the event. For example, consider the following query:

```
query Q {
  inputs {
    A() key surname as lastName, dob as dateOfBirth retain 5;
    B() key lastName, dateOfBirth retain 4;
  }
}
```



```
    find A as a -> B as b ...
}
```

In the `find` block of this query, you can use the following

- `a.surname`, `a.dob` - Names of event fields
- `b.lastName`, `b.dateOfBirth` - Names of event fields
- `lastName`, `dateOfBirth` - Names of key fields

Defining actions as query keys

A query may also use the result of an action call on the event as the key for a query. To use an action as a query input key, you must provide the action name, parameters and an alias. The action call must always return a value (`chunk`, `listener` and `action` are invalid return types).

The following example calls the `getName()` action within the `A` event to generate the input key:

```
query Q {
  inputs {
    A() key getName() as name retain 1;
  }
  find A as a ...
}
```

The parameters passed to an input key action can include query parameters or literals, or can be blank in which case empty parenthesis must still be supplied. Passing a query parameter allows for specializing the partitions depending on the query parameter, which can reduce duplicating query code when only the input keys differ.

The following example calls the `getName()` action within the `B` event, supplying the query parameter `nameToPartition` into the action. The action can then return a different field (`firstname` or `surname`) depending on the parameters of this query instance:

```
query Q {
  parameters {
    string nameToPartition;
  }
  inputs {
    B() key getName(nameToPartition) as name retain 1;
  }
  find B as b ...
}
```

An alias must always be supplied and can be used to identify the returned value of the action call in the `find` block. For example, the alias `name` will identify the value returned from the call to `getName()` and can be used in the `find` block:

```
find B as b {
  print name;
}
```

If using multiple input events, the action return type must match the type for the key in all other inputs, and the alias must match between inputs. The following example uses both action and

field input keys; the surname field and the return from `getName()` must be the same type, and they are both mapped to the alias name:

```
query Q {
  inputs {
    A() key surname as name retain 1;
    B() key getName("Surname") as name retain 1;
  }
  find A as a -> B as b ...
}
```

We suggest that query parameters that are passed into action keys are not updated. Updating them can cause unexpected partitioning as the returned value from the action call may not be as expected.

Query partition example with one input

A key can be one event field. For example:

```
query ImprobableWithdrawalLocations {
  inputs {
    Withdrawal() key cardNumber within (600.0);
  }
  find (Withdrawal as w1 -> Withdrawal as w2)
    where (w1.country != w2.country) {
    getAccountInfo();
    sendEmail();
  }
}
```

In the previous code fragment, the key is the `cardNumber` field in the incoming `Withdrawal` event type. When a `Withdrawal` event arrives the correlator adds it to the window in the partition identified by the value of the `Withdrawal` event's `cardNumber` field. For each partition, each unique card number in this example, the correlator maintains the window and evaluates the pattern separately from every other partition.

Suppose that `cardNumber` is the first field in `Withdrawal` events. The following table shows what happens at runtime.

Incoming Event	Goes Into Window in Partition Identified by This Key Value	Window Contents
Withdrawal (12345, 50.0, ...) 12345	12345	Withdrawal (12345, 50.0, ...)
Withdrawal (24601, 60.0, ...) 24601	24601	Withdrawal (24601, 60.0, ...)
Withdrawal (12345, 10.0, ...) 12345	12345	Withdrawal (12345, 50.0, ...), Withdrawal (12345, 10.0, ...)

In the execution of this query, there is no interaction between the `Withdrawal` events for account number 12345 and the `Withdrawal` event for account number 24601.

Query partition example with multiple inputs

The following query provides an example of partitioning with two inputs. This query operates on APNR (Automatic Plate Number Recognition) events and Accident events:

```
query DetectSpeedingAccidents {
  inputs {
    APNR() key road within(150.0);
    Accident() key road within(10.0);
  }
  find APNR as checkpointA -> APNR as checkpointB -> Accident as accident
  where checkpointA.plateNumber = checkpointB.plateNumber
  and checkpointB.time - checkpointA.time < 100
  // Which indicates the car was speeding
  {
    emit NotifyPolice(accident.road, checkpointA.plateNumber);
  }
}
```

The road field in an APNR event must be the same type as the road field in an Accident event. Assuming that road is a string, each partition is identified by a unique value for that string.

Suppose the correlator processes the following events in top to bottom order and that road is the first field in each event:

```
Accident("M11")

APNR("A14", "FAB 1", ...)

APNR("A14", "BSG 75", ...)

APNR("M11", "ZC 158", ...)

APNR("A14", "BSG 75", ...)

APNR("M11", "ZC 158", ...)

APNR("A14", "FAB 1", ...)

Accident("A14")
```

The following table shows which events are in which partitions. Note that in each partition, the APNR events are in one window and the Accident events are in another window. Although the events are in separate windows, the correlator time-orders the events across all windows in a partition.

Events in Partition Identified by "M11"	Events in Partition Identified by "A14"
Accident("M11")	APNR("A14", "FAB 1", ...)
APNR("M11", "ZC 158", ...)	APNR("A14", "BSG 75", ...)
APNR("M11", "ZC 158", ...)	APNR("A14", "BSG 75", ...)
	APNR("A14", "FAB 1", ...)

Events in Partition Identified by "M11"	Events in Partition Identified by "A14"
	Accident("A14")

In each partition, the query evaluates the event pattern against the events in the windows in that partition. The query does this for each partition separately from every other partition. In this example, when the correlator adds the `Accident("A14")` event to the partition identified by "A14" the event pattern is triggered if the `where` clause in the `find` statement evaluates to `true`. The event pattern is not triggered in the partition identified by "M11".

About keys that have more than one field

A key can be made up of several event fields. For example, a `Transaction` event might contain a field that indicates the transaction source account and another field that indicates the transaction destination account. You can specify that you want to partition `Transaction` events according to each unique source/destination combination:

```
query TransactionMonitor {
  inputs {
    Transaction() key source, dest within PERIOD;
  }
  ...
}
```

In this example, there is a partition identified by the value of each `source/dest` combination. Each of the following events is added to a window in a different partition:

This Event	Is Added to the Window in the Partition Identified By
<code>Transaction(1, 100, ...)</code>	1, 100
<code>Transaction(1, 102, ...)</code>	1, 102
<code>Transaction(2, 100, ...)</code>	2, 100
<code>Transaction(2, 102, ...)</code>	2, 102

Regardless of the event pattern in the query, this query monitors the transfer of money from one specific account to another specific account. This query handles each transfer between the same two accounts separately from all other transactions.

Now suppose that there is an `Acknowledgement` event that acknowledges that a transaction has succeeded. It also has account source and account destination fields, but they are inverted when compared to the transaction event fields. That is, the source account for an acknowledgment is the destination account of the transaction. To ensure that the acknowledgments are added to the same partition as the corresponding transactions, the key definition specifies the `as` keyword:

```
inputs {
  Transaction() key source as txSource, dest as txDest within PERIOD;
  Acknowledgement() key dest as txSource, source as txDest within PERIOD
}
```

The query partitions events according to the combined values of the fields identified by `txSource` and `txDest`. The following table shows the partition that each event is added to.

This Event	Is Added to a Window in the Partition Identified By
Transaction(1, 100, ...)	1, 100
Acknowledgement(100, 1, ...)	1, 100
Transaction(1, 102, ...)	1, 102
Transaction(2, 100, ...)	2, 100
Acknowledgement(100, 2, ...)	2, 100

As you can see, a Transaction event and its Acknowledgement event are each added to a window in the same partition.

Defining query input

In a query definition, you must specify an `inputs` block that defines at least one input. The input definitions identify the events that you want the query to operate on. An input definition can specify particular content and it can also specify a number of events or a time period. For example:

```
query FraudulentWithdrawalDetection {
  inputs {
    Withdrawal(amount > 10.0)
      key cardNumber, cardType
      within 600.0;
    AddressChange()
      key cardNumber, typeOfCard as cardType
      retain 1;
  }
  find (Withdrawal as w1 -> Withdrawal as w2)
    where (w1.country != w2.country or w1.city != w2.city)
    without AddressChange as ac {
      getAccountInfo();
      if preferredContactType = "Email" {
        sendEmail();
      }
      if preferredContactType = "SMS" {
        sendSMS();
      }
    }
}
```

The previous code defines two inputs. For each input, there is an associated window of events. The first input window contains `Withdrawal` events and the second contains `AddressChange` events.

The input definition for the `Withdrawal` events specifies that each `Withdrawal` event in the window must have a value greater than 10.0 in the `amount` field. The input definition for the `AddressChange` events does not specify an event filter. Therefore, each `AddressChange` event that arrives is eligible to be in the window.

The next element in an input definition is the key definition. The key definition indicates how you want to partition the incoming events. If you define more than one input, the number, type and order of the key fields must be the same for each input. In the previous sample code, assume that the key fields for `Withdrawal` events, `cardNumber` and `cardType` are integer and string, respectively, and that the key fields for `AddressChange` events, `cardNumber` and `typeOfCard` are also integer and string, respectively. The two input keys match in number, type and order of key fields.

After the key definition, you can specify a `within` clause, a `retain` clause, or both. If you specify both, the `within` clause must be before the `retain` clause. A `within` clause specifies a period of time. Only events that arrive within that period of time are in the window. In the window that contains `Withdrawal` events, only `Withdrawal` events that have arrived in the last 10 minutes (600.0 seconds) are in the window. A `retain` clause specifies how many events can be in the window. In the window that contains `AddressChange` events, only the last `AddressChange` event that arrived can be in the window. When an `AddressChange` event arrives, if an `AddressChange` event is already in the window it is ejected.

After the duration, you can optionally specify a `with unique` clause to prevent repeated values appearing in the window. If specified, the `with unique` clause lists one or more fields or actions on the event type (action names should be followed by open and close parentheses). If there is more than one event in the window after the `within` and `retain` specifications, then all but the latest are discarded. See [“Matching only the latest event for a given field” on page 101](#).

The final, optional, element of an input definition is the specification of the event source timestamp and the associated wait period. If you expect that input events from a source will be subject to delays or may be received out of order, then you can specify a `time from` clause with the name of an action that returns a float specifying the number of seconds from the epoch (midnight, 1 Jan 1970 UTC) that the event was created. If you do this, you must also add a `wait` clause which requires a float or time literal specifying the maximum delay expected for these events. This tells the query runtime how long it must wait if it has not received any events before it can continue processing the events it has received. Both of these clauses require that the event definition must have a source timestamp recording the time of creation of the event, and a corresponding action that returns that timestamp in the form of a float representing the number of seconds since the epoch. In the example below, the query is gathering data from cars, which may be delayed if a vehicle goes out of network coverage. Accordingly, the input definitions specify that the source timestamps of the events are to be obtained from the events' `getEcuTime` actions which simply return the value of the events' `ts` float field. Further, the input definitions specify that in each case, the runtime should wait for up to 1 hour before continuing to process any events already received to allow for possible delays. For further details, see [“Using source timestamps of events” on page 90](#).

```
event CarRPM {
    string carId;
    float ts;
    float rpm;

    action getEcuTime() returns float {
        return ts;
    }
}

event CarEngineTemp {
    string carId;
    float ts;
```

```

    float temp;

    action getEcuTime() returns float {
        return ts;
    }
}

event CarEngineMisfire {
    string carId;
    float ts;

    action getEcuTime() returns float {
        return ts;
    }
}

query DetectEnginePerformanceProblems {

    inputs {
        CarEngineTemp() key carId within 1 hour time from getEcuTime wait 1 hour;
        CarRPM() key carId within 1 hour time from getEcuTime wait 1 hour;
        CarEngineMisfire() key carId within 1 hour time from getEcuTime wait 1 hour;
    }

    find CarEngineTemp as t and CarRPM as r -> wait 1 minute
    where t.temp > T_THRESHOLD
    where r.rpm > R_THRESHOLD
    without CarEngineMisfire as misfire {
        log "Possible engine performance problem" + t.toString() + r.toString();
    }
}

```

Typically, you define one to four inputs. If you define more than one input, each must be a different event type. In other words, two inputs to the same query cannot be the same event type.

Queries can share windows

All query instances that have the same input definitions share the same windows. Two queries have the same input definitions when they specify:

- the same input event types (the order can be different)
- the same keys
- the same (if any) input filters
- the same use of source timestamps - that is, the same action named in `time from` clauses (wait times may be different)
- the same use of heartbeat events

Any `wait`, `within`, `retain` or `with` unique specifications can be different.

When two query instances have the same input definitions and no parameters are used in any input filters, then all instances of those query definitions can share window data. If parameters

are used in input filters, then parameterizations with different parameter values each store data separately. This increases total storage requirements and cost of processing the queries.

If a query is already running and you inject a query that defines the same inputs or create a parameterization that defines the same inputs then the new query instance or new parameterization uses the same windows as the query that was already running. This means that events that were received before the new query was injected or before the parameterization was created can be in a match set for the new query instance or new parameterization. This can happen when an event arrives after the new query is injected or after the parameterization is created and that event completes the pattern that the new instance or parameterization is looking for.

To reduce the amount of memory storage required to run queries, you might want to adjust the input definition for a query so that it is the same as another query. For example, suppose query Q is consuming inputs A, B, and X, while query P is consuming inputs A, B, and Y. If both queries define both X and Y as inputs (as well as A and B) then they can share the same windows. This can be an advantage when there are many A and B events but comparatively few X and Y events. If many queries can be written with similar input sections then they can share windows, which can lead to very efficient use of memory.

If the reason for adding an input using source timestamps is simply in order to share window contents, then the wait time for this input can be zero to avoid unnecessary delays.

Format of input definitions

In a query definition, you define one or more inputs in the `inputs` block. The format of the `inputs` block is as follows.

```
inputs {
  event_type(event_filter)
  key query_key [within_clause] [retain_clause]
  [with_unique_clause]
  [time_from_clause wait_clause [or_clause]] ;

  [ event_type(event_filter)
  key query_key [within_clause] [retain_clause]
  [with_unique_clause]
  [time_from_clause wait_clause [or_clause]] ; ]...
}
```

Syntax Element	Description
<code>event_type</code>	Name of the event type that you want to operate on. The event type must be parsable. See “Type properties summary” on page 587 . Event type names can come from the root namespace, a using declaration, or a local package as specified in a package declaration.
<code>event_filter</code>	Optionally filter which events of this type you want to be in the window. For example, you might define the window to contain only the events whose amount field is greater than 10. The rules

Syntax Element	Description
	for what you can specify for the event filter are the same as for what you can specify in an event template in EPL. See “Event templates” on page 604 .
<i>query_key</i>	<p>Specify one or more event fields or actions. You can specify event fields of type <code>boolean</code>, <code>decimal</code>, <code>float</code>, <code>integer</code>, <code>string</code> or <code>location</code>. When an action is used as an input key, an alias must be supplied.</p> <p>The correlator uses the key to partition the events. Each partition is identified by a unique key value. One or two keys is typical. Three is unusual and rarely needed. More than three key values is discouraged.</p> <p>When you define more than one input in a query</p> <ul style="list-style-type: none"> ■ The number, type, and order of the key fields in each input definition must be the same. ■ If the names of the key fields are not the same in each input definition, you must specify aliases so that the names match. For details, see “Partitioning queries” on page 71.
<i>retain_clause</i>	<p>Optionally specify <code>retain</code> followed by an EPL integer expression that indicates how many events to hold in the window. For example, if you specify <code>retain 1</code>, only the last event that arrived that is of the specified type and that has the key value(s) associated with that partition is in the window. You must specify a <code>retain</code> clause or a <code>within</code> clause or both.</p> <p>While it is possible to retain any number of events, you must ensure that you define an input that allows a match with the event pattern specified in the corresponding <code>find</code> statement. For example, the following query never finds a match:</p> <pre> query Q { inputs { A() key k retain 3; } find A as a1 -> A as a2 -> A as a3 -> A as a4 { print a1.toString()+ " - "+a4.toString(); } } </pre>
<i>within_clause</i>	<p>Optionally specify <code>within</code> followed by a <code>float</code> expression or time literal that specifies the length of time that an event remains in the window. You must specify a <code>retain</code> clause or a <code>within</code> clause or both. See “Specifying event duration in windows” on page 83.</p>
<i>with_unique_clause</i>	<p>Optionally specify a set of secondary keys which constrains the window to only include the latest event for each value for the</p>

Syntax Element	Description
	set of keys. See “Matching only the latest event for a given field” on page 101 .
<i>time_from_clause</i>	Optionally specify <i>time from</i> followed by the name of an action that specifies how the source timestamp of the event can be obtained. The named action must be an action defined on that input event type. It must take no parameters and must return a float. This is taken to be when the event occurred, specified as seconds since the epoch. Note: You are not permitted to use the event's built-in <code>getTime()</code> method. This method returns the time when the correlator either processed or created the event, which defeats the purpose of the source timestamp functionality.
<i>wait_clause</i>	If a <i>time_from_clause</i> is provided, a <i>wait_clause</i> is required, which specifies <i>wait</i> followed by a <code>float</code> expression or time literal which specifies the maximum delay expected for events. This is how long a query will wait for events if it has not received any events. See also “Using heartbeat events with source timestamps” on page 96 and “Out of order events” on page 97 .
<i>or_clause</i>	Optionally specify a heartbeat event type which informs the query runtime when communication with the data source is not delayed. See “Using heartbeat events with source timestamps” on page 96 . This can only be specified if the <i>time_from_clause</i> and <i>wait_clause</i> are specified.

Behavior when there is more than one input

The correlator orders the events in a window according to the time it processes each event, that is, the time it adds the event to its window. When a query defines more than one input then, for each partition, the correlator maintains a single time-order for all events in all windows.

Suppose the correlator adds an event to a window and within 0.1 seconds the correlator adds a different event to the same window or to another window in the same partition. Outside a query, these two events might have the same timestamp because default correlator behavior is to increment the timestamp only every tenth of a second. In a query, however, if an event is added to a window within 0.1 seconds after another event was added to a window, the correlator assigns the second event a timestamp with enough significant digits to ensure that time order is preserved. The following code fragment shows the result of calling the `getTime()` method on two events that arrive within 0.1 seconds of each other:

```
find E as e -> F as f {  
  print e.getTime().toString(); // Yields "1365761429.1"  
  print f.getTime().toString(); // Yields "1365761429.100001"  
}
```

The order of the events is important when the event pattern in a `find` statement specifies the followed-by operator. Consider this example:

```
query Q {
  inputs {
    A() key k retain 20;
    B() key k retain 10;
  }
  find A as a -> B as b { ... }
}
```

This pattern does not trigger when the correlator adds an A event to the A window. But if there is already an A event in the A window then this pattern triggers when a B event is added to the B window.

In a partition, at any one time, it is possible for the set of windows to contain multiple sets of events that, each taken in isolation, would match the defined event pattern. In this case, the event matching policy determines which of the candidate sets triggers an action. See [“Event matching policy” on page 126](#) for a description of how the query chooses the event set that triggers an action. To illustrate event matching policy, that topic provides an example of query behavior when there is more than one window.

Specifying event duration in windows

In an input definition, you can specify an optional `within` clause that indicates the length of time that an event remains in the window. For example:

```
query FraudulentWithdrawalDetection {
  inputs {
    Withdrawal() key userId within 1 hour;
  }
  find Withdrawal as w1 -> Withdrawal as w2
  where w1.city != w2.city {
    log "Suspicious withdrawal: " + w2.toString() at INFO;
  }
  ...
}
```

In this example, a `Withdrawal` event remains in the window for 1 hour. After 1 hour in the window, an event is ejected. Each time an event is added to one of the windows in a partition, the correlator evaluates the `find` pattern for that partition. Ejection of an event from a window does not trigger pattern evaluation. There are two formats for specifying a `within` clause:

- `within time_literal`
- `within float_expression`

Parentheses in `within` clauses are allowed. The rules for specifying a time literal are:

- Specify one or more integer or float literal(s) and follow each one with a keyword that indicates a time unit.
- Time unit keywords are:
 - `day, days`

- hour, hours
- min, minute, minutes
- sec, second, seconds
- msec, millisecond, milliseconds

Outside a query, you can use these keywords as identifiers. Inside a query, you cannot use these keywords as identifiers unless you prefix them with a hash symbol (#). See also [“Keywords” on page 687](#).

- A space is required between an integer or float literal and its time unit. A space is required after a time unit if it is followed by an integer or float literal. Additional whitespace is allowed.
- If you specify more than one time unit keyword they must be in the order of decreasing size. For example, days must be before minutes.
- You need not specify all time units.
- Each time unit keyword must represent a different time unit, that is, you cannot, for example, specify both day and days.

Examples of valid time literals:

- 10 hours
- 1 days 12 hours
- 1 day 2 hours 30 minutes 4 sec
- 2 days 5 minutes
- 2.5 sec
- 10 seconds - This is equivalent to specifying the float expression 10.0.

Note:

While it is possible to define time literals using float values, for example, 3.5 days 12.5 hours 33.3 min, it is recommended that you use only integer values when the specification includes more than one time unit. For example, rather than specifying 2 days 65.75 minutes, you should specify 2 days 1 hour 5 min 45 sec.

If you open and edit a query in Apama's Query Designer in Software AG Designer, it modifies the time literal (if necessary) such that it contains only integers. Also, the allowable range of integers is 0 to 23 for hours, 0 to 59 for minutes, 0 to 59 for seconds, and 0 to 999 for milliseconds. Where necessary, the Query Designer rounds up to a whole number of milliseconds. For example, suppose you specify the following time literal in EPL code:

```
3.5 days 4 hours 27.5 minutes 1002.75 milliseconds
```

The Query Designer converts this to 3 days 16 hours 27 minutes 31 seconds 3 milliseconds. The actual Query Designer display is: **3d 16h 27m 31s 3ms**.

When you specify a float expression it indicates a number of seconds.

Consider the example at the beginning of this topic as the following events are added to their appropriate windows:

Time	Event Added to Window
10:00	Withdrawal("Dan", "London")
10:30	Withdrawal("Dan", "Dublin")
10:45	Withdrawal("Dan", "Paris")
11:15	Withdrawal("Ray", "Honolulu")
11:30	Withdrawal("Dan", "Rome")

For the partition identified by user ID `Dan`, the query evaluates the pattern at the following times:

Time	Window Contents	Matching Events
10:00	Withdrawal("Dan", "London")	
10:30	Withdrawal("Dan", "Dublin")	w1=Withdrawal("Dan", "London")
	Withdrawal("Dan", "London")	w2=Withdrawal("Dan", "Dublin")
10:45	Withdrawal("Dan", "Paris")	w1=Withdrawal("Dan", "Dublin")
	Withdrawal("Dan", "Dublin")	w2=Withdrawal("Dan", "Paris")
	Withdrawal("Dan", "London")	
11:30	Withdrawal("Dan", "Rome")	w1=Withdrawal("Dan", "Paris")
	Withdrawal("Dan", "Paris")	w2=Withdrawal("Dan", "Rome")

An event remains in its window for exactly the specified duration. For example, at 11:00, `Withdrawal("Dan", "London")` is no longer in the window and at 11:30, `Withdrawal("Dan", "Dublin")` is no longer in the window. Although the contents of the window have changed, ejection of an event does not cause evaluation of the event pattern.

At 11:15, there is no evaluation of the event pattern for the partition identified by user ID `Dan` because an event is added to a window in the partition identified by user ID `Ray`.

Using the output of another query as query input

While it is possible to have a query send an event explicitly containing all of the coassignments in the pattern or aggregates using the `%send` construct, this requires setting each field and defining an event type, which currently can only be defined in EPL. In this case, however, this event definition needs to be kept in sync with the query. If the query pattern is modified, then the query's `%send` construct and the event definition may both need to be updated. It is therefore recommended that you use the query output event to chain queries together.

Every query will automatically generate a query output event, which can be used as an input to other queries. This makes it easy to connect multiple queries together. One query may compute an aggregate such as an average over a sensor reading, while another query checks if a number of averaged readings match some condition.

The query output event

With each query definition, a query output event definition is automatically defined which is named after the query (see below) and has all of the values available to actions defined as fields.

The query output event definition comprises:

- All parameters.
- All keys (using the alias name).
- For find patterns:
 - All “positive” event coassignments (that is, excluding without events and wait nodes).
- For find every patterns (that is, a query which contains aggregates):
 - All aggregate select values.

Consider the two examples below, where we assume that the Measure event is defined as follows:

```
event Measure {  
    string deviceId;  
    integer userId;  
    float value;  
}
```

Example 1

```
query Q1 {  
    parameters {  
        float threshold;  
    }  
  
    inputs {  
        Measure() key deviceId within 5 minutes;  
        Error() key deviceId within 5 minutes;  
    }  
    find Measure:m1 -> (Measure:m2 or Error:e1)  
        without Error:err {  
    }  
}
```

Query output event definition for Q1:

```
event Q1 {  
    float threshold;  
    string deviceId;  
    Measure m1;  
    optional<Measure> m2;  
    optional<Error> e1;  
}
```

In query Q1, the parameter `threshold` of type `float`, the solitary input key `deviceId` of type `string`, and the positive coassignments `m1`, `m2` and `e1` are mapped in the query output event Q1. `m2` and `e1` are wrapped in `optional` since they might or might not contain any value (just `m2` or `e1` is enough to trigger the find pattern).

Example 2

```
package com.apamax.mypkg;
query Q2 {
  inputs {
    Measure() key deviceId, userId as id within 5 minutes;
  }
  find every Measure:m1
    select mean(m1.value):avg_value
    select last(m1.userId):last_value{
  }
}
```

Query output event definition for Q2:

```
package com.apamax.mypkg;
event Q2 {
  string deviceId;
  integer id;
  float avg_value;
  integer last_value;
}
```

In query Q2, which uses aggregates, both the keys `deviceId` of type `string` and `userId` (aliased as `id`) of type `integer` are mapped in the query output event Q2. The query contains two `select` statements both of whose values are also mapped in the query output event.

Note that the query output event definition resides in the same package as the query, and that it can be used in any EPL application in the usual way you use any external event definition.

Note:

Only the first 32 fields of indexable event types can be used in listeners. Beyond this, fields will be marked as wildcard fields. See [“Improving performance by ignoring some fields in matching events” on page 159](#) for more information on wildcards.

When is the query output event generated?

Whenever a query's `find` statement triggers, the query output event is routed, no matter whether it was triggered by an event or a timer firing. Another query can use this query output event as an input, thus allowing the “chaining” of one query to another.

For example, query Q3 below uses both Q1 and Q2 as input:

```
using com.apamax.mypkg.Q2;
query Q3 {
  inputs {
    Q1() key deviceId within 5 minutes;
    Q2() key deviceId within 5 minutes;
  }
  find Q1: q1 and Q2 : q2 {
    print "Query Q3 is triggered";
  }
}
```

```
}  
}
```

Q3 will trigger on receiving the inputs of type Q1 and Q2. Hence, Q3 will trigger when both queries Q1 and Q2 trigger.

Note:

It is recommended to use separate packages for queries (and hence the query output event) and any external event definitions defined in EPL. This makes it clear where the event definitions are and avoids name clashes.

Cycles are illegal. For example, if a query Q2 uses Q1 as input and if Q1 in turn also uses Q2 as input, any such cyclic dependency is illegal to use.

Specifying maximum number of events in windows

In an input definition, you can specify a `retain` clause that indicates how many events can be in the window. For example:

```
query FraudulentWithdrawalDetection2 {  
  inputs {  
    Withdrawal() key userId retain 3;  
  }  
  find Withdrawal as w1 -> Withdrawal as w2 where w1.city != w2.city {  
    log "Suspicious withdrawal: " + w2.toString() at INFO;  
  }  
}
```

In this query, only the most recent three `Withdrawal` events can be in the window. In other words, the window cannot contain more than three events. If only zero, one or two `Withdrawal` events with a particular key have arrived since the application was started then there would be only zero, one or two events, respectively, in the window.

The correlator evaluates the event pattern each time an event is added to the window. Suppose that at the indicated times the following events are added to the window in the partition identified by user ID `Dan`:

Time	Event Added to Window
10:00	<code>Withdrawal("Dan", "Dublin")</code>
10:10	<code>Withdrawal("Dan", "London")</code>
10:20	<code>Withdrawal("Dan", "London")</code>
10:30	<code>Withdrawal("Dan", "London")</code>
11:30	<code>Withdrawal("Dan", "Paris")</code>

For the partition identified by user ID `Dan`, the query evaluates the pattern at the following times:

Time	Window Contents	Matching Events
10:00	Withdrawal("Dan", "Dublin")	
10:10	Withdrawal("Dan", "Dublin")	w1=Withdrawal("Dan","Dublin")
	Withdrawal("Dan", "London")	w2=Withdrawal("Dan","London")
10:20	Withdrawal("Dan", "Dublin")	w1=Withdrawal("Dan","Dublin")
	Withdrawal("Dan", "London")	w2=Withdrawal("Dan","London")
	Withdrawal("Dan", "London")	
10:30	Withdrawal("Dan", "London")	
	Withdrawal("Dan", "London")	
	Withdrawal("Dan", "London")	
11:30	Withdrawal("Dan", "London")	w1=Withdrawal("Dan","London")
	Withdrawal("Dan", "London")	w2=Withdrawal("Dan","Paris")
	Withdrawal("Dan", "Paris")	

It is important to note that at 10:30, the `Withdrawal("Dan", "Dublin")` event that arrived at 10:00 is no longer in the window because the window retains three events at most and there are three `Withdrawal` events that have been added to the window more recently.

Specifying event duration and maximum number of events

In an input definition, you can specify a `within` clause that indicates how long an event can remain in the window and a `retain` clause that indicates how many events can be in the window. When you specify both a `within` clause and a `retain` clause the `within` clause must be before the `retain` clause. For example:

```
query FraudulentWithdrawalDetection3 {
  inputs {
    Withdrawal() key userId within 1 hour retain 3;
  }
  find Withdrawal as w1 -> Withdrawal as w2 where w1.city != w2.city {
    log "Suspicious withdrawal: " + w2.toString() at INFO;
  }
}
```

In this query, a `Withdrawal` event can be in the window for up to one hour and only the three most recent `Withdrawal` events, if each one arrived during the previous hour, can be in the window. In other words, the window cannot contain an event that arrived more than an hour ago and it cannot contain more than three events. If only two `Withdrawal` events arrived in the previous hour then there would be only two events in the window.

Suppose that at the indicated times the following events are added to the window in the partition identified by user ID `Dan`:

Time	Event Added to Window
10:00	Withdrawal("Dan", "Dublin")
10:10	Withdrawal("Dan", "London")
10:20	Withdrawal("Dan", "London")
10:30	Withdrawal("Dan", "London")
11:30	Withdrawal("Dan", "Paris")

For the partition identified by user ID Dan, the query evaluates the pattern at the following times:

Time	Window Contents	Matching Events
10:00	Withdrawal("Dan", "Dublin")	w1=Withdrawal("Dan", "Dublin")
		w2=Withdrawal("Dan", "London")
10:10	Withdrawal("Dan", "Dublin")	w1=Withdrawal("Dan", "Dublin")
	Withdrawal("Dan", "London")	w2=Withdrawal("Dan", "London")
10:20	Withdrawal("Dan", "Dublin")	
	Withdrawal("Dan", "London")	
	Withdrawal("Dan", "London")	
10:30	Withdrawal("Dan", "London")	
	Withdrawal("Dan", "London")	
	Withdrawal("Dan", "London")	
11:30	Withdrawal("Dan", "Paris")	

It is important to note that at 10:30 the `Withdrawal("Dan", "Dublin")` event that arrived at 10:00 is no longer in the window because the window retains three events at most and there are three `Withdrawal` events that have been added to the window more recently. Also, at 11:30 there are no `Withdrawal("Dan", "London")` events in the window as they have been ejected because more than one hour has elapsed since each one was added to the window.

Using source timestamps of events

By default, the query runtime assumes that events should be treated to be in the order in which they are processed, and the time of each event is the correlator's time at the point the event is processed. This is suitable if events are delivered reliably to the Apama correlator in a short amount of time and in order. However, if the events are delayed, accumulated into batches before being sent or delivered over unreliable networks, then it may be necessary to use the time at which an event happened at the event source, which would have to be available in the event in order for

queries to use the source timestamp. For example, a car may measure the engine's temperature, RPM and other important statistics along with a timestamp, and record these in a small computer in the car. Periodically, when the car is connected to a wireless network, the car will send this data as a batch of events. For the correct behavior of queries that make use of the time or ordering of events, the query will need to be configured to use the source timestamp.

Note:

Source timestamps are not intended to be a replacement for Xclock. They can, however, be used in conjunction with Xclock for testing purposes. Xclock is controlling the correlator's time (see [“Disabling the correlator's internal clock” on page 182](#)). Source timestamps indicate the time at which an event occurred.

In order to use the source timestamp:

- Every event which may be delayed should contain the source timestamp in some form.
- An action must be defined on the event definition, which takes no parameters and returns a float. This should calculate the source time of the event - typically the time the event occurred - based on the fields of the event. The return value of the action should specify the time in seconds since the epoch (midnight, 1 Jan 1970 UTC). If the event contains the time in seconds since the epoch (in this example, stored in a field named `sourceTime`), this can be as simple as the following:

```
action getSourceTime() returns float { return sourceTime; }
```

Otherwise, the `TimeFormat` event library can be used to help convert from time of day and date, and perform timezone conversions as necessary. For example, if the source timestamps in your events are not already in the UTC timezone, then one way to do this is to include a `timezone` field and then use the `TimeFormat` event's `parseTimeWithTimeZone` action to obtain the source time in the correct form as shown in the following event definition:

```
using com.apama.correlator.timeformat.TimeFormat;
using com.apama.queries.TimeFrom;

@TimeFrom("getSourceTime")
event E {
    integer k;
    string sourceTime;
    string timeZone;

    action getSourceTime() returns float {
        TimeFormat timeFormat := TimeFormat();
        return timeFormat.parseTimeWithTimeZone("HH:MM:SS", sourceTime,
            timeZone);
    }
}
```

See also [“Using the TimeFormat Event Library” on page 341](#).

- The event definition should have a `@TimeFrom` annotation as in the above example (see also [“Adding predefined annotations” on page 52](#)) or queries that use the event as an input must specify a `time from` clause that names the action that provides the source time. In either case, queries must always specify a maximum time to wait for the events (see below). If both are specified, the `time from` clause in the query takes precedence.

Note:

You are not permitted to use the event's built-in `getTime()` method. This method returns the time when the correlator either processed or created the event, which defeats the purpose of the source timestamp functionality.

Waiting for delayed events

If using source timestamps, we assume events may be delayed between the source time at which they occur and being processed by the Apama correlator. If no events are received by the correlator, it needs to distinguish between no events having occurred and events being delayed. If events are delayed, the query runtime will wait before evaluating the query, as it does not have a definitive view of all of the events. A query that uses source timestamps must specify the maximum wait time that a query will wait before it will process events. This is the maximum delay that the query will tolerate and the maximum delay between an event having occurred and the query processing that event. The wait time is inclusive - that is, an event delayed by exactly the value specified in the `wait` clause will be considered valid.

The maximum wait time must be specified and must be set to a reasonable value, as it can increase the number of events stored by the query runtime, and processing of the query may be delayed by up to that duration. The maximum wait time for an input may be less than or more than the within duration, but should not represent a large number of events for typical event rate for that input.

The wait time must be specified in a query using the `wait` clause in an input declaration. The `wait` clause can specify a time as a time literal (using days, hours, minutes and seconds) or as a float expression. Both the source timestamp action and `wait` clause must be specified (or neither). The source timestamp action can be specified via the `time from` clause in the query or a `@TimeFrom` annotation on the event type definition.

It is possible to mix inputs that have source times and events that do not have source times in a single query. Event inputs without a source time are equivalent to using `currentTime` (that is, the correlator's current time, see [“currentTime” on page 680](#)) as the source time, and a `wait` time of 0.

Event definitions may have an annotation defined `@DefaultWait` which specifies the default value to use for the wait time (see also [“Adding predefined annotations” on page 52](#)). This is only informational and used by the **Design** tab in Software AG Designer when editing query files as a means of setting the default wait time. The query must always specify the wait time, even if it is using the default value. Note that the editor will copy the value from the annotation, so changing the annotation will not affect existing query definitions.

Definitive time of a query event source

Given that input events may be delayed or out of order, how does the query runtime know when it is safe to process events? To answer this question, we introduce the concept of definitive time. The point in time for which the query runtime is entitled to think that it has received all the events it is going to receive is called the “definitive time”. All events at or before this point in time are considered definitive and can be used to evaluate the query. Events after the definitive time will not be processed until they become definitive (that is, the definitive time has changed so that the events are now at or before the definitive time). The query runtime will assume that no further

events will be received with a time before the definitive time, and will only evaluate events that have occurred before the definitive time.

In the case of an individual query input, the definitive time of that input is the latest of:

- The timestamp of the latest event received (unless the event definition is marked as occurring out of order, see [“Out of order events” on page 97](#)).
- The timestamp of the latest heartbeat event, if specified (see [“Using heartbeat events with source timestamps” on page 96](#)).
- The correlator's current time less the maximum wait time of a query.

The query's overall definitive time is then determined as the minimum or earliest of the definitive times for each input.

If no events (either input or heartbeat events) are received, then a query may need to wait in order to evaluate the events it has received (particularly if using the `wait` operator in the pattern, or more than one input, where some inputs have no events received).

The concept of definitive time is best explained using worked examples. Consider, first, a query with a single input event type.

```
using com.apama.queries.TimeFrom;

@TimeFrom("getSourceTime")
event E {
    integer k;
    float sourceTime;

    action getSourceTime() returns float {
        return sourceTime;
    }
}

query SingleInput {
    inputs {
        E() key k within 1 hour wait 2 hours;
    }
    find E as e1 -> E as e2 where e2.getSourceTime() - e1.getSourceTime() > 600.0
    {
        log "Time gap " + (e2.getSourceTime() - e1.getSourceTime()).toString();
    }
}
```

In this case, where there is only a single input type, the definitive time will be the latest or most recent of either: the source timestamp of the last event, or the current time minus the wait time (2 hours in this example). The following table shows how the query runtime keeps track of the definitive time as it receives input events.

Wall Time	E event source timestamp	Query definitive time	Result	Explanation
10:00	07:00	08:00		

Wall Time	E event source timestamp	Query definitive time	Result	Explanation
10:05	07:30	08:05		Nothing - events are too old.
10:10	08:30	08:30		
10:24	08:32	08:32		Nothing - event timestamps were only 2 minutes apart.
10:26	08:50	08:50	Time gap 18 minutes	
10:30	10:30	10:30		Nothing - only 1 event in the "within 1 hour" window.

Now consider a more complex case where the query has two input event types. Events of type E are defined as above, but we add another definition for events of type X.

```
@TimeFrom("getSourceTime")
event X {
    integer k;
    float sourceTime;

    action getSourceTime() returns float {
        return sourceTime;
    }
}

query MultipleInputs {

    inputs {
        E() key k within 1 hour wait 1 hour;
        X() key k within 1 hour wait 1 hour;
    }

    find E as e1 -> E as e2 without X as x {
        log "Got (" + e1.sourceTime.toString() + ", "
            + e2.sourceTime.toString() + ")";
    }
}
```

Once again the table below shows how the definitive time of the query is determined. In this case, the runtime must take the definitive time as being the earliest of the definitive times of the input types because, as the pattern depends on all input types, it is only up until that point that it has a definitive view of all the query inputs.

For example, at wall time 09:22, even though the runtime has got E events with source timestamps 08:32 and 08:40, it is not entitled to conclude that we have a match for the query pattern because

the most recent X event has a timestamp of only 08:25, so we do not yet know if there was an X event between 08:32 and 08:40 that would prevent a match. The wait time of 1 hour has not yet elapsed, so the definitive time of the query remains at 08:25, which is the source time of the most recent X event.

It is not until wall time 09:23 that we receive another X event with a source timestamp of 08:50. At this point, given that in this example we know that events are being delivered in order, it is safe for the runtime to assume that there were no other X events between 08:25 and 08:50 and so it can proceed to execute the query and match for the two pairs of E events ("08:30, 08:32" and "08:32, 08:40"). Further, at this time (wall time 09:23) the receipt of the X event with source timestamp 08:50 allows the runtime to update the definitive time of the overall query to 08:40, which has become the earliest of the definitive times of the query inputs.

Wall Time	E event source timestamp	X event source timestamp	Query definitive time	Result	Explanation
09:20	08:30	08:25	08:25		
09:21	08:32		08:25		Nothing yet. Still waiting for an X.
09:22	08:40		08:25		
09:23		08:50	08:40	Got (08:30, 08:32) Got (08:32, 08:40)	
09:24	08:55		08:50		No 08:40 - 08:55 match, there is an X at 08:50.
09:25	09:00		08:50		Nothing yet - still waiting for X after 08:50.
09:26		08:57	08:57		No 08:55 - 09:00 match, there is an X.
09:27	09:10		08:57		Nothing yet - still waiting for X after 08:57.
10:10			09:10	Got (09:00, 09:10)	We waited for 1 hour for an X.

Using heartbeat events with source timestamps

When using source timestamps, if a query's input has no events for a period of time, then the query will wait for the specified wait time for that query before evaluating events. This can cause unacceptable delays in processing events from other inputs. Some data sources may provide *heartbeat* events with timestamps which signal that communication from the data source to the queries system is working correctly. If these events occur but no input events have been received, then the query can infer that no input events, or only the input events received, have occurred, and thus the query's input is definitive upon receiving a heartbeat, without having to wait any further. If communication is disrupted or delayed, then the heartbeat events will similarly be delayed, and the query will wait, as it has to in order to process delayed events.

Heartbeat events are specified on the input event type's definition or per input of the query. They are only used if a query input is using source timestamps, that is, it has a `wait` clause specified. The heartbeat can be specified as a `@Heartbeat` annotation on the event definition, which should name the fully qualified event type to use as heartbeat events.

If a query input contains a `time from` clause, then the heartbeat must be explicitly named with an `or heartbeat-type` clause after the `wait` clause. For example, these two are equivalent:

```
@TimeFrom("getEcuTime")
@Heartbeat("CarHeartbeat")
event CarEngineTemp { .. }
...
query ... {
  inputs {
    CarEngineTemp() key carId within 1 hour wait 6 hours;
  }
  ...
}
```

or:

```
query ... {
  inputs {
    CarEngineTemp() key carId within 1 hour time from getEcuTime
    wait 6 hours or CarHeartbeat;
  }
  ...
}
```

The following rules apply for the heartbeat event:

- The heartbeat event cannot be filtered.
- The heartbeat event must share the same key fields and the same types as the input event type. In the above example, both `CarEngineTemp` and `CarHeartbeat` must have a field named `carId` which is of the same type in each event type. If actions are used in the input key, the heartbeat event must also supply the same action with the same signature (same parameters and return type).
- The heartbeat event must have a matching action for obtaining the source time. In the above example, both `CarEngineTemp` and `CarHeartbeat` must have an action of the signature `action getEcuTime() returns float`. Typically, these would have the same implementation, as the heartbeat would have source timestamps in the same form as the input events; but the

implementation of these methods may be different for heartbeat events (see [“Out of order events” on page 97.](#))

- The heartbeat event cannot be used as an input in the pattern, unless it is also listed as an input event in its own right.
- The same heartbeat event type may be used for different inputs of the same query (this is typical, as a query may use a number of different types of events from the same data source, such as a car in the above example).

When a heartbeat event is received and processed, it will step forward the definitive time for all inputs that specify that heartbeat event. Thus, if all inputs use the same heartbeat event, then that heartbeat can step forward the definitive time, allowing the query to evaluate events received on some inputs without having to wait for the input wait time on other inputs where no input events were received.

Typically, heartbeat events will be delivered regularly. The rate at which heartbeat events are sent is dependent on the data source, but the queries system must be able to handle all of the heartbeat events from all data sources as well as the input events. Some devices may only send the heartbeats under certain conditions, for example, a car may only send heartbeats if the engine is running or the car is occupied. If no heartbeat events are received, then queries will use the wait time specified in the input before evaluating any events received, as needed.

Note that queries assume that the heartbeat events are delivered in the same order as input events. If an input event arrives with a timestamp before a previous heartbeat event, it will be discarded.

Typically, heartbeat events will be events that come from the same data source as the input events they are used with. Thus, any communications disruption affecting the input events will affect the heartbeat events in the same way. This is not a requirement; if some other system has knowledge of when a data source is connected or disconnected, the heartbeat events could be sent from that system - but if the system incorrectly sends heartbeat events and input events are delayed, then input events may be discarded.

Out of order events

When using source timestamps (see also [“Using source timestamps of events” on page 90](#)), the query runtime by default expects events to arrive in order. If an event arrives with an earlier source timestamp than a previous event for that same partition, it will be discarded. However, there are two cases where this behavior does not occur (see below), and queries will store events which arrive out of order and re-order them so that when they are processed, they are processed in order according to the source time.

Note:

In both cases described below (with the `@OutOfOrder()` annotation and delayed events), heartbeat events (if specified) are always considered definitive, even if they are delayed. You cannot use an event definition with an `@OutOfOrder()` annotation as a heartbeat event. Note that as soon as a heartbeat event is processed, the query will ignore any events with earlier timestamps.

Case 1: Using the @outOfOrder() annotation on the event definition

If the event definition (in an EPL file) has the @outOfOrder() annotation which is available in the package `com.apama.queries` (see also [“Adding predefined annotations” on page 52](#)), then the queries runtime will treat it as not occurring in order.

This means that definitive time is not affected by the timestamp on the events. Thus, events will not be processed until the specified wait time has elapsed since their source time, or a heartbeat event (if specified) with a later timestamp has been processed (and all inputs have had their definitive time moved forward).

It is recommended to use heartbeats when using @outOfOrder() events. They are not required, but if not used, the query execution will be delayed by the longest input wait specified in the query.

The following example compares the behavior if @outOfOrder() is or is not specified on the input:

```
query FindAdjacentAEvents {
  inputs {
    A() within 30.0 wait 20 seconds;
  }
  find A as a1 -> A as a2 {
    print "a1 = "+a1.toString()+" "; a2 = "+a2.toString();
  }
}
```

In the following tables, the events are listed in the order in which they are processed, but they occur in the order A(1), A(2), A(3), A(4). Note that A(2) is delayed by more than the wait time of the query (the actual events would have a source timestamp, but we show that as a separate column for clarity).

The following table applies if the event definition does have @outOfOrder():

Input event	Input event timestamp	Correlator time	Notes	Query definitive time	Query output
A(1)	10:00:10	10:00:20		10:00:00	
A(4)	10:00:20	10:00:30		10:00:10	
A(3)	10:00:15	10:00:32		10:00:12	
		10:00:35	20 seconds after A(3)'s source time (10:00:15)	10:00:15	a1=A(1); a2=A(3)
A(2)	10:00:12	10:00:37	discarded - more than 20 seconds old	10:00:17	

Input event	Input event timestamp	Correlator time	Notes	Query definitive time	Query output
		10:00:40	20 seconds after A(4)'s source time (10:00:20)	10:00:20	a1=A(3); a2=A(4)

The following table applies if the event definition does *not* have `@OutOfOrder()`:

Input event	Input event timestamp	Correlator time	Notes	Query definitive time	Query output
A(1)	10:00:10	10:00:20		10:00:10	
A(4)	10:00:20	10:00:30		10:00:20	a1=A(1); a2=A(4)
A(3)	10:00:15	10:00:32		10:00:20	(nothing - event is discarded as it is out of order)
A(2)	10:00:12	10:00:37	discarded - more than 20 seconds old	10:00:20	

Case 2: Events are delayed

Even in the case where events are normally delivered in order from the data source, if there is a delay which is then resolved, a number of delayed events may all be processed in a very short space of time. Even if they are delivered to Apama correlators in the correct order, the query runtime runs in parallel within the correlator, so events processed close together in time may be processed out of order, even if they do not have an `@OutOfOrder()` annotation on the event definition. If an event is delayed, then the query runtime will wait before considering the event's time as definitive for that input.

By default, the query runtime considers an event as delayed if its source time is more than 10 seconds before the correlator's time at the point it is processed, and it will wait for 10 seconds before considering the event's time as definitive for that input. These settings can be modified by sending in a `SetDelayedEventsLeeway(delayLeeway, reorderBuffer)` event:

```
com.apama.queries.SetDelayedEventsLeeway(5, 20.0)
```

The above example would set the query runtime to consider events older than 5 seconds as delayed, and would not consider them definitive until 20 seconds after they were received.

To consider all events in order regardless of delay, send an event with the first value set to `infinity` (as all actual delays must be less than infinity):

```
com.apama.queries.SetDelayedEventsLeeway(infinity, 0.0)
```

These events should be sent to all correlators in a cluster, typically as part of the initialization of the correlator along with injecting the query definitions.

The following example compares the behavior with different configurations and some delayed events:

```
query FindAdjacentAEvents {
  inputs {
    A() within 30 minutes wait 10 minutes;
  }
  find A as a1 -> A as a2 {
    print "a1 = "+a1.toString()+" "; a2 = "+a2.toString();
  }
}
```

The following table lists the events where the A event does not have `@OutOfOrder()`. The last three columns give the behavior with different configurations:

- **Default config. A.** Matches with the default values: 10 seconds delay threshold and 10 seconds reorder buffer.
- **Config. B.** Matches if `SetDelayedEventsLeeway(300, 10)` is sent: 5 minutes (300 seconds) delay threshold and 10 seconds reorder buffer.
- **Config. C.** Matches if `SetDelayedEventsLeeway(10, 60)` is sent: 10 seconds delay threshold and 1 minute reorder buffer.

Input event	Input event timestamp	Correlator time	Definitive time of the query for default leeway values	Default config. A	Config. B	Config. C
A(1)	10:06:10	10:10:30	10:00:30 (10 minutes ago)			
A(4)	10:06:20	10:10:31	10:00:31 (10 minutes ago)		a1=A(1); a2=A(4)	
A(3)	10:06:15	10:10:32	10:00:32 (10 minutes ago)		(A(3) out of order and discarded)	
A(2)	10:06:13	10:10:33	10:00:33 (10 minutes ago)		(A(2) out of order and discarded)	

Input event	Input event timestamp	Correlator time	Definitive time of the query for default leeway values	Default config. A	Config. B	Config. C
		10:10:43	10:06:20 (latest A event received)	a1=A(1); a2=A(2) a1=A(2); a2=A(3) a1=A(3); a2=A(4)		
		10:11:33				a1=A(1); a2=A(2) a1=A(2); a2=A(3) a1=A(3); a2=A(4)
A(6)	10:12:05	10:12:10	10:12:05 (latest A event received)	a1=A(4); a2=A(6)	a1=A(4); a2=A(6)	a1=A(4); a2=A(6)
A(5)	10:12:04	10:12:11	10:12:05 (latest A event received)	(none - event A(5) is discarded)	(none - event A(5) is discarded)	(none - event A(5) is discarded)

Note that A(6) is treated as occurring in order, as it is delayed by less than the *delayLeeway* value. Thus A(5) is discarded, as it has occurred out of order.

Matching only the latest event for a given field

A query input can optionally limit the window to only contain the most recent item for each value of a given field or action of the event. This is performed by the `with unique` operator, which is followed by one or more fields or actions of the input event type.

For example, consider a query looking at sensor data from a number of sensors on the same production line, with events that specify the `productionLine` and `sensorId`. The query compares sensor values between different machines and sensors on the same production line, so the query can be keyed on the `productionLine` field of events, but not on the `sensorId` field. However, only the latest event for each sensor is required. By specifying a `with unique sensorId` clause, only the latest value of each sensor is used.

If you add a `with unique` clause, if there is more than one item in the window that has the same value for all the fields or actions listed in the `with unique` clause, then only the most recent event is considered to be in the window and can match the pattern. The suppression of duplicates occurs after the `within` and/or `retain` clauses apply. For example:

```
inputs {  
    Sensor() key productionLine retain 3 with unique sensorId;  
}
```

Given the following events, the window contains only those marked in the third column of the following table (assuming all are for the same `productionLine`):

Event	sensorId	Window contains	Notes
1	A	1(A)	
2	B	1(A), 2(B)	
3	C	1(A), 2(B), 3(C)	
4	B	3(C), 4(B)	Event 1 is discarded due to <code>retain 3</code> . Event 2 is discarded as event 4 has the same <code>sensorId</code> .
5	D	3(C), 4(B), 5(D)	
6	C	4(B), 5(D), 6(C)	Event 3 is discarded due to <code>retain 3</code> .
7	D	6(C), 7(D)	Event 4 is discarded due to <code>retain 3</code> . Event 5 is discarded as event 7 has the same <code>sensorId</code> .

Note that the `with unique` is applied after the `retain` expression. Any `with unique` expression does not affect window sharing (see also [“Queries can share windows” on page 79](#)) nor how much data is stored.

The `with unique` clause comes after the sizing of the window (`within`, `retain`) and before, if present, the `time from`, `wait or` or `or` clauses used for specifying source time.

`with unique` can list a number of comma-separated members or calls to actions, where the action name is followed by parentheses. Actions used in a `with unique` clause must take no parameters and return a value. The ordering is unimportant.

For example, using `with unique upperName()` for an event definition such as the following would only keep one event for each value of the `name` field, ignoring case:

```
event E {  
    string name;  
    action upperName() returns string { returns name.toUpperCase(); }  
}
```

Finding and acting on event patterns

In a query, the `find` statement specifies the event pattern you are interested in. At runtime, for each event that the correlator adds to a window, the query checks for a match. Depending on the definition of the event pattern, the set of events that matches the pattern contains one or more events. This is the match set. A match set

- Always contains the latest event, which is the event that was most recently added to a window.
- Satisfies the event pattern.
- Is always the most recent set that matches the event pattern. This is important when there is more than one set that is a candidate for the match set.

The format of a `find` statement is as follows:

```
find pattern block
```

Syntax Element	Description
<i>pattern</i>	The event pattern that you want to find. See “Defining event patterns” on page 103 .
<i>block</i>	The procedural code to execute when a match is found. See “Acting on pattern matches” on page 127 .

Defining event patterns

In a query definition, you specify a `find` statement when you want to detect a particular event pattern. The `find` statement specifies the event pattern of interest followed by a procedural block that specifies what you want to happen when a match is found. For example:

```
query ImprobableWithdrawalLocations
{
  inputs {
    Withdrawal() key cardNumber within 24 hours;
  }
  find
    Withdrawal as w1 -> Withdrawal as w2 where w2.country != w1.country {
      log "Suspicious withdrawal: " + w2.toString() at INFO;
    }
}
```

In this example, the window that the query operates on contains any `Withdrawal` events that have arrived in the last 24 hours. The key is the card number so each partition contains only `Withdrawal` events that have the same value in their `cardNumber` field. In other words, each partition contains the `Withdrawal` events for one particular account. For more information about input definitions, see [“Defining query input” on page 77](#).

The `find` statement specifies that the event pattern of interest is a `Withdrawal` event followed by another `Withdrawal` event.

In each partition, the `where` clause filters the `Withdrawal` events so that there is a match only when the values of their `country` fields are different. The two event templates in the `find` statement coassign matching `Withdrawal` events to `w1` and `w2`, respectively.

In this example, the two matching `Withdrawal` events might or might not have arrived in the partition consecutively. For details, see [“Query followed-by operator” on page 106](#).

When there is a match the query executes the action in the `find` block.

The format for defining a `find` statement is as follows:

```
find
  [every] [wait duration as identifier]
  event_type as identifier [find_operator event_type as identifier]...
  [wait duration as identifier]
  [where_clause] [within_clause] [without_clause]
  [select_clause] [having_clause] {
    block
  }
```

Syntax Element	Description
<code>event_type</code>	Name of the event type you are interested in. You must have specified this event type in the <code>inputs</code> section.
<code>every</code>	Specify the optional <code>every</code> modifier in conjunction with the <code>select</code> and <code>having</code> clauses. This lets you specify a pattern that aggregates event field values in order to find data based on many sets of events. See “Aggregating event field values” on page 122 .
<code>wait</code>	Specify the optional <code>wait</code> modifier followed by a time literal or a <code>float</code> expression. A <code>wait</code> modifier indicates a period of elapsed time at the beginning of the event pattern and/or at the end of the event pattern. A <code>float</code> expression always indicates a number of seconds, See “Query wait operator” on page 112 .
<code>identifier</code>	<p>Coassign the matching event to this identifier. A coassignment variable specified in an event pattern is within the scope of the <code>find</code> block and it is a private copy in that block. The exception to this is in an aggregating <code>find</code> statement, only the projection expression can use the coassignments from the pattern. The procedural block of code can use projection coassignments and any parameters, but it cannot use coassignments from the pattern. Changes to the content that the variable points to do not affect any values outside the query.</p> <p>Unlike EPL event expressions, you need not declare this identifier before you coassign a value to it.</p>

Syntax Element	Description
	In an event pattern in a <code>find</code> statement, each coassignment variable identifier must be unique. You must ensure that an identifier in an event pattern does not conflict with an identifier in the <code>parameters</code> section, or <code>inputs</code> section.
<i>find_operator</i>	<p>Optionally specify <code>and</code> or <code>-></code> and then specify an <i>event_type</i> and coassignment variable. Parentheses are allowed in the pattern specification and you can specify multiple operators, each followed by an <i>event_type</i> and coassignment variable. For example, the following is a valid <code>find</code> statement:</p> <pre>find (A as a1 -> ((A as a2)) -> (A as a3) -> (A as a4 -> A as a5 -> A as a6) -> (((A as a7) -> A as a8) -> A as a9) -> A as a10) { print "query with 10: "+a1.toString()+ " - "+a10.toString(); }</pre> <p>You can use either <code>as</code> or the colon (<code>:</code>) as the coassignment operator.</p>
<i>where_clause</i>	<p>To filter which events match, specify <code>where</code> followed by a Boolean expression that refers to the events you are interested in. The Boolean expression must evaluate to <code>true</code> for the events to match. The <code>where</code> clause is optional. Coassignment variables specified in the <code>find</code> or <code>select</code> statements are in scope in the <code>where</code> clause. Also available in a <code>where</code> clause are any parameter values and key values. This <code>where</code> clause applies to the event pattern and is referred to as a <code>find where</code> clause to distinguish it from a <code>where</code> clause that is part of a <code>without</code> clause, which is referred to as a <code>without where</code> clause. See “Query conditions” on page 113.</p>
<i>within_clause</i>	<p>A <code>within</code> clause sets the time period during which events in the match set must have been added to their windows. A pattern can specify zero, one, or more <code>within</code> clauses. See “Query conditions” on page 113.</p>
<i>without_clause</i>	<p>A <code>without</code> clause specifies event types whose presence prevents a match. See “Query conditions” on page 113.</p>
<i>select_clause</i>	<p>A <code>select</code> clause indicates that aggregate values are to be computed. See “Aggregating event field values” on page 122.</p>
<i>having_clause</i>	<p>A <code>having</code> clause restricts when the procedural code is invoked for a pattern that aggregates values. See “Aggregating event field values” on page 122.</p>

Syntax Element	Description
<i>block</i>	<p>Specify one or more statements that operate on the matching event(s). For details about code that is permissible in the <code>find</code> block, see “Acting on pattern matches” on page 127.</p> <p>Items available in a <code>find</code> block can include:</p> <ul style="list-style-type: none">■ Any parameters defined in the parameters section■ Coassignment variables specified in the event pattern (or projection coassignments in the case of aggregating <code>find</code> statements).■ Key values

Query followed-by operator

You can specify the `->` (followed-by) operator in the `find` statement. The `->` operator matches events that come after each other. The event on the left of the operator always arrives in the correlator before the event on the right. In other words, the `->` operator is always between two distinct events. For example, `A as a1 -> A as a2` requires the arrival of two instances of an `A` event for the query to find a match. Also, any `where` clauses in the `find` statement must evaluate to `true` for an event pattern to match. Finally, the match set always includes the latest event.

Thus, the rules for when there is a match for an event pattern that specifies one or more followed-by operators are as follows. All of these requirements must be met for there to be a match.

- There are events in the partition that match the subpatterns on both sides of the followed-by operator(s).
- There is a match for the subpattern on the left of a followed-by operator before there is a match for the subpattern on the right of a followed-by operator. One event cannot match more than one subpattern in an event pattern.
- If a subpattern contains a `where` clause then the `where` clause must evaluate to `true` for the subpattern to match.
- The match set contains the latest event.
- If there is more than one candidate event set for the match set then it is the most recent candidate event set that is the match set. See [“Event matching policy” on page 126](#).

The following sections provide examples that illustrate these rules.

Two coassignments

Consider the following code in which the `Withdrawal` event contains only one field of interest, which is the country. Assume that the query partitions arriving `Withdrawal` events into windows according to the account number field.

```
find Withdrawal as w1 -> Withdrawal as w2
```

```

where w1.country = "UK" and w2.country = "Narnia" {
  // Recent card fraud in Narnia against UK customers
  emit SuspiciousWithdrawal(w2);
}

```

To make it easier to understand the behavior of the `->` operator in more populated windows, the following example events omit the account number field but include a unique identifier field. Suppose the window for this query contains the following events, in arrival order top to bottom:

```

Withdrawal("Belgium", 1)
Withdrawal("UK", 2)

```

Although there is a `Withdrawal` event followed by another `Withdrawal` event, the `where` clause does not evaluate to `true` so there is no match. Now suppose the window contains these events:

```

Withdrawal("UK", 3)
Withdrawal("Narnia", 4)

```

Now the query finds a match. There is a `Withdrawal` event followed by another `Withdrawal` event, and the `where` clause evaluates to `true`. `Withdrawal("UK", 3)` is coassigned to `w1` and `Withdrawal("Narnia", 4)` is coassigned to `w2`. The query executes the statements in its `find` block, which in this example is to emit the event that triggered the match.

In this example, the `Withdrawal` events in the match set arrived consecutively. However, this is not a requirement. Consider a window that contains the following events:

```

Withdrawal("UK", 5)
Withdrawal("Belgium", 6)
Withdrawal("Belgium", 7)
Withdrawal("Narnia", 8)

```

When `Withdrawal("Narnia", 8)` is added to its window, the query finds a match because the `Withdrawal("UK", 5)` event is followed by the `Withdrawal("Narnia", 8)` event and the `where` clause evaluates to `true` for those two events. The effective behavior is that all combinations of events in the window are inspected to find a combination that matches. The `Withdrawal("UK", 5)` event is coassigned to `w1` and `Withdrawal("Narnia", 8)` is coassigned to `w2`. The query executes the statements in its `find` block.

A match must include the event that arrived most recently in the window (the latest event). This ensures that a query does not detect more than one match for the same combination of events. In the previous example, the query found a match when the `Withdrawal("Narnia", 8)` event arrived.

Imagine that another `Withdrawal` event arrives and the window now contains the following events:

```

Withdrawal("UK", 5)
Withdrawal("Belgium", 6)
Withdrawal("Belgium", 7)
Withdrawal("Narnia", 8)
Withdrawal("Belgium", 9)

```

While the window still contains the `Withdrawal("UK", 5)` event followed by the `Withdrawal("Narnia", 8)` event, the arrival of the `Withdrawal("Belgium", 9)` event does not trigger a new match because it is not part of that combination. However, suppose the `Withdrawal("Narnia", 10)` event arrives. The window now contains the following events:

```

Withdrawal("UK", 5)

```

```
Withdrawal("Belgium", 6)
Withdrawal("Belgium", 7)
Withdrawal("Narnia", 8)
Withdrawal("Belgium", 9)
Withdrawal("Narnia", 10)
```

Now the query finds a new match. The `Withdrawal("UK", 5)` event is followed by the just arrived `Withdrawal("Narnia", 10)` event and the where clause evaluates to true for these two events. This match set contains `Withdrawal("UK", 5)` and `Withdrawal("Narnia", 10)`. While this match set contains the same `Withdrawal("UK", 5)` event that was in the previous match set, it is a new match set because it contains the event that arrived most recently, which is the `Withdrawal("Narnia", 10)` event.

Suppose that the `Withdrawal("Narnia", 14)` event has just arrived in the following window:

```
Withdrawal("Belgium", 11)
Withdrawal("UK", 12)
Withdrawal("UK", 13)
Withdrawal("Narnia", 14)
```

In this situation, there is a match set that contains the two most recently arrived events, that is, `Withdrawal("UK", 13)` and `Withdrawal("Narnia", 14)`. The `Withdrawal("UK", 12)` event is not part of the match set because it is not the most recently arrived `Withdrawal` event whose country field is "UK".

Three coassignments

The code example below shows three coassignments in the `find` statement. This query partitions the arriving events into windows according to their Automated Transaction Machine identifier numbers (`atmId`).

```
query RepeatedMaxWithdrawals {
  inputs {
    Withdrawal() key atmId within 4 minutes;
  }
  find Withdrawal as w1 -> Withdrawal as w2 -> Withdrawal as w3
    where w1.amount = 500 and w2.amount = 500 and w3.amount = 500 {

    log "Suspicious withdrawal: " + w3.toString() at INFO;
  }
}
```

Each window contains the `Withdrawal` events that occurred in the last four minutes at a particular ATM. For simplicity, the following examples show only the `amount` and `transactionId` event fields. Suppose the following events are in the window and that they arrived in order from top to bottom:

```
Withdrawal(500, 101)    w1
Withdrawal(500, 102)    w2
Withdrawal(500, 103)    w3
```

After the third event arrives, the event pattern is matched, the where clause evaluates to true, and the events are coassigned to `w1`, `w2`, and `w3` as shown above.

Another event arrives in the window:

```
Withdrawal(500, 101)
```

```
Withdrawal(500, 102)    w1
Withdrawal(500, 103)    w2
Withdrawal(500, 104)    w3
```

When the fourth event arrives there is a new match and the events are coassigned as shown above. The `Withdrawal(500, 101)` event is not part of the new match set. A match set always includes the most recent events that satisfy the event pattern and that allow the where clause to evaluate to true.

Another event arrives and the window now contains these events:

```
Withdrawal(500, 101)
Withdrawal(500, 102)
Withdrawal(500, 103)
Withdrawal(500, 104)
Withdrawal(100, 105)
```

The latest event, `Withdrawal(100, 105)`, does not have 500 in its amount field. Consequently, its arrival in the window does not trigger a new match because a match set must always include the latest event. While the window still contains three events that satisfy the event pattern, the actions in the `find` block are not executed as a result of the arrival of `Withdrawal(100, 105)` because it did not trigger a new match.

Another event arrives and the window now contains these events:

```
Withdrawal(500, 101)
Withdrawal(500, 102)
Withdrawal(500, 103)    w1
Withdrawal(500, 104)    w2
Withdrawal(100, 105)
Withdrawal(500, 106)    w3
```

With the arrival of the `Withdrawal(500, 106)` event, there is a new match and the events are coassigned as shown above. The coassigned events are the three most recently arrived events that satisfy the event pattern. It does not matter that `Withdrawal(100, 105)` arrived after some events that are in the match set. That event does not satisfy the event pattern and so it is not included in the match set.

Finally, suppose all of the following events have arrived in the window within the specified four minutes:

```
Withdrawal(500, 101)
Withdrawal(500, 102)
Withdrawal(500, 103)
Withdrawal(500, 104)
Withdrawal(100, 105)
Withdrawal(500, 106)    w1
Withdrawal(500, 107)    w2
Withdrawal(100, 108)
Withdrawal(100, 109)
Withdrawal(500, 110)    w3
```

As you can see, the latest event causes a new match. This match set does not include the two events that arrived just before the latest event. Those two events do not satisfy the event pattern.

Query and operator

In a `find` statement, you can specify the `and` operator in the event pattern. The events on both sides of the `and` operator must be matched for the query to fire. The condition on each side of an `and` operator can be a single event template or a more complex expression.

In the next example, assuming that an `X` event and a `Y` event have already been added to their respective windows, adding an `A` event to its window causes a match.

```
(X as x -> A as a1) and (Y as y -> A as a2)
```

In the second example, suppose events were added to their windows in this order: `X(1)`, `A(1)`, `Y(1)`, `A(2)`. The `A(1)` event is not included in the match set. Only `A(2)` is in the match set because it is the most recent `A` event to follow `X(1)` as well as the most recent `A` event to follow `Y(1)`.

When a single event is used in more than one coassignment you must coassign the event, `A` in these examples, to distinct identifiers, `a1` and `a2` in these examples.

Specification of an `and` operator implies that there are no requirements regarding the order in which the events specified in the event pattern are added to the window. For example, events specified in the right-side condition can be added to their windows before events specified in the left-side condition. When conditions specify multiple events, the events that cause one side of the `and` operator to evaluate to `true`:

- can all be added to their windows before the events that cause the other side to evaluate to `true`;
- can all be added to their windows after the events that cause the other side to evaluate to `true`;
- can arrive in their windows at times interleaved with the arrival of the events that cause the other side to evaluate to `true`;
- can contain the events that cause the other side to evaluate to `true`;
- can be contained by the events that cause the other side to evaluate to `true`.

When there is an order requirement or when you require multiple instances of the same event type, specify the followed-by (`->`) operator.

The `and` operator has a higher precedence than the followed-by (`->`) operator, and lower precedence than the `or` operator. For clarity, use brackets in expressions that specify more than one type of operators.

Query or operator

In a `find` statement, you can specify the `or` operator in the event pattern. The events on one side or the other of the `or` operator must be matched for the query to fire. The condition on each side of an `or` operator can be a single event template or a more complex expression.

In the next example, assuming that a `FlagAccount` event and an `OrderPlaced` event have already been added to the query's window, adding either a `CreditCardAdded` or `OrderCancelled` event to its window causes a match.

```
FlagAccount as account -> (CreditCardAdded as added or
  (OrderPlaced as placed and OrderCancelled as cancelled) )
```

A pattern normally only matches one side of an or operator, as it matches the most recent events. However, if one event matches both sides of an or operator, then both events may be coassigned.

Optional or-terms

Events on one side of the or operator are not required to be present when matching the pattern. In the example above, the added, placed and cancelled coassignments are not all required to be present. It will match if either an added event, or a placed and canceled event appears in the query's window. These terms are referred to as "or-terms". It is possible for the pattern to match with matching only some of those events, and others are left without an event assigned to them. These or-terms are thus optional rather than definitely having a value matched by the pattern. The following rules apply to or-terms:

- Or-terms can only be used in where clauses (see ["Query conditions" on page 113](#)) if the where clause does not make use of or-terms on the opposite side of the or operator in the pattern. In the above example, added is opposite placed and cancelled. Therefore, the following where clauses are not legal:
 - where added.cardId = placed.cardId
 - where added.cardId = 5 or placed.cardId = 5
 (but see the next point for an example of how to express these conditions)
- If one of the where clauses uses or-terms that are not being matched by the pattern, then they are ignored as they cannot be evaluated. For example, only one of the following where clauses is required to match (as it is not possible for both to match):
 - where added.cardId = 5
 - where placed.cardId = 5
- In the action of the query (see ["Acting on pattern matches" on page 127](#)), the type of an or-term is optional<EventType>. The types in the above example are:
 - optional<CreditCardAdded>
 - optional<OrderPlaced>
 - optional<OrderCancelled>

Use the ifpresent statement to handle the contents of such events. See ["Defining conditional logic with the ifpresent statement" on page 274](#) for further information.

- If using the %send construct (see "Adding query send event actions" in *Using Apama with Software AG Designer*), any or-terms required by the fields of the event should be included in an ifpresent entry of the %send. This uses the ifpresent statement, thus the contents of the or-terms events are available to the send action. When using the **Query Send Event Action** dialog in Software AG Designer, the ifpresent is automatically filled out.

- If one side of an `or` term matches and the other side is incomplete, then no `or`-terms from the incomplete side of the `or` operator are included in the matched events. Each side of an `or` operator can either match completely or not at all. In the above example, if a `CreditCardAdded` event occurs, the `OrderPlaced` event is discarded, despite being present in the window. Thus, detecting the presence of just the placed or canceled event with `ifpresent` is sufficient to detect which side of the `or` has matched.
- Aggregates (see [“Aggregating event field values” on page 122](#)) cannot use `or`-terms.

The `or` operator has a higher precedence than the `and` operator, and lower precedence than the followed-by (`->`) operator. For clarity, use brackets in expressions that specify more than one type of operator.

Query wait operator

You can specify the `wait` operator in an event pattern. The `wait` operator indicates that there must be a time interval either at the beginning of the matching pattern or at the end of the matching pattern. The format for specifying the `wait` operator is as follows:

```
wait ( durationExpression ) as coassignmentId
```

You can use either `as` or the colon (`:`) as the coassignment operator.

Syntax Element	Description
<i>durationExpression</i>	A time literal (such as <code>2 min 3 seconds</code>) or a float expression. A float expression can use constants and parameters. It indicates a number of seconds.
<i>coassignmentId</i>	An identifier. You can specify this identifier only in a <code>between</code> clause. See “Query condition ranges” on page 118 .

Typically, you specify the `wait` operator in conjunction with an event pattern condition. For example:

```
find A as a -> B as b -> wait(10) as t
  without X as x between ( b t )
```

There is a match for this pattern when these things happen in this order:

1. An `A` event is added to a window in a partition.
2. A `B` event is added to a window in the same partition.
3. Ten seconds go by without an `X` event being added to a window in that partition.

The `wait` operator can be unambiguously at the beginning of a pattern that uses the followed-by operator or unambiguously at the end of a pattern that uses the followed-by operator. For example:

```
X as x -> wait(1.0) -> Y as y           // Not allowed
X as x and wait(1.0) and Y as y         // Not allowed
X as x and Y as y and wait(1.0)         // Not allowed
wait(1.0) -> (X as x and Y as y)         // Allowed
wait(1.0) -> X as x -> Y as y -> wait(1.0) // Allowed
```


The following code fragment detects the opening of a door without security authorization:

```
find wait( 5 seconds ) as p -> DoorOpened as e
  without SecurityAuthorization as s where s.doorId = e.doorId {
    emit UnauthorizedAccess(e.doorId);
  }
```

Suppose the following events were received:

Time	Event
00	SecurityAuthorization("door1")
02	DoorOpened("door1")
07	DoorOpened("door1")
15	DoorOpened("door2")

The first DoorOpened event for door1 does not generate an alert because a SecurityAuthorization event was received within the 5 seconds that preceded the first DoorOpened event and the doorId field is the same for both events. That is, because the Boolean expression in the where clause of the without clause evaluates to true, a match is prevented and so an alert is not sent.

The second DoorOpened event for door1 causes an UnauthorizedAccess alert because the SecurityAuthorization event was received more than 5 seconds before the second DoorOpened event for door1.

The DoorOpened event for door2 causes an UnauthorizedAccess alert because a SecurityAuthorization event was not received within the 5 seconds that preceded that DoorOpened event. Since there was no SecurityAuthorization event, the Boolean expression in the where clause that is in the without clause evaluates to false, which allows a match.

Query conditions

A find statement can specify conditions that determine whether there is a match for the specified event pattern. The following table provides an overview of the conditions you can specify.

Condition:	where	within	without
Specifies:	Boolean expression	Time period	Event type coassigned to an identifier
Latest event can cause a match when:	The Boolean expression evaluates to true.	Events in the pattern (or, if specified, the between range) must have been received within the time specified. That is, the elapsed time from when the first event was received to when the last	An event of a specified type was not added to a window after the addition of the oldest event in the potential match set nor before the addition of the latest event.

Condition:	where	within	without
		event was received must be less than the within time period.	
Number allowed:	Zero or more	Zero or more	Zero or more
Order when all conditions are specified:	1st	2nd	3rd
Format:	where <i>boolean_expression</i>	within <i>time_literal</i>	without <i>typeId</i> as <i>coassignmentID</i> You can use either as or the colon (:) as the coassignment operator.
Notes:	where x where y is equivalent to where x and y A where clause that precedes any within or without clauses is referred to as a find where clause.	Alternatively, you can specify within <i>expression</i> . The expression must evaluate to a float. Optionally, after each within clause, you can specify a between clause. See “Query condition ranges” on page 118 .	Optionally, after each without clause you can specify one where clause, which is referred to as a without where clause to distinguish it from a find where clause. Optionally, after each without clause, you can specify a between clause. See “Query condition ranges” on page 118 .

Query where clause

A where clause filters which events match. A where clause consists of the where keyword followed by a Boolean expression that refers to the events you are interested in.

Coassignment variables specified in the find statement are in scope in the find where clause. Also available in the find where clause are any parameter values and key values. However, each where clause cannot use or-terms (coassignments that are not required as they are on one side of an or operator in the pattern) from both sides of an or operator in the pattern. Each where clause can only use coassignments from at most one side of every or operator in the pattern. Different where clauses can use coassignments from different sides of an or operator (see [“Query or operator” on page 110](#)). Thus, when writing where clauses which apply to either side of an or, use separate where clauses for each condition. They cannot be combined into a single where clause with an or or and operator in the where clause; they may require both sides to be evaluated.

For example, instead of

```
find OrderPlaced:placed or OrderCancelled:cancelled
```

```
where placed.orderId = 1 or cancelled.orderId = 2
```

write the following:

```
find OrderPlaced:placed or OrderCancelled:cancelled
  where placed.orderId = 1
  where cancelled.orderId = 2
```

For where clauses that do not use any coassignments, all of the Boolean expressions must evaluate to true for the events to match.

For where clauses that use or-terms, they only apply if the events they make use of are matched by the pattern. If they use or-terms that have not been matched by the pattern, then those where clauses are ignored as they cannot be evaluated.

All of the where clauses that can be evaluated must be true for the pattern to match. If a single where clause combines (with an and or or operator) conditions on an or-term and a normal coassignment, then the entire where clause is ignored if the or-term is not matched.

The where clause is optional. You can specify zero, one or more where clauses.

Note:

You can specify a find where clause that applies to the event pattern and you can also specify a without where clause that is part of a without clause. Any where clauses that you want to apply to the event pattern must precede any within or without clauses.

Query within clause

A within clause sets the time period during which events in the match set must have been added to their windows. A pattern can specify zero, one, or more within clauses. These must appear after any find where clauses and before any without clauses. The format of a within clause is as follows. The between clause is optional.

```
within durationExpression [ between ( identifier1 identifier2 ... ) ]
```

The *durationExpression* must be a time literal (such as 2 min 3 seconds) or it must evaluate to a float value. A float expression can use constants and parameters. It indicates a number of seconds.

For example, consider the following find statement:

```
find LoggedIn as lc -> OneTimePass as otp
  where lc.user = otp.user
  within 30.0 {
    emit AccessGranted(lc.user);
  }
```

If specified, the between clause lists two or more items. Each item can be a coassigned event in the pattern. A wait coassignment can also be specified. These items define a range. See [“Query condition ranges” on page 118](#). For example:

```
find wait(1.) as w -> A as a {
  ...
  within (5.0) between w a
```

Now assume that the following events arrive:

Time	Event	Access Granted?
10	LoggedIn("Andy")	
15	OneTimePass("Andy")	Yes. Both events received within 30 seconds.
20	LoggedIn("Mike")	
60	OneTimePass("Mike")	No. OneTimePass event received more than 30 seconds after corresponding LoggedIn event.
60	LoggedIn("Sam")	
90	OneTimePass("Sam")	No. OneTimePass event received exactly 30 seconds after corresponding LoggedIn event. For there to be a match, the OneTimePass event must be received less than 30 seconds after its corresponding LoggedIn event.

As mentioned earlier, a `find` statement can specify multiple `within` clauses. This is useful when the pattern of interest refers to multiple events and you specify a `between` range as part of each `within` clause. When you specify multiple `within` clauses they must all be satisfied for there to be a match.

Query without clause

A `without` clause specifies event types, which must be specified in the `inputs` block of the query, whose presence prevents a match. For example, if a potential match set contains 3 events, it can be a match only if a type specified in a `without` clause was not added to a window neither after the first event nor before the third event. Any event type that can be used in the `find` pattern can be used in the `without` clause.

Optionally, after each `without` clause, you can specify one `where` clause, which is referred to as a `without where` clause to distinguish it from a `find where` clause. The following table compares `find where` clauses and `without where` clauses.

Find where clause	Without where clause
<code>true</code> allows a match. Think of this as a positive <code>where</code> clause.	<code>false</code> allows a match. Think of this as a negative <code>where</code> clause.
Can only be before any <code>within</code> or <code>without</code> clauses	Can only be part of a <code>without</code> clause
Applies to the event pattern	Applies to the event specified in its <code>without</code> clause
Cannot refer to event specified in <code>without</code> clause	Can refer to event specified in <code>without</code> clause

The absence of an event of a type specified in a `without` clause has the same effect as the presence of an event for which the `without where` clause evaluates to `false`.

In addition to being able to refer to parameters and coassignment identifiers in the event pattern, a `without where` clause can refer to the one event mentioned in its `without` clause. When a `without where` clause evaluates to `true`, the presence of the `without` event prevents a match. If a `without where` clause is `false`, then that `without` event instance is ignored; that is, a match is possible.

A `without` clause cannot use the `->` or `and` pattern operators. However, you can specify multiple `without` clauses. If there are multiple `without` clauses each one can refer to only its own coassignment and not coassignments in other `without` clauses. However, all `without` clauses can make use of the pattern's standard coassignments, such as `od.user` in the example at the end of this topic.

If there are multiple `without` clauses a matching event for any one of them prevents a pattern match. Multiple `without` clauses can use the same type and the same coassignment, which is useful only when their `where` conditions are different.

Typically, a `without where` clause references the event in its `without` clause, but this is not a requirement.

Optionally, after each `without` clause, you can specify a `between` clause, which lists two or more coassigned events. It can also list a `wait` coassignment. For an event to cause a match, the type specified in the `without` clause cannot be added to the window between the points specified in the `between` clause. See [“Query condition ranges” on page 118](#).

Any `without` clauses must be after any `find where` clauses and `within` clauses. If you specify both optional clauses, the `without where` clause must be before the `between` clause.

When a `without` clause includes both optional clauses, `where` and `between`, the format looks like this:

```
without typeId as coassignmentId
  where boolean_expression
  between ( identifier1 identifier2... )
```

As mentioned previously, a `find where` clause applies to the event pattern while a `without where` clause applies to the event specified in its `without` clause. The following table shows the resulting behavior according to the type of the `where` clause and the value of its Boolean expression:

Type of where clause	Boolean expression evaluates to true	Boolean expression evaluates to false
Find where clause applies to event pattern	Allows match	Prevents match
Without where clause applies to its without event	Prevents match	Allows match

Example

Consider the following `find` statement:

```
find OuterDoorOpened as od -> InnerDoorOpened as id
  where od.user = id.user
  without SecurityCodeEntered as sce where od.user = sce.user {
    emit Alert("Intruder "+id.user);
```

```
}

```

Now suppose the following events arrive:

Event Received	Result
OuterDoorOpened("Andrew")	
SecurityCodeEntered("Andrew")	Causes the without where clause to evaluate to true, which prevents a match.
InnerDoorOpened("Andrew")	No alert is set.
OuterDoorOpened("Brian")	
InnerDoorOpened("Brian")	Because there is no intermediate SecurityCodeEntered event, there is a match and the query sends an alert. This is an example of how the absence of an event of a type specified in a without clause has the same effect as the presence of an event for which the without where clause evaluates to false.
OuterDoorOpened("Chris")	
SecurityCodeEntered("Charlie")	Causes the without where clause to evaluate to false, which allows a match.
InnerDoorOpened("Chris")	Causes a match and the query sends an alert.
OuterDoorOpened("Dan")	
SecurityCodeEntered("David")	Causes the without where clause to evaluate to false, which allows a match.
SecurityCodeEntered("Dan")	Causes the without where clause to evaluate to true, which prevents a match.
SecurityCodeEntered("Dense1")	Causes the without where clause to evaluate to false, which allows a match.
InnerDoorOpened("Dan")	There is no match because one of the SecurityCodeEntered events caused the without where clause to evaluate to true, which prevents a match.

Query condition ranges

The within and without clauses (see [“Query conditions” on page 113](#)) can each have an optional between clause that restricts which part of the pattern the within or without clause applies to. The format for specifying a range is as follows:

```
between ( identifier1 identifier2 ... )
```

At least two identifiers that are specified in the event pattern are required. The identifiers specify a period of time that starts when one of the specified events is received and ends when one of the other specified events is received. A *between* clause is the only place in which you can specify a coassignment identifier that was assigned in a *wait* clause. You cannot specify identifiers used in a *without* clause. Also, the same event cannot match both the coassignment identifier in the *without* clause and an identifier in a *between* clause.

The condition that the *between* clause is part of must occur in the range of identifiers specified in the *between* clause. For example, consider the following *find* pattern:

```
find A as a and B as b and C as c without X as x between ( a b )
```

For there to be a match set for this pattern, no *X* event can be added to its window between the arrivals of the *a* and *b* events. If events are received in the order *B A X C*, then there is a match set because the *x* event is not between the *a* and *b* events. If the events are received in the order *B C X A*, then there is no match set because an *x* event occurred between the *a* and *b* events.

Here is another example:

```
find A as a -> B as b -> (C as c and D as d)
  within 10.0 between (a b)
  within 10.0 between (c d)
```

Range	Description
(a b)	This duration starts when an <i>A</i> event is received because the pattern is looking for an <i>A</i> event followed by a <i>B</i> event. For there to be a match, the <i>B</i> event must arrive less than 10 seconds after the <i>A</i> event.
(c d)	After an <i>A</i> event followed by a <i>B</i> event has been received, this duration starts when either a <i>C</i> event or a <i>D</i> event is received. Since the pattern is looking for a <i>c</i> and a <i>d</i> , it does not matter which event is received first. For there to be a match, the event that is not received first must be received less than 10 seconds after the first event.

The following table provides examples of match sets.

Time	Event Received	Match Set
10	A(1)	
15	B(1)	
20	D(1)	
25	C(1)	A(1), B(1), D(1), C(1)
37	D(2)	No match. More than 10 seconds elapsed between C(1) and D(2).
40	C(2)	A(1), B(1), D(2), C(2)

The range is exclusive. That is, the range applies only after the first event is received and before the last event is received. For example, consider this pattern:

```
find A as a1 -> A as a2 without A as repeated between ( a1 a2 )
```

A match set for this pattern is two consecutive A events. If three consecutive A events are added to the window, the first and third do not constitute a match set event though the first A was followed by the third A. This is because the second A was added between the first and the third A events. In other words, the events that match a1 and a2 are excluded from the range in which the repeated event can match. The following table provides examples of match sets for this pattern. It assumes that A(1) is still in the window when A(4) is added.

Event Added to Window	Match Set	Not a Match Set
A(1)		
A(2)	A(1), A(2)	
A(3)	A(2), A(3)	A(1), A(3)
A(4)	A(3), A(4)	A(1), A(4) and A(2), A(4)

The query below is a real world example of the pattern just discussed. It emits the average price change in the last minute.

```
query FindAveragePriceMove {
  inputs {
    Trade() key symbol within 1 minute;
  }
  find every Trade as t1 -> Trade as t2
    without Trade as mid between (t1 t2)
    select avg(t2.price - t1.price) as avgPriceChange {
      emit AveragePriceChange(symbol, avgPriceChange);
    }
}
```

It is illegal to have two `within` clauses with identical between ranges. This would be redundant, as only the shortest within duration would have any effect. It is, however, legal to have more than one `without` clause with the same between range. Typically, these would refer to different event types or where conditions.

If or-terms (see [“Query or operator” on page 110](#)) are included in the range of a condition, then if an or-term is not matched, that coassignment is ignored in the range. If this means that the range has less than two points, the condition is ignored. There must be a combination of events for which there are at least two coassignments definitely in the range. Using only or-terms on opposite sides of an or operator in the pattern is an error, as the condition will never apply.

Special behavior of the `and` operator

To optimize performance when evaluating a query where clause, the correlator evaluates each side of an `and` operator as early as possible even if evaluation is not in left to right order. This behavior is different from the behavior outside a query. That is, outside a query, the left side of an `and` operator is guaranteed to be evaluated first. See [“Logical intersection \(and\)” on page 663](#).

For example, suppose you specify the following event pattern:

```
A as a -> B as b where a.x = 1 and b.y = 2
```

Consider what happens when the following events are added to their windows:

```
A(1), A(2), A(3), B(5), B(4), B(3)
```

The correlator can identify that

- only the `a` coassignment target is needed to evaluate the `a.x = 1` condition;
- only the `b` coassignment target is needed to evaluate the `b.y = 2` condition.

Because none of the `B` events cause the `b.y = 2` condition to evaluate to `true`, the correlator does not evaluate the `a.x = 1` condition.

In a `where` clause, because the right side of an `and` operator might be evaluated first, you should not specify conditions that have side effects. Side effects include, but are not limited to:

- `print` or `log` statements
- `route`, `emit`, `enqueue...to` statements
- Modifying events, sequences, dictionaries, etc.
- Causing a runtime error
- Calling an action that has a side effect statement in it
- Calling plug-ins that have side effects

If a `where` clause calls an action that has a side effect, you should not rely on when or whether the action is executed.

Whether the correlator can optimize evaluation of the `where` clause depends on how you specify the `where` clause conditions. For example, consider the following event definition:

```
event Util {
    static action myWhereClause(A a, B b) returns boolean {
        return a.x = 1 and b.y = 2;
    }
}
```

Suppose you specify the following event pattern:

```
A as a -> B as b where Util.myWhereClause(a, b)
```

If the same `A` and `B` events listed above are added to their windows, the result is the same as the result of evaluating the following:

```
A as a -> B as b where a.x = 1 and b.y = 2
```

However, evaluation might take longer because the correlator cannot separate evaluation of `b.y = 2` from evaluation of `a.x = 1`. The `myWhereClause()` action returns `a.x = 1 and b.y = 2` as a single expression. Consequently, the correlator evaluates `Util.myWhereClause(a, b)` for each combination of `a` and `b`. Given the `A` and `B` events listed above, this is a total of 9 times.

While the correlator might evaluate some where clause conditions in a right-to-left order, the correlator always evaluates each where clause condition as soon as it is ready to be evaluated. When multiple conditions become ready to be evaluated at the same time then the correlator evaluates those conditions in the order they are written. For example, the typical pattern of checking whether a dictionary contains a key before operating on the value with that key continues to work reliably:

```
E as e -> F as f where e.dict.containsKey("k") and e.dict["k"] = f.x and f.y = 1
```

In this example, `f.y = 1` might be evaluated before the other two conditions, but `e.dict.containsKey("k")` is always evaluated before `e.dict["k"] = f.x`, and the latter is not evaluated if the `containsKey()` method returns `false`.

Aggregating event field values

A `find` statement can specify a pattern that aggregates event field values in order to find data based on many sets of events. A pattern that aggregates values specifies the `every` modifier in conjunction with `select` and `having` clauses.

Based on a series of values, an aggregate function computes a single value, such as the average of a series of numbers. See the *API Reference for EPL (ApamaDoc)* for detailed information on all built-in aggregate functions.

Note:

If a built-in aggregate function does not meet your needs, you can use EPL to write a custom aggregate function. A custom aggregate function that you want to use in a query must either be a bounded function or it must support both bounded and unbounded operation. See [“Defining custom aggregate functions” on page 221](#).

For example, the following query watches for a withdrawal amount that is greater than some threshold multiplied by the average withdrawal amount of the `ATMWithdrawal` events in the window, which might be as many as 20 events. This query uses the `last()` aggregate function to identify the event added to the window most recently and uses the `avg()` aggregate function to find the average withdrawal amount of the events in the window. The `having` clause must evaluate to `true` for the query to send the `SuspiciousTransaction` event, passing the transaction ID of the suspicious withdrawal. You can use either `as` or the colon (`:`) as the coassignment operator.

```
using com.apama.aggregates.avg;
using com.apama.aggregates.last;
query FindSuspiciouslyLargeATMWithdrawals {
    parameters {
        float THRESHOLD;
    }
    inputs {
        ATMWithdrawal() key accountId retain 20;
    }
    find every ATMWithdrawal as w
        select last(w.transactionId) as tid
        having last(w.amount) > THRESHOLD * avg(w.amount){
            send SuspiciousTransaction(tid) to SuspiciousTxHandler;
        }
}
```

To use an aggregate function in a `find` statement, specify the `every` modifier and specify one or more `select` or `having` clauses. A `select` clause indicates that aggregate values are to be computed. Each `select` clause specifies a projection expression and a projection coassignment. The projection expression can use coassignments from the pattern if the coassignments are within a single aggregate function call. For example, the following pattern computes the average value of the `x` member of event type `A` in the query's input and coassigns that average value to `aax`.

```
find every A as a select avg(a.x) as aax
```

A `select` clause can use parameter values. For example the following two `select` clauses are both valid if there is a parameter `param`:

```
find every A as a
  select avg(param * a.x) as apax
  select param * avg(a.x) as paax
```

You can specify multiple `select` clauses to produce multiple aggregate values.

In an aggregating `find` statement, only the projection expression can use the coassignments from the pattern. The procedural block of code can use projection coassignments and any parameters, but it cannot use coassignments from the pattern.

The `first()` and `last()` built-in aggregate functions are useful if you want to refer to the coassignment value of the oldest or newest event, respectively, in the window.

The following example determines the average price of trades other than your own:

```
find every Trade as t
  where t.buyer != myId and t.seller != myId
  select wavg(t.price, t.amount) as avgprice
```

Match sets used in aggregations

In `find` statements without the `every` modifier, only the most recent set of events that match the pattern are used to invoke the procedural code block. With the `every` modifier, every set of events that matches the pattern is available for use by the aggregate function, provided that the latest event is present in one of the sets of events. Any events or combinations of events that do not match the pattern or do not match the `where` clause, or are invalidated due to a `within` or `without` clause, are ignored; their values are not used in the aggregate calculation.

For example, consider the following `find` statement:

```
find every A as a -> B as b
  where b.x >= 2
  select avg(a.x + b.x) as aabx {
    print aabx.toString();
  }
```

The following table shows what happens as events are added to the window.

Event Added to Window	Match Sets	Average Of	Value of aabx
A(1)	None		
A(2)	None		
B(2)	A(1), B(2) A(2), B(2)	3 and 4	3.5
B(1)	None because B(1) causes the where clause to be false.		
B(3)	A(1), B(2) A(2), B(2) A(1), B(3) A(2), B(3)	3, 4, 4, and 5	4

Note:

Only coassignments that definitely have a value may be used in aggregates. Or-terms that are on one side of an or operator in the pattern may not be used in aggregate expressions (see also [“Query or operator” on page 110](#)).

Using aggregates in namespaces

As with event types, an aggregate function is typically defined in a namespace. To use an aggregate function, specify its fully-qualified name or a using statement. The built-in aggregate functions are in the `com.apama.aggregates` namespace. For example, to use the `avg()` aggregate function you would specify the following in the query:

```
using com.apama.aggregates.avg;
```

Filtering unwanted invocation of procedural code

Each `select` clause defines an aggregate value to be produced. You can also specify one or more `having` clauses to restrict when the procedural code is invoked. For example, consider the following `find` statement:

```
find every A as a
  select avg(a.x) as aax
  having avg(a.x) > 10.0 {
    print aax.toString();
  }
```

This example calculates the average value of `a.x` for the set of `A` events in the window. However, it executes the procedural block only when the average value of `a.x` is greater than `10.0`.

Multiple having clauses

You can specify multiple having clauses and you can use parameter values in having clauses. For example,

```
find every A as a
  select avg(a.x) as aax
  select sum(a.y) as aay
  having avg(a.x) > 10.0
  having sum(a.y) > param1
  having max(a.z) < param2
  {
    print aax.toString(), + " : " + aay.toString();
  }
```

When you specify more than one having clause it is equivalent to specifying the and operator, for example:

```
...
having avg(a.x) > 10.0 or sum(a.y) > param1
having max(a.z) < param2
...
```

is equivalent to

```
...
having ( avg(a.x) > 10.0 or sum(a.y) > param1 ) and ( max(a.z) < param2 )
...
```

Using a select coassignment in a having clause

Rather than specifying an aggregate expression twice, once in a select clause and then subsequently in a having clause, it is possible to refer to the aggregate value by using the select coassignment name. For example:

```
find every A as a
  select avg(a.x) as aax
  having avg(a.x) > 10.0 {
    print aax.toString();
  }
```

You can rewrite that as follows:

```
find every A as a
  select avg(a.x) as aax
  having aax > 10.0 {
    print aax.toString();
  }
```

Using a having clause without a select clause

When you want to test for an aggregate condition but you do not want to use the aggregate value, you can specify a having clause without specifying a select clause. For example,

```
find every A as a
```

```
having avg(a.x) > 10.0 {  
  print "Average value is greater than ten!";  
}
```

Event matching policy

It is possible for the windows for a given key to contain multiple sets of events that, each taken in isolation, would match the defined event pattern. In this case, the matching policy determines which of the candidate event sets is the match set that triggers the query. There are two event matching policies:

- **Recent** — This is the only policy followed for queries that do not specify the `every` keyword, that is, they do not specify aggregate functions.
- **Every** — This is the only policy followed for queries that specify the `every` keyword. That is, they specify aggregate functions.

For both policies, the match set must include the latest event. The latest event is the event that was most recently added to the set of windows identified by a particular key.

For the recent matching policy, to identify which candidate match set triggers the query, the correlator compares the times of the second-most-recent events in the candidate event sets. If one of these events is more recent than its corresponding event(s) then the candidate event set it is in is the match set. However, if two or more candidate event sets share the second-most-recent event, then the correlator compares the times of the third-most-recent events in those candidate event sets. The correlator continues this until it finds an event that is more recent than its corresponding event(s) in other candidate event set(s). The candidate event set that becomes the match set is referred to as the most recent set that matches the event pattern.

Once the correlator determines which candidate event set is the match set, it ignores the order of any earlier events in any event sets. This means that it is possible for the most recent set of events to contain an event that was added earlier than an event in a set that is not the most recent set. The following event definitions and sample query illustrate this.

```
event APNR {  
  // Automatic Plate Number Recognition  
  string road;  
  string plateNumber;  
  integer time; // Represents time order for illustration purposes  
}  
  
event Accident {  
  string road;  
}  
  
event NotifyPolice {  
  string road;  
  string plateNumber;  
}
```

The following query uses these events:

```
query DetectSpeedingAccidents {  
  inputs {
```

```

    APNR() key road within(150.0);
    Accident() key road within(10.0);
  }
  find APNR as checkpointA -> APNR as checkpointB -> Accident as accident
    where checkpointA.plateNumber = checkpointB.plateNumber
    and checkpointB.time - checkpointA.time < 100
    // Which indicates the car was speeding
  {
    emit NotifyPolice(accident.road, checkpointA.plateNumber);
  }
}

```

Suppose the following events are in the query windows:

APNR("MyRoad", "2N2R4", 1000)

APNR("MyRoad", "FAB 1", 1010)

APNR("MyRoad", "FAB 1", 1080)

APNR("MyRoad", "2N2R4", 1090)

Accident("MyRoad")

There are two candidate event sets:

Coassignment identifier	A candidate event set	Another candidate event set
checkpointA	APNR("MyRoad", "2N2R4", 1000)	APNR("MyRoad", "FAB 1", 1010)
checkpointB	APNR("MyRoad", "2N2R4", 1090)	APNR("MyRoad", "FAB 1", 1080)
accident	Accident("MyRoad")	Accident("MyRoad")

Both sets match against the single Accident event. The next most recent events are APNR("MyRoad", "2N2R4", 1090) and APNR("MyRoad", "FAB 1", 1080). The APNR("MyRoad", "2N2R4", 1090) event is more recent. Consequently, after the Accident event is added to its window, there is a match and the match set includes the Accident event and the 2N2R4 APNR events. This is the most recent set of events.

In this example, in the most recent set of events, the earliest event, APNR("MyRoad", "2N2R4", 1000) is earlier than the earliest event, APNR("MyRoad", "FAB 1", 1010), in the other set of events.

Acting on pattern matches

When a query finds a set of events that matches the specified pattern it executes the statements in its `find` block. The `find` block specifies one or more statements that operate on the matching event(s). The items available in a `find` block include:

- Any parameters defined in the `parameters` section.
- Coassignment variables specified in the event pattern.

In the case of an aggregating `find` statement, only the projection expression can use the coassignments from the pattern. The `find` block can use projection coassignments, but it cannot use coassignments from the pattern.

- Key values.
- Actions that are defined in the same query after the `find` block. Any expression in the `find` statement pattern or block can reference an action defined after the `find` block.
- EPL constructs and statements that are allowed in queries. See [“Restrictions in queries” on page 134](#).

Defining actions in queries

In a query, after a `find` statement, you can define one or more actions in the same form as in EPL monitors. See [“Defining actions” on page 250](#).

In a given query, an action that you define can be referenced from any expression in that query's `find` statement, including any statements in its `find` block. For example:

```
query CallingQueryActions {
  parameters {
    float distanceThreshold;
    float period;
  }
  inputs {
    Withdrawal() key account within period;
  }
  find Withdrawal as w1 -> Withdrawal as w2
  where distance(w1.coords, w2.coords) > distanceThreshold
  {
    logIncident( w1, w2 );
    sendSmsAlertToCustomer(
      getTelephoneNumber(w1), getAlertText(w1,w2) );
  }

  action distance( Coords a, Coords b) returns float {
    integer x := a.x - b.x;
    integer y := a.y - b.y;
    return ( x*x + y*y ).sqrt();
  }

  action logIncident ( Withdrawal w, Withdrawal w2 ) { ... }
  action getTelephoneNumber(Withdrawal w ) returns string { ... }
  action getAlertText ( Withdrawal w1, Withdrawal w2 ) returns string { ... }
  action sendSmsAlertToCustomer( string telephoneNumber, string text ) { ... }
}
```

Note:

In a query, do not define an action whose name is `onload`, `ondie`, `onunload`, `onBeginRecovery`, or `onConcludeRecovery`. In EPL monitors, actions with these names have special meaning. For more information, see [“Monitor actions” on page 623](#).

Implementing parameterized queries

An Apama query can define parameters and then refer to those parameters throughout the query definition. This enables a query definition to function as a template for multiple query instances.

A query that defines parameters is referred to as a parameterized query. An instance of a parameterized query is referred to as a parameterization.

A parameterized query offers the following benefits:

- Patterns of interest (`find` patterns) may be customized from a single generic query. This can significantly reduce the amount of code that needs to be written and maintained.
- Parameterizations exist only at runtime. There is no need to maintain a file for each instance.
- Parameters can be used throughout the query in which they are defined. For example, you can use them in the definition of inputs, in `find` actions, and in user-defined actions. Values do not need to be hardcoded.

You define query parameters in the `parameters` section of a query definition. See also [“Format of query definitions” on page 68](#). The format for specifying the parameters section is as follows:

```
parameters {
    data_type parameter_name;
    [ data_type parameter_name; ]...
}
```

In the following example, the `parameters` section is in bold as are the references to the parameters.

```
query FaultyProduct {
    parameters {
        string product;
        float thresholdCost;
        float warrantyPeriod;
    }
    inputs {
        Sale() key customerId within warrantyPeriod;
        Repair() key customerId retain 1;
    }
    find Sale() as s1 -> Repair() as r1
    where s1.product = product
    and r1.product = product
    and r1.cost >= thresholdCost
    {
        log "Cost of warranty covered repair for product \" + product +
            "! above threshold $" + thresholdCost.toString() + " by $"
            + (r1.cost - thresholdCost).toString() at INFO;
    }
}
```

See also: [“Query lifetime” on page 628](#).

Parameterized queries as templates

When a parameterized query is injected into a correlator no instances of the query are created until a request to create a parameterization is sent using the Scenario Service (that is, the `com.apama.services.scenario` client API). This request must include valid values for the query's parameters. For example, if the query in the previous topic is injected, the request to create a parameterization must include valid values for the `product`, `thresholdCost`, and `warrantyPeriod` parameters. Only then does the query become active.

A parameterized query lets you define a generic query `find` pattern that operates on a particular group of input types and that can be customized for particular criteria. The query in the previous topic could be created for any product with the threshold cost and warranty period specified as required. To achieve the same result with a non-parameterized query, you would have to define a query such as the following:

```
query FaultyProduct {
  inputs {
    Sale() key customerId within 1 week; //warrantyPeriod
    Repair() key customerId retain 1;
  }
  find Sale() as s1 w-> Repair() as r1
  where s1.productId = "Mobile device A" // productId
  and r1.productId = "Mobile device A" // productId
  and r1.cost >= 50.00 // thresholdCost
  {
    log "Cost of warranty covered repair for product \"Mobile device A\"
    \" above threshold $50.00 by $" + (r1.cost - 50.00).toString() at INFO;
  }
}
```

While this query is valid it has the drawback that whenever you want to perform a similar query for a product that differs by type, warranty coverage period or threshold repair cost then a new query will need to be written (or most likely copied and pasted) with the new set of values and then injected into the correlator. The benefit of a parameterized query is that only one query definition needs to be injected into the correlator and you can then manually or programmatically create as many different instances for the different product-value combinations as required.

Using the Scenario Service to manage parameterized queries

There are several ways to manage (create/edit/remove) parameterizations:

- Use the `ScenarioService` API in Java or .NET client libraries. See "Developing Custom Clients" in *Connecting Apama Applications to External Components*.
- Use Apama's Scenario Browser view in Software AG Designer. See "Scenario Browser view" in *Using Apama with Software AG Designer*.
- Write dashboards that control the instances of a parameterized query. See "Building Dashboard Clients" in *Building and Using Apama Dashboards*.

The Scenario Service is also used to read and manage instances of `DataViews` and `MemoryStore`.

To these tools, a query will appear with the fully qualified name declared in the `.qry` file prefixed with `QRY_` to highlight that the entity being viewed is a query. For parameterized queries, instances can be created, edited or deleted. For unparameterized queries, a single instance will appear as soon as the query is injected. This instance cannot be edited nor deleted, nor new instances created.

When there is a request to create a parameterization, the Scenario Service tries to validate the supplied parameter values. If the values are valid, the result is as if a query with those values had just been injected.

End users have the ability to define conditions on parameter values when setting them in dashboards. Parameter values can be modified only by the Scenario Service. Updates by the Scenario Service do not occur atomically across all contexts if the query is running in multiple contexts. Consequently, it is possible to observe the effects of the old parameter values interleaved with the effects of the new parameter values. For example, consider a query that has a pattern such as the following:

```
find A as a -> wait(paramValue) as t
```

The wait period will be based on the value the parameter had when the wait period started. If the parameter value is edited after the `A` event enters the partition the `wait` still fires according to the old value. Such transitions are typically short. The actual time required depends on various factors such as machine load and memory.

Some important differences between parameterized queries and other strategies include:

- Parameterized queries have input variables but not output variables. DataViews and MemoryStore have both input variables and output variables. All queries have an empty list of output variables.
- Requests to create or update a parameterization with values that are invalid will be denied. Invalid values are values that would cause `wait`, `within` or `retain` clauses to evaluate to less than or equal to zero, or would cause them to fail to evaluate, for example, by causing a runtime exception to be thrown.

For example, consider the following query:

```
query ParameterizationExample {
  parameters {
    integer intParam;
    integer floatParam;
  }

  inputs {
    A() key id retain (10/intParam);
    B() key id within (5.0 - floatParam);
  }

  find A as a -> B as b -> wait(-1.0 * floatParam)
    where (a.intField/intParam > 0) {
    log "Found match" at INFO;
  }
}
```

Suppose that there is a request to create a parameterization of this query. The request indicates that `intParam` is equal to 0 and `floatParam` is equal to 10.0. If the parameterization were created then every expression that contains a parameter value would immediately throw an exception or

be invalid. In the `inputs` block, evaluation of the `retain` expression would result in a divide-by-zero exception. The `within` expression would evaluate to `-5.0`, which is not valid. Similarly, upon evaluating the elements in the `find` block the `wait` expression would be a negative value and the `where` clause would also result in a divide-by-zero exception. Since a parameterization such as this would lead to either invalid expressions or exceptions being thrown, these values are not allowed. If you try to pass disallowed values to the Scenario Service `createInstance()` method then the Scenario Service returns `null`. Similarly, if you try to pass invalid values to the Scenario Service `editInstance()` method, then the Scenario Service returns `false`, which indicates an error.

Referring to parameters in queries

You can refer to parameters throughout a query definition.

You cannot change parameter values in the query code itself. Parameter values can be modified only by the Scenario Service.

CAUTION:

Apama recommends that you do not change parameter values used in input filters because it is possible to miss events that would cause a match. In a given parameterization, when an input filter refers to a parameter and you change the value of that parameter, it causes the parameterization to stop and restart. Events sent during the changeover are ignored. Also, there might have been earlier events that match the new parameter value but that did not make it into the window because they did not match the previous parameter value. An alternative is to use a parameter in a `where` clause in the `find` statement instead. This can be more efficient when the parameter value needs to be changed frequently. Using parameter values in input filters can also increase memory usage, see [“Queries can share windows” on page 79](#).

Examples of using parameters in queries:

- In `retain` and `within` expressions that are in the `inputs` block:

```
parameters {
    integer maxRetention;
    float maxDuration;
}
inputs {
    A() key id retain maxRetention;
    B() key id with maxDuration;
}
```

- In the filter of the event template in the `inputs` block:

```
parameters {
    float threshold;
}
inputs {
    Withdrawal(amount > threshold) key k;
}
```

- In `where` and `within` clauses that are in the `find` pattern:

```
parameters {
    float maxDuration;
    float maxDifference;
```

```

}
inputs {
    A() key id retain 2;
}
find A as a1 -> A as a2 where (a2.cost - a1.cost) >
    maxDifference within maxDuration {
    ...
}

```

- In wait expression(s) that are in the find pattern:

```

parameters {
    float interval;
}
inputs {
    A() key id retain 2;
}
find A as a1 -> wait(interval) as w1 -> A as a2 {
    ...
}

```

- In an aggregate expression that is in the find pattern:

```

parameters {
    float avg;
}
inputs {
    A() key id within 1 day;
}
find every A as a
    select avg(a.cost - avg) as avgDeviation {
    ...
}

```

- In an action that is in the find block:

```

parameters {
    float avg;
}
inputs {
    A() key id retain 1;
}
find A as a {
    log "Deviation from mean = " + (a.value - avg).toString();
}

```

- In a user-defined action block:

```

parameters {
    float avg;
}
inputs {
    A() key id retain 1;
}
find A as a {
    log "Deviation from mean = " + getDeviation(a).toString();
}
action getDeviation(A a) returns float {
    return (a.value - avg);
}

```

While parameter values can be used anywhere within the query it is illegal to mutate the parameter values. They can be modified only by the Scenario Service.

Scaling and performance of parameterized queries

Depending on the machine architecture a user can expect to be able to create several hundred parameterizations, which all concurrently process events.

As a result of the time required to process a parameterization edit request, the recommendation is to avoid multiple simultaneous edit requests for the same parameterization. There is no guarantee that all of the threads executing the parameterization will hold the same parameter values during the update period. During the update period, there might be a mix of results based on old parameter values and results based on new parameter values. Any requests to the same parameterization should be spaced approximately 1 second apart to allow time for requests to be executed throughout the parameterization. This applies to create, edit and delete requests.

In a cluster of correlators, the correlators share the same set of parameter values across the cluster. While a Scenario Service client can connect to any correlator in the cluster, it is not recommended to edit the same parameterization from multiple Scenario Service clients concurrently, as the results will be undefined.

Restrictions in queries

There are some EPL elements that are appropriate for monitors but not queries, for example `spawn` and `die`. This is because queries scale automatically, with multiple threads of execution processing the events for different partitions as and when they arrive. Hence, within query code, the `spawn` and `die` operations are meaningless. Queries operate on the events in their windows and do not need to set up event listeners, stream queries, or stream listeners. Also, queries cannot subscribe to receive events sent to particular channels.

The following EPL features cannot be used in queries:

- Event listeners, that is, on statements
- Stream queries and stream listeners
- `spawn` and `spawn...to` statements
- `die` statements
- `monitor.subscribe()` and `monitor.unsubscribe()`
- An identifier cannot start with two consecutive underscore characters. For example, `__MyEvent` is an invalid event type name in a query (it is valid in a monitor). A single underscore at the beginning of an identifier is valid.
- Predefined `self` variable

Of course, you cannot call an action on an event when that action uses a restricted feature listed here.

The recommended means to send events from queries to monitors is by sending to a channel. See [“Generating events with the send statement” on page 265](#).

The debugger does not support debugging query execution - it is not possible to set breakpoints in a query file. Use of the debugger can also affect how quickly queries are ready to respond to events, and should not be used in a production system (where it would cause significant pauses of the correlator).

Note:

Several restrictions are enforced on queries if a license file cannot be found while the correlator is running. See "Running Apama without a license file" in *Introduction to Apama*.

Best practices for defining queries

Use values for the length of the window that will not store too much data in the window.

Given the expected incoming event rate, set the `within` and/ or `retain` window lengths so that typically less than a hundred events per partition will be within the window. With more than that the cost of executing queries can become excessive and the system will not perform efficiently. There is no limit on the number of events within any partition. If a very small proportion of exceptional partitions has many more, then that is not a problem. The important factor is that if the average number is large, this can affect the performance of executing queries.

Use parameters instead of creating many similar queries.

Rather than writing many separate queries which are very similar in structure and differ only in values, it may be easier to write a template query and create multiple parameterizations of it. See also [“Parameterized queries as templates” on page 130](#).

If a query requires different fields for its keys depending on the query parameters, it should use an action as a query key. See also [“Defining actions as query keys” on page 73](#).

Use `within` in input durations if the partition values change over time

In some queries, the key used by the query may correspond to a transient object, that is, any given value for the partition is not permanent. For example, if tracking parcels being delivered, then each consignment ID will be short-lived. Once a parcel is delivered, there would in most cases be no more events for that consignment ID (and future deliveries may never re-use the same consignment ID). In these cases, over long periods, the number of different key values processed will only increase, as new IDs are generated. Such queries should include a `within` specification in the inputs for all event types. Otherwise, if inputs only have a `retain` specification, then the events will be held forever, and more and more storage will be required by the queries system. This is not typically necessary if the key corresponds to more permanent objects, such as ATMs or distribution depots.

Use input within that is larger than the value of all waits, within in the pattern

If your inputs specify a `within` and there is `wait` or `within` in the pattern, then the input `within` should be larger than the longest `wait` and `within` in the pattern. If not, the pattern will not have the intended effect, as events will be expired from the input window while a `wait` or `within` in the pattern is still active.

Use same set of inputs to allow sharing of data

If you have many queries of different types and they are using a lot of memory or are running slowly, then check if they are using the same inputs definitions (see also [“Queries can share windows” on page 79](#)). Memory usage can be reduced and performance increased by making multiple queries use the same set of input definitions, even if some queries have some event types in their inputs that they are not using.

Understand the difference between filters and where clauses

Filters in the input section filter events before they are stored in the distributed cache. By contrast, the `where` clause filters events (or combinations of events) after they have been stored in the distributed cache. The `where` clause is more powerful, but also more expensive, especially if most events do not match the `where` clause.

- A filter applies before the event window. Thus:
 - Events not matching the filter are ignored and do not need to be stored anywhere. This makes filtering a very cheap way of reducing the number of events that need to be processed. The retain count only applies to the events that match the filter. For example, the following query input will match events where there have been two events with `value = 5`; it will match if another event for the same `k` has occurred between them with `value` not equal to 5.

```
query Q1 {
  inputs {
    Event( value = 5) key k retain 2;
  }
  find Event as e1 -> Event as e2 {
  }
}
```

Compare the above with:

```
query Q2 {
  inputs {
    Event() key k retain 2;
  }
  find Event as e1 -> Event as e2 where e1.value = 5 and e2.value = 5 {
  }
}
```

This only matches if the last two events for a given value of `k` both have the value 5 - as we only retain 2 events and after retaining 2 events, check that they have `value = 5`.

- A filter applies to all events. Note that in query Q2 above we had to repeat the `value = 5` check.

- A where clause does not affect the definition of the inputs. Query Q2 could share window contents with other queries that are concerned with different values of value, or do not filter at all.
- A filter is restricted to range or equality matches per field of the incoming events. where clauses can be more complex (for example, where `e1.field1 + e2.field2 = 10` is valid, as is `e1.isTypeA` or `e1.isTypeB`; but neither could be expressed in a filter).

Avoid changing parameter values used in filters

If using parameters in filters, avoid changing the values of those parameters. As this changes which events should be stored in the window, this is similar in effect to stopping a query instance and creating a new query instance. It involves creating new tables in the distributed cache, and events that are delivered to correlators while a new table is opened will be dropped. It may be more desirable to use a where clause to restrict which events match a pattern.

Use custom aggregates to get data from multiple match sets

As well as the built-in aggregates, it is possible to define new aggregates in EPL to collate information about all events that matched a pattern. For example, it may be desirable to have a list of all events that matched a pattern. This can be achieved by writing a new custom aggregate. For example:

```
// file MyAggregates.mon:
aggregate CollateEvents(Event e) returns sequence<Event> {
    sequence<Event> allEvts;
    action add(Event e) {
        allEvts.append(e);
    }

    action value() returns sequence<Event> {
        return allEvts;
    }
}
```

```
// file PrintAllEvents.qry:
query PrintAllEvents {
    inputs {
        Event() within 2 hours;
    }

    find every Event as e1 select CollateEvents(e1) as c1 {
        Event e;
        for e in c1 {
            print e.toString();
        }
    }
}
```

Testing query execution

When writing queries, as with any programming, it is important to test that the query is behaving as expected. Testing can be as simple as a small Apama project with the event definitions, the queries, and an `.evt` file of events to send to the query. You can use this project to check whether the query sends out the correct events. In Software AG Designer, use the **Engine Receive** view to observe the output of the query. Whether or not a query is written to send output events, you can add `log` statements to the query file to verify whether it has or has not triggered.

Be sure to test queries in an environment that is separate from your production environment. Of course, preventing problems is the best way to avoid the need to troubleshoot so ensure that queries are sufficiently tested before deploying them.

The following background information and troubleshooting tips provide some guidance. See also [“Overview of query processing” on page 64](#).

Exceptions in queries

In a query, exceptions can occur in the following places:

- Procedural code in a `find` statement block
- `having` clause
- `retain` clause
- `select` clause
- `wait` clause
- All `where` clauses
- All `within` clauses

An exception in the `inputs` block (`retain` or `within` clause) or the `find` block's `wait` or `within` clause causes the query to terminate. If there is an exception elsewhere, the query continues to process incoming events. An exception that occurs in a `where` or `having` clause causes the Boolean expression to evaluate to `false`.

Event ordering in queries

Unlike EPL monitors, the order in which queries process events is not necessarily the order in which they were sent into the correlator. In particular, if two events that will be processed by the same query with the same key value are sent very close together in time (both events received less than about .1 seconds of each other) then they may be processed as if they had been sent in a different order. For example, consider a query that is looking for an `A` event followed by an `A` event. If two `A` events with the same key arrive 1 millisecond apart then the events might not be processed in the order in which they were sent.

Queries use multiple threads to process events and to scale across multiple correlators on multiple machines. To do this efficiently, there is no enforcement that the events are processed in order.

However, when events that have the same key arrive roughly about .5 seconds apart or more then out-of-order processing is typically avoided provided the system can keep up with the load. Therefore, you want to specify a query so that it operates on partitions in which the arrival of consecutive events is spaced far enough apart. For example, consider a query that operates on credit card transaction events, which could mean thousands of events per second. You want to partition this query on the credit card number so that there is one event or less per partition per second. By following this recommendation, it becomes possible to process events that are generated at rates of up to 10,000 events per second.

When creating an .evt file for testing purposes, the recommendation is to begin the file with a `&FLUSHING(1)` line to cause more predictable and reliable event-processing behavior. See "Event timing" in *Deploying and Managing Apama Applications*.

Query diagnostics

To help you monitor queries that are running on a given correlator, Apama provides data about active queries in DataViews. See "Monitoring running queries" in *Deploying and Managing Apama Applications*.

When deploying Apama queries it is possible to enable generation of diagnostic information. These are `log` statements that explain some of the internal workings of the query evaluation. In particular, events coming into the query and the contents of the windows before the pattern is evaluated are both logged. This can aid understanding of how the query evaluation occurs. If a query is misbehaving then providing this diagnostics logging to Apama support can help in understanding the issue.

Note:

Diagnostic logs contain the event data. You may want to consider using fake data rather than real data if the real data is sensitive.

Logging in `where` statements

It can be useful to modify a query so that rather than including the expression that needs to be evaluated in a `where` clause, the query calls an action on the query to execute the expression used by the `where` clause. This allows logging of inputs and the result of the expression. For example, instead of a query that contains the following:

```
find A as a -> B as b where a.x >= b.x { ...
```

Write the query this way:

```
action compareAB(A a, B b) returns boolean {
  log "compareAB; inputs:  A as a = "+a.toString()+ ", B as b = "+b.toString();
  boolean r:= (a.x >= b.x);
  log "compareAB; result is "+r.toString();
  return r;
}

find A as a -> B as b where compareAB(a, b) { ...
```

You can then use these `log` statements to check if the query is behaving as expected.

Divide and conquer

One of the advantages of testing a query with a known set of input events is that it is possible to see how changing the query affects the results. For example, if a query is not matching any events and has many `within` and `without` clauses, try removing all of them. One way to do this is to place them onto separate lines and use `//` as a comment at the beginning of the lines in the source view. If the query still does not fire, use query diagnostics to check that events are being evaluated. If the query is firing, then add `within` and `without` clauses one at a time until the query stops firing. The problem is at the condition that stops it from firing when it should.

Query performance

A critical factor that affects the performance of queries is the size of the windows specified in the `inputs` block of the query. Aim for windows that contain no more than 100 events. Depending on the distributed cache used to store data, it may also be necessary to change the number of parallel contexts per correlator. Experiment with different values for the number of worker contexts. See also [“Overview of query processing” on page 64](#).

Using external clocking when testing

When testing queries, as well as switching into single context execution, it is often useful to use external clocking. This allows `&TIME` events to be sent into the correlator to simulate the passage of time, which allows queries involving long durations (for example, multiple days) to be tested easily. To ensure the correct ordering of processing between events and `&TIME` events, you should also include `&FLUSHING(1)` at the beginning of the event file, before any events. See [“Externally generating events that keep time \(&TIME events\)” on page 183](#) and “Event timing” in *Deploying and Managing Apama Applications*.

Communication between monitors and queries

Queries can be used with or without monitors written in EPL. The following statements can be used to send events between queries, or between queries and monitors or vice versa:

- **route statement.** A `route` statement from a query sends the event to be processed by other query instances. This is the recommended mechanism for sending events between queries. See also [“The route statement” on page 649](#).

The `route` statement cannot be used to send events to a monitor.

- **send...to statement.** A `send...to` statement can be used to send an event to all Apama queries running on that correlator by sending it to the `com.apama.queries` channel or the default channel. To send the event to a monitor, send it to a channel the monitor is listening to. See also [“The send...to statement” on page 649](#).

Note:

Queries receive events sent to the default channel, which is useful for testing.

The order in which events are processed is not guaranteed for queries. See [“Event ordering in queries” on page 138](#).

In case the events are expected to be received by a monitor, the monitor author should make it clear which channel they are expecting events on. The channel name can be a single name for a given monitor or a name constructed from data in the event, so that different values are processed in parallel.

If you are using multiple correlators, be aware that communication between queries and monitors normally takes place within a single correlator. However, it is possible to use `engine_connect` or Universal Messaging to connect correlators. This allows an event sent on a channel on one correlator to be processed by a monitor subscribed to that channel on another correlator.

Unlike a query's history window, any state stored in EPL monitors, including in the listeners, is independent in each correlator, and is not automatically moved or shared between correlators.

4 Defining Event Listeners

■ About event expressions and event templates	144
■ Specifying the on statement	147
■ Using a stream source template to find events of interest	148
■ Defining event expressions with one event template	148
■ Terminating and changing event listeners	153
■ Specifying multiple event listeners	155
■ Listening for events that do not match	156
■ Specifying completion event listeners	157
■ Improving performance by ignoring some fields in matching events	159
■ Defining event listeners for patterns of events	160
■ Specifying and/or/not logic in event listeners	162
■ How the correlator executes event listeners	167
■ Defining event listeners with temporal constraints	174
■ Understanding time in the correlator	179
■ Out of band connection notifications	185

In an EPL monitor, an `on` statement specifies an event expression and a listener action.

Note:

Queries do not need to set up event listeners so you cannot specify an `on` statement in a query. The information about defining event listeners applies only to monitors.

When the correlator executes an `on` statement it creates an event listener. An event listener observes each event processed by the context until an event or a pattern of events matches the pattern specified in the event listener's event expression. When this happens the event listener triggers, causing the correlator to execute the listener action. At this point, depending on the form of the event expression, the event listener either terminates or continues listening for additional matching event patterns.

An event listener analyzes the event stream until one of the following happens:

- The event listener finds the pattern defined in its event expression.
- The `quit()` method is called on the event listener to kill it.
- The monitor that defines the event listener dies.
- The correlator determines that the event listener can never trigger.

The correlator can support large numbers of concurrent event listeners each watching for an individual pattern.

About event expressions and event templates

To create an event listener, you must specify an event expression. An event expression

- Identifies an event or event pattern that you want to match
- Contains zero or more event templates
- Contains zero or more event operators

An event template specifies an event or any type and encloses in parentheses the set of, or set of ranges of, event field values to match. An event template can specify wildcards for event fields or can specify that certain event fields must have particular values or ranges of values.

An event expression can specify a temporal operator and zero event templates.

Following are event expressions that are each made up of one event template:

Event Expression	Description
<code>StockTick(*,*)</code>	The event listener that uses this event expression is interested in all <code>StockTick</code> events regardless of the event's field values.
<code>NewItem("ACME",*)</code>	The event listener that uses this event expression is interested in <code>NewItem</code> events that have a value of <code>ACME</code>

Event Expression	Description
	in their first field. Any value can be in the second field.
<code>ChainedResponse(reqId="req1")</code>	The event listener that uses this event expression is interested in <code>ChainedResponse</code> events whose <code>reqId</code> field has a value of <code>req1</code> . If a <code>ChainedResponse</code> event has any other fields, their values are irrelevant.
<code>any()</code>	The event listener that uses this event expression is interested in all events, regardless of their type.

You can specify more than one event template in an event expression by adding event operators. The following table describes the operators that you can use in an event expression.

Category	Operator	Operation
Followed by	<code>-></code>	The event listener detects a match when it finds an event that matches the event template specified before the followed-by operator and later finds an event that matches the event template that comes after the followed-by operator.
Repeat matching	<code>all</code>	The event listener detects a match for each event that matches the specified event template. The event listener does not terminate after the first match.
Logical operators	<code>and</code>	Logical intersection. The event listener detects a match after it finds events that match the event templates on both sides of the <code>and</code> operator. The order in which the listener detects the matching events does not matter.
	<code>not</code>	Logical negation. The event listener detects a match only if an event that matches the event template that follows the <code>not</code> operator has not occurred.
	<code>or</code>	Logical union. The event listener detects a match as soon as it finds an event that matches one of the event templates on either side of the <code>or</code> operator.
	<code>xor</code>	Logical exclusive or. The event listener detects a match if it finds an event that matches exactly one of the event templates on either side of the <code>xor</code> operator. For example, consider this event: <code>A(1,1)</code> . This event does not trigger the following listener because it matches the event templates on both sides of the <code>xor</code> operator: on <code>A(1,*)</code> <code>xor</code> <code>A(*,1)</code> .
Temporal operators	<code>at</code>	The event listener triggers at specific times or repeatedly at a specified interval.

Category	Operator	Operation
	<code>wait</code>	Limits the amount of time that an event listener can detect a match.
	<code>within</code>	The event listener can find a match only within the specified timeframe.

Consider the following example:

```
event Test
{
    float f;
}

monitor RangeExample
{
    action onload()
    {
        on Test (f >= 9.0 ) and Test (f <= 10.0) processTest();
    }

    action processTest();
    {
        do something    }
}
```

The event expression is:

```
Test (f >= 9.0 ) and Test (f <= 10.0)
```

This event expression specifies the `and` operator so the event listener must detect an event that matches both of the event expression's event templates or two events, where one matched the first template and another matched the second. It does not have to be a single event that matches both event templates. The order in which the templates are matched does not matter.

Consider this event expression:

```
A(a = "foo") xor A(b > 9)
```

An event listener that defines this event expression triggers for `A("foo", 9)` but not `A("foo", 10)`. On `A("foo", 10)`, the `A` templates would trigger simultaneously, so the `xor` would remain false.

The correlator can match on up to 32 fields per event. If you specify an event template for an event that has more than 32 fields, you must ensure that the correlator maintains indexes for the particular fields for which you specify values that you want to match.

In other words, when the event definition was loaded into the correlator, the fields that did not have the `wildcard` keyword formed the set of fields that you can match on. An event template can try to match on only those fields. If an event template specifies any of the `wildcard` fields, it must be with an asterisk.

If you try to load a monitor that defines an event template that specifies more than 32 fields without an asterisk or a `wildcard` field without an asterisk, the correlator rejects the monitor. You must correct the template in order to load the monitor.

Specifying the on statement

You specify an on statement to define an event listener. The format of an on statement is as follows:

```
[listener identifier := ] on event_expr [ coassignment ] listener_action;
```

Syntax description

Syntax Element	Description
<i>identifier</i>	Optionally, you can specify a variable of type <code>listener</code> and assign the new event listener to that variable. This gives you a handle to the event listener — if you want to terminate it you can call the <code>quit()</code> method on the listener.
<i>event_expr</i>	The event expression identifies the event or pattern of events that you want to match. An event expression is made up of one or more event templates and zero or more event operators.
<i>coassignment</i>	Optionally, you can coassign the matching event to a variable. Use the <code>as</code> operator to implicitly declare the variable in the scope of the following <i>listener_action</i> , or the <code>:</code> assignment operator to coassign to a local or global variable of the same event type. Coassignments are part of event templates. Each event template can have a coassignment, so there can be multiple coassignments in a listener.
<i>listener_action</i>	The statement or block that you want the correlator to perform when the event listener triggers.

Examples

In the following example, the event expression contains one event template: `StockTick(*,*)`. The asterisks indicate that the values of the `StockTick` event's two fields are not relevant when matching. When this event listener detects a `StockTick` event, the listener triggers and causes the correlator to execute the `processTick()` listener action.

```
on StockTick(*,*) processTick();
```

Following is an example that implicitly declares the `newTick` variable in the scope of the *listener_action*.

```
on StockTick(*,*) as newTick {
    processTick(newTick);
}
```

Following is an example that explicitly coassigns the matching event to the `newTick` variable. The `newTick` variable must be a `StockTick` event type variable. Coassignment simply assigns the event to the variable.

```
on StockTick(*,*):newTick processTick();
```

The next example begins with the declaration of a `listener` variable. The statement assigns the event listener to the `l` variable.

```
listener l := on StockTick(*,*) as newTick processTick();
```

Suppose that after finding a matching event, the listener action includes specification of an `on` statement. For example:

```
listener l := on StockTick(*,*) as newTick {  
    on StockTick(newTick.symbol, > newTick.value) as risingTick {  
        processRisingTick();  
    }  
}
```

The correlator creates an entirely new event listener to handle the nested `on` statement. This new event listener is completely independent of the enclosing event listener. For example, the enclosing event listener does not wait for the nested event listener to find a matching event.

Using a stream source template to find events of interest

In addition to event listeners, EPL provides stream source templates for finding events of interest. A stream source template is an event template prefixed with the `all` keyword. The result of a stream source template is a stream.

Use streams on a continuous flow of incoming items when you want to aggregate, join to other streams, and/or narrow the scope of the matching items based on content, arrival time, or the most recent particular number of items.

Use an event listener for discrete events or discrete patterns of events for which you want to independently trigger the listener action.

For information about using stream source templates, see [“Working with Streams and Stream Queries” on page 187](#).

Defining event expressions with one event template

This section provides examples of specifying event expressions that contain just one event template. It is important to understand the various ways that you can specify a single event template. When you are familiar with this, it is easier to start applying operators and combining multiple event templates in an event expression.

Listening for one event

Consider the following on statement:

```
on StockTick() processTick();
```

This event listener is watching for one `StockTick` event. The values of the `StockTick` event's fields are irrelevant, as indicated by the empty parentheses. When this event listener finds a `StockTick` event, it triggers and terminates. When the event listener triggers, it causes the correlator to execute the `processTick()` action.

Listening for all events of a particular type

Consider the straightforward case where an event expression consists of a single event template. When the event listener finds an event that matches its event template, the event listener triggers, and the correlator executes the listener action. Since the event listener has found the event it was looking for, it terminates.

In some situations, you might want the event listener to continue watching for the same event so that you can act on each one. You do not want the event listener to terminate after it finds one event. In this situation, specify the `all` keyword before the event template, as in the following example:

```
on all StockTick() processTick();
```

When the `all` operator appears before an event template, when that event template finds a match, it continues to watch for subsequent events that also match the template.

Listening for events with particular content

The sample monitor is very simple. It just logs all `StockTick` events. The content of the `StockTick` event is not relevant when matching. See [“Example of a simple monitor” on page 37](#). However, you can filter events according to their content. To alter the example so that the monitor logs only `StockTick` events for a given stock, you must specify a filter on the first field in the event template. For example, suppose you want to log only ACME stock ticks. You need to change the following line:

```
on all StockTick(*,*) :newTick processTick();
```

to this:

```
on all StockTick("ACME",*) :newTick processTick();
```

Now the event listener triggers on only `StockTick` events whose `name` field matches ACME.

To filter `StockTick` events based on their price, you might specify the event template shown below. This event template specifies that you are interested in all `StockTick` events whose price is 50.5 or greater.

```
on all StockTick(*, >=50.5) :newTick processTick();
```

Using positional syntax to listen for events with particular content

You can specify that you want to listen for `StockTick` events that have a particular name and a particular price. In the `on` statement below, the event listener is looking for `StockTick` events in which the name is `ACME` and the price is 50.5 or less.

```
on all StockTick("ACME", <=50.5) as newTick processTick();
```

When you specify this syntax, called positional syntax, the event template must define a value (or a wildcard) to match against for every field of that event's type. You must specify these values in the same order as the fields in the event type definition. Consider the following event type:

```
event MobileUser {
    integer userID;
    location position;
    string hairColour;
    string starsign;
    integer gender;
    integer incomeBracket;
    string preferredHairColour;
    string preferredStarsign;
    integer preferredGender;
}
```

Following is an event listener definition for this event type:

```
on MobileUser(*,*, "red", "Capricorn", *, *, *, *, 1) some_action();
```

Using name/value syntax to listen for events with particular content

Specification of every field in an event can get unwieldy when you are working with event types with a large number of fields and you are specifying values for only a few of them. In this case, you can use the name/value syntax in which you specify only the fields of interest. In the name/value syntax, it is as if you had specified a wildcard (*) for each field for which you do not specify a value. For example:

```
on MobileUser(hairColour="red", starsign="Capricorn",
    preferredGender=1) some_action();
```

The table below shows equivalent event expressions that demonstrate how to specify each syntax. The table uses these event types:

```
event A {
    integer a;
    string b;
}

event B {
    integer a;
}
```

```
event C {
    integer a;
    integer b;
    integer c;
}
```

Comparison Criterion	Positional Syntax	Equivalent Name/Value Syntax
Equality	on A(3,"string")	on A(a=3,b="string")
	on A(=3,="string")	on A(b="string",a=3)
Relational comparisons	on B(>3)	on B(a>3)
Ranges	on B([2:3])	on B(a in [2:3])
Wildcards	on C(*,4,*)	on C(b=4)
	on C(*,*,*)	on C(a=*,b=4,c=*)
		on C()

For details about the operators and expressions that you can specify in event templates, see [“Expressions” on page 659](#).

It is possible to mix the two syntax styles as long as you specify all positional fields before named fields. For example:

- Correct event template: on D(3,>4,i in [2:4])
- Incorrect event template: on D(k=9,"error")

Listening for events of different types

A monitor is not limited to listening for events of only one type. A single monitor can listen for any number of event types. The following sample monitor uses the `StockTick` event type and the `StockChoice` event type, which specifies a stock name. When the event listener finds a `StockChoice` event, a second event listener then looks for only stocks that match the name in the `StockChoice` event.

```
// Definition of a type of event that the correlator will receive.
// These events represent stock ticks from a market data feed.
event StockTick {
    string name;
    float price;
}

// Definition of a type of event that describes the stock to process.
// These events come from a second live data feed.
event StockChoice {
    string name;
}

// The following simple monitor listens for two different event types.

monitor SimpleShareSearch {
```

```
// A global variable to store the matching StockTick event:
StockTick newTick;

// A global variable to store the StockChoice event:
StockChoice currentStock;

// Wait for a StockChoice event and use its name field to
// filter for StockTick events.
action onload() {
    on StockChoice(*) : currentStock {
        on all StockTick(currentStock.name, *) : newTick processTick();
    }
}

action processTick() {
    log "StockTick event received" +
        " name = " + newTick.name +
        " Price = " + newTick.price.toString() at INFO;
}
}
```

The differences between the sample in [“Example of a simple monitor” on page 37](#) and this monitor are the following:

- Definition of an additional event type (StockChoice)
- Definition of a new global variable (currentStock)
- A more complex onload() action

While the first two changes are straightforward, the new onload() action introduces new behavior. The first line in the onload() action is similar to that in the earlier example. In the new example, the monitor creates an event listener for a StockChoice event. The content of the StockChoice event is not relevant when matching. When the event listener finds an event of this type, it stores the value of the StockChoicename field in the currentStock variable and triggers the creation of a second event listener.

In this example, the first event listener defines the action of creating the second event listener in-line. The first event listener looks for a StockChoice event. The second event listener looks for all StockTick events whose name field corresponds to the value of currentStock.name.

Listening for events of all types

You can specify a listener which listens for all events types as shown below:

```
on any() as anyVar {
    print "Event received : " + anyVar.toString();
}
```

In the above listener, the on any() event template will match against all events in the corresponding context, regardless of their type.

You can provide an optional typeName to specify which type you are listening for. This must name a routable type.

You can provide an optional values. This must be a dictionary<string, any> which names indexable fields. The listener will filter to only match events where all of the specified fields are equal. If you specify values, you must also specify the typeName. For example:

```
dictionary<string, any> values := {"quantity", 100};
on any(typeName="Event", values = values)
```

The above definition is equivalent to:

```
on Event(quantity=100)
```

Example:

```
event A{}
event B{}

on all any() as anyVar {
    print "Event received : " + anyVar.toString();
}

route A(); //listener fires
route B(); //listener fires
```

You can use all event expression operators (all, and, not, -> and so on) with the any listener.

Note:

Due to unpredictable results, it is recommended that you do not use any() listeners in onLoad actions. Instead, do so in response to an event (a start event or, more likely, a configuration event) once everything has been injected.

Terminating and changing event listeners

After the correlator creates an event listener, you cannot change it. Instead of changing an event listener, you terminate it and create a new one.

The example in [“Listening for events of different types” on page 151](#) looks for only one StockChoice event. The monitor would be more useful if it continued looking for subsequent StockChoice events, and on every new StockChoice event it changed the second event listener to look for stock ticks for the new company.

When the correlator creates an event listener, it copies from the action the value of any local variables. However, if the variable is of a reference type, changes to the object referred to by the value are seen by other listeners.

The steps and example below shows how to terminate an event listener with the quit() operation. See also, [“Specifying and not logic to terminate event listeners” on page 164](#).

When you want to change an event listener, do the following:

1. Obtain a handle to the event listener you want to change.
2. Terminate that event listener with the quit() operation.

3. Create a new event listener to take its place.

The following sample monitor does just this.

```
// Definition of a type of event that the correlator will receive.
// These events represent stock ticks from a market data feed.
event StockTick {
    string name;
    float price;
}

// Definition of a type of event that describes the stock to process.
// These events come from a second live data feed.
event StockChoice {
    string name;
}

// The following simple monitor listens for two different event types.

monitor SimpleShareSearch {
    // A global variable to store the matching StockTick event:
    StockTick newTick;

    // A global variable to store the StockChoice event:
    StockChoice currentStock;

    // A handle to the second listener:
    listener l;

    // Record the latest StockChoice event and use its name field
    // to filter the StockTick events.
    action onload() {
        on all StockChoice(*):currentStock {
            l.quit();
            l := on all StockTick(currentStock.name, *):newTick processTick();
        }
    }

    action processTick() {
        log "StockTick event received" +
            " name = " + newTick.name +
            " Price = " + newTick.price.toString() at INFO;
    }
}
```

The differences between the example in [“Listening for events of different types” on page 151](#) and this example are as follows:

- The monitor in this example declares an additional global variable, `l`, whose type is `listener`.
- The initial `on` statement now specifies the `all` operator. After this event listener finds a `StockChoice` event, it watches for the next `StockChoice` event.
- The `onload()` action specifies a new listener action. Each time the first event listener finds a `StockChoice` event, the listener action:

- Terminates the second event listener by calling the `l.quit()` method. Of course, upon finding the first `StockChoice` event there is no second event listener to terminate. This is not a problem as in this case the `l.quit()` method does not do anything.
- Creates a new event listener to seek `StockTick` events for the company named in the `StockChoice` event just detected.
- Stores a handle to the new event listener in the `l` global variable. The first event listener uses this handle when it needs to terminate the second event listener.

Specifying multiple event listeners

When the correlator encounters an `on` statement, it creates an event listener to watch for events that match the event expression specified in the `on` statement. When the event listener finds a matching event, the event listener triggers and the correlator executes the listener action. Ordinarily the event listener then dies. That is, the event listener processes only a single matching event.

When you require multiple matching events specify the `all` operator before the template for the event for which you want multiple matches. This prevents termination of the event listener upon an event match.

Another way to match multiple events is to define two (or more) event listeners for the same event type. If you specify two `on` statements that require the same event, they both trigger when they find that event. The order in which they trigger is not defined. For example:

```
on all StockTick(*,*) as newTick1 { print newTick1.name; }
on all StockTick(*,*) as newTick2 { print newTick2.name; }
```

When the correlator receives a single `StockTick` event, the correlator populates both the `newTick1` variable and the `newTick2` variable with the event value. The correlator then prints the value of the `name` field in each variable. This means that an event of the format `StockTick("ACME", 50.10)` causes this output:

```
ACME
ACME
```

Adding further `on` statements to those above would increase the number of times the string `ACME` is printed. This is true regardless of where (that is, in which action) the `on` statements are defined. For example:

```
action action1() {
    on all StockChoice("ACME") as currentStock processTick();
}
action action2() {
    on all StockChoice("ACME") as currentStock processTick();
}
```

If both the `action1()` and `action2()` actions have been invoked, both will invoke the `processTick()` action when an "ACME" `StockChoice` event is received.

Now consider the following example:

```
on all StockTick("ACME", *) action1();
on all StockTick(*,50.0) action1();
```

The event `StockTick("ACME", 50.0)` will trigger both event listeners. It is not possible to determine which one will execute the action first but the actions will be executed atomically. That is, the correlator will start executing `action1()`, finish it, and only then will the correlator execute `action1()` again. The correlator processes only one listener action at a time.

See [“Spawning monitor instances” on page 39](#) for another way to have multiple event listeners.

Listening for events that do not match

Sometimes it is useful to catch events that do not match other event templates. To do this, specify the unmatched keyword in an event template. An unmatched event template matches against events for which both of the following are true:

- Except for `completed` and `unmatched` event templates, the event does not cause any other event expression in the same context as the unmatched event template to match. For information about `completed` event templates, see the next topic.
- The event matches the unmatched event template.

The correlator processes an event as follows:

1. The correlator tests the event against all normal event templates. Normal event templates do not specify the `completed` or `unmatched` keyword.
2. If the correlator does not find a match, the correlator tests the event against all event templates that specify the `unmatched` keyword. If the correlator finds one or more matches, the matching event templates now evaluate to `true`. That is, if there are multiple `unmatched` event templates that match the event, they all evaluate to `true`.

The scope of an unmatched event template is the context that contains it. Suppose an event goes to two contexts. In one context, there is a matching event listener and in the other context there is a match against an unmatched event template. Both matches trigger the listener actions.

Specify the unmatched keyword with care. Be sure to communicate with other developers. If your code relies on an unmatched event template, and someone else injects a monitor that happens to match some events that you expected to match your unmatched event template, you will not get the results you expect. The firing of an event-specific unmatched listener is suppressed if an `on any()` listener matches the event (not just a type-specific listener).

A typical use of the unmatched keyword is to spawn a monitor instance to process a particular subset of events. For example:

```
event Tick{ string stock; ... }
monitor TickProcessor {
    Tick tick;
    ...
    action onload() {
        on all unmatched Tick():tick spawn processTick();
    }
    action processTick() {
        on all Tick( stock=tick.stock ) ...;
    }
    ...
}
```

```
}
```

You can use the `unmatched` keyword with an `any()` listener as shown below:

```
on unmatched any() {
    print "An unmatched any() listener triggered!";
}
```

An unmatched `any()` event template will match against all unmatched events (as defined above) in the context, regardless of the event's type.

See also:

- [“Example using unmatched and completed” on page 158.](#)
- [“Writing echo monitors for debugging” on page 335](#)

Specifying completion event listeners

In some situations, you want to ensure that the correlator completes all work related to a particular event before your application performs some other work. In your event template, specify the `completed` keyword to accomplish this. For example:

```
on all completed A(f < 10.0) {}
```

Suppose an `A` event whose `f` field value is less than 10 arrives in the correlator. What happens is as follows:

1. If there are normal or `unmatched` event listeners whose event expression matches this `A` event, those event listeners trigger.
2. The correlator executes listener actions and then processes any routed events that result from those actions, and any routed events that result from processing the routed events, and so on until all routed events have been processed.
3. The `completed` event listener triggers.

A common situation in which the `completed` keyword is useful is when a piece of data comes into the system and that piece of data causes a cascade of event listeners to trigger. Each listener action updates some data. When all listener actions have been executed, you want to take a survey of the new state of things and do something in response.

For example, consider a pricing engine made up of many individual pricing engines. When a new piece of market data arrives all pricing engines update their prices and then the controller uses some metric to pick the best price, which it publishes. The controller should publish the new price only after all individual engines have updated their output. The controller achieves this by listening for all the updates but only publishing when the market data event causes the `completed` event listener to trigger. The EPL for this scenario follows.

```
// Request/return best price from *all* markets
event RequestSmartBestPrice{ string stock; integer id; }
event BestSmartPriceReply{ integer id; float price; }
```

```
//Request/return best price from individual market(s)
event RequestBestPrice{ string stock; integer id; }
event BestPriceReply{ integer id; float price; }

// Simple example: Assume 'best' is 'lowest' and no account
// is taken of 'side'.
monitor SmartPriceGetter {
    RequestSmartBestPrice request;
    sequence< float > prices;

    action onload() {
        on all RequestSmartBestPrice(*,*):request spawn getPrices();
    }

    action getPrices() {
        on all BestPriceReply( request.id, * ) as reply
            prices.append(reply.price);
        on completed RequestSmartBestPrice( request.stock, request.id ) {
            prices.sort();
            route BestSmartPriceReply( request.id, prices[0]);
            die;
        }
        route RequestBestPrice( request.stock, request.id );
    }
}
```

You can use the completed keyword with an any() listener as shown below:

```
on completed any() {
    print "A completed any() listener triggered!";
}
```

A completed any() event template will match against all events in the context once they are processed, regardless of their type.

Example using unmatched and completed

The following example shows the use of both the unmatched and completed keywords. After the example, there is a discussion of the processing order.

```
on all A("foo", < 10) as a {
    print "Match: " + a.toString();
    a.count := a.count+1; // count is second field of A
    route A;
}

on all completed A("foo", < 10) as a {
    print "Completed: " + a.toString();
}

on all unmatched A(*,*)as a {
    print "Unmatched: " + a.toString();
}
```

The incoming events are as follows:

```
A("foo", 8);
```

```
A("bar", 7);
```

The output is as follows:

```
Match: A("foo", 8)
Match: A("foo", 9)
Unmatched: A("foo", 10)
Completed: A("foo", 9)
Completed: A("foo", 8)
Unmatched: A("bar", 7)
```

`A("foo", 8)` is the first item on the queue. The correlator processes all matches for this event except for any matching on `completed` expressions. The correlator processes those after it has processed all routed events originating from `A("foo", 8)`, which includes the processing of all routed events produced from all routed events produced from `A("foo", 8)`, and so on.

Correlator processing goes like this:

1. Processing of `A("foo", 8)` routes `A("foo", 9)` to the front of the queue.
2. Processing of `A("foo", 9)` routes `A("foo", 10)` to the front of the queue.
3. Processing of `A("foo", 10)` finds a match with the unmatched event expression.
4. All routed events that resulted from `A("foo", 9)` have now been processed. The completed `A("foo", 9)` event template now matches so the correlator executes its listener action.
5. All routed events that resulted from `A("foo", 8)` have now been processed. The completed `A("foo", 8)` event template now matches so the correlator executes its listener action.
6. Processing of `A("bar", 7)` matches the unmatched `A(*,*)` event template and the correlator executes its listener action.

For another example of the use of `unmatched` and `completed`, see [“Writing echo monitors for debugging” on page 335](#).

Improving performance by ignoring some fields in matching events

In applications where a particular field of an event type will never participate in the match criteria for that event type, the performance of Apama can be improved (at times drastically) by marking that field as a wildcard field in the event type definition.

For example, consider a version of the `StockTick` event type that has four fields: `name`, `volume`, `price`, and `source`. If in our application `volume` and `source` are never going to be used for matching on within event templates, that is, they will always be marked as `*` (wildcard), they could be tagged so explicitly in the event type:

```
event StockTick {
    string name;
    wildcard float volume;
    float price;
    wildcard string source;
}
```

The `wildcard` keyword tells Apama not to include this field in its internal indexing, as it will never be required in a match operation. This not only saves memory, but can significantly improve performance, particularly when there are many such fields which never occur in match conditions. Note that removing fields from an event type altogether is even more efficient than using `wildcard`, but this is not always possible. For example, the field might not be relevant in match conditions but it might be input to calculations within an action block, or it might need to be included in an event specified in a `send...to` statement.

When a field has been declared as a `wildcard` then any subsequent attempt to define a match condition using that field will result in a parser error, and the offending monitor will not be injected.

Therefore, given the above event type definition, the following will result in a parser error:

```
action someAction() {
    on StockTick("ACME", >125.0,*, "NASDAQ") doSomething();
}
```

while the following is correct:

```
action someAction() {
    on StockTick("ACME", *, 50.0, *) doSomething();
}
```

Defining event listeners for patterns of events

One way to search for an event pattern in EPL is to define an event listener to search for the first event, and then, in that listener action, define a second event listener to search for the second event in the pattern, and so on.

However, the `on` statement takes an event expression, and this can be more than just a single event template.

Consider the following very simple example: locate a news event about ACME followed by a stock price update for ACME. With the EPL explored so far, one would write this as:

```
event StockTick {
    string name;
    float price;
}

event NewsItem {
    string subject;
    string newsHeading;
}

monitor NewsSharePriceSequence_ACME {
    // Look for a news item about ACME, if successful execute the
    // findStockChange() action
    //
    action onload() {
        on NewsItem("ACME",*) findStockChange();
    }

    // Look for a StockTick about ACME, if successful execute the
    // notifyUser() action
```



```
//
action findStockChange() {
    on StockTick("ACME",*) notifyUser();
}

// Print a message, event sequence detected
//
action notifyUser() {
    log "Event sequence detected.";
}
}
```

If, as in this example, you do not intend to express any custom actions after finding an event other than searching for another event, the whole pattern of events to look for can be encoded in a single event expression within a single `on` statement.

An event expression can define a pattern of events to match against. Each event of interest is represented by its own event template. You can apply several constraints on the temporal order that the events have to occur in to match the event expression.

In the more declarative syntax of an event expression, the above monitor would be written as follows:

```
event StockTick {
    string name;
    float price;
}

event NewsItem {
    string subject;
    string newsHeading;
}

monitor NewsSharePriceSequence_ACME {
    // Look for a NewsItem followed by a StockTick
    action onload() {
        on NewsItem("ACME",*) -> StockTick("ACME",*)
        notifyUser();
    }

    // Print a message, event sequence detected
    //
    action notifyUser() {
        log "Event sequence detected.";
    }
}
```

Here, instead of just one event template, the `on` keyword is now followed by an event expression that contains two event templates.

The primary operator in event expressions is `->`. This is known as the followed-by operator. It allows you to express a pattern of events to match against in a single `on` statement, with a single action to be executed at the end once the whole pattern is encountered.

In EPL, an event pattern does not imply that the events have to occur right after each other, or that no other events are allowed to occur in the meantime.

Let A, B, C and D represent event templates, and A', B', C' and D' be individual events that match those templates, respectively. If a monitor is written to seek (A > B), the event feed {A', C', B', D'} would result in a match once the B' is received by Apama.

Followed-by operators can be chained to express longer patterns. Therefore, one could write:

```
on A -> B -> C -> D executeAction();
```

Notes:

- An event template is in fact the simplest form of an event expression. All event expression operators, including `->`, actually take event expressions as operands. So in the above representation, A, B, C, D could in fact be entire nested event expressions rather than simple event templates.
- It is useful to think of event expressions as being Boolean expressions. Each clause in an event expression can be `true` or `false`, and the whole event expression must evaluate to `true` before the event listener triggers and the action is executed.

Consider the above event expression: A -> B -> C -> D

The expression starts off as `false`. When an event that satisfies the A event template occurs, the A clause becomes `true`. Once B is satisfied, A -> B becomes `true` in turn, and evaluation progresses in a similar manner until eventually all of A -> B -> C -> D evaluates to `true`. Only then does the event listener trigger and cause execution of the listener action. Of course, this event expression might never become `true` in its entirety (as the events required might never occur) since no time constraint (see [“Defining event listeners with temporal constraints” on page 174](#)) has been applied to any part of the event expression.

Specifying and/or/not logic in event listeners

When the correlator creates an event listener each event template in the event expression is initially `false`. For an event listener to trigger on an event pattern, the event expression defining what to match against must evaluate to `true`. Consequently, in an event expression, you can specify logical operators.

Specifying the 'or' operator in event expressions

The `or` operator lets you specify event expressions where a variety of event patterns could lead to a successful match. It effectively evaluates two event templates (or entire nested event expressions) simultaneously and returns `true` when either of them becomes `true`.

For example,

```
on A() or B() executeAction();
```

means that either A or B need to be detected to match. That is, the occurrence of one of the operand expressions (an A or a B) is enough for the event listener to trigger.

Specifying the 'and' operator in event expressions

The `and` operator specifies an event pattern that might occur in any temporal order. It evaluates two event templates (or nested event expressions) simultaneously but only returns `true` when they are both true.

```
on A() and B() executeAction();
```

This will seek “an “A” followed by a “B or “a “B” followed by an “A. Both are valid matching patterns, and the event listener will seek both concurrently. However, the first to occur will terminate all monitoring and cause the event listener to trigger.

Example event expressions using 'and/or' logic in event listeners

The following example event expressions indicate a few patterns that can be expressed by using `and/or` logic in event listeners.

Event Expression	Description
<code>A -> (B or C)</code>	Match on an A followed by either a B or a C.
<code>(A -> B) or C</code>	Match on either the pattern A followed by a B, or just a C on its own.
<code>A -> ((B -> C) or (C -> D))</code>	Find an A first, and then seek for either the pattern B followed by a C or C followed by a D. The latter patterns will be looked for concurrently, but the monitor will match upon the first complete pattern that occurs. This is because the <code>or</code> operator treats its operands atomically, that is, in this case it is looking for the patterns themselves rather than their constituent events.
<code>(A -> B) and (C -> D)</code>	Find the pattern A followed by a B (that is, <code>A -> B</code>) followed by the pattern <code>C -> D</code> , or else the pattern <code>C -> D</code> followed by the pattern <code>A -> B</code> . The <code>and</code> operator treats its operands atomically. That is, in this case it is looking for the patterns themselves and the order of their occurrence, rather than their constituent events. It does not matter when a pattern starts but it occurs when the last event in it is matched. Therefore <code>{A', C', B', D'}</code> would match the specification, because it contains an <code>A -> B</code> followed by a <code>C -> D</code> . In fact, the specification would match against either of the following patterns of event instances; <code>{A', C', B', D'}</code> , <code>{C', A', B', D'}</code> , <code>{A', B', C', D'}</code> , <code>{C', A', D', B'}</code> , <code>{A', C', D', B'}</code> and <code>{C', D', A', B'}</code>

Specifying the 'not' operator in event expressions

The not operator is unary and acts to invert the truth value of the event expression it is applied to.

```
on ((A() -> B()) and not C()) executeAction();
```

therefore means that the event listener will trigger `executeAction` only if it encounters an A followed by a B without a C occurring at any time before the B is encountered.

The not operator can cause an event expression to reach a state where it can never evaluate to true. That is, it becomes permanently false.

Consider the above event listener event pattern: `on (A() -> B()) and not C()`

The event listener starts by seeking both `A -> B` and `not C` concurrently. If an event matching C is received before one matching B, the C clause evaluates to true, and hence `not C` becomes false. This means that `(A -> B) and not C` can never evaluate to true, and hence this event listener will never trigger. The correlator terminates these zombie event listeners periodically.

It is possible to specify the not operator in an event expression in such a way that the expression always evaluates to true immediately. Since this triggers the specified action without any events occurring, you want to avoid doing this. See [“Avoiding event listeners that trigger upon instantiation” on page 168](#).

Specifying 'and not' logic to terminate event listeners

A typical situation is that you want to listen for a pattern only until a particular condition occurs. When the condition occurs you want to terminate the event listener. In pseudocode, you want to specify something like this:

```
on all event_expression until stop_condition
```

To define an event listener that behaves this way, you specify `and not`:

```
on all event_expression and not stop_condition
```

The following example listens for a price increase for a particular stock while the market is open.

```
event Price {
    string stock;
    float price;
}
on all Price("IBM",>targetPrice) as p and not MarketClosed() {
    ...do something}
```

When you inject a monitor that contains this code, the correlator sets up an event template to listen for Price events and another event template to listen for MarketClosed events. As long as the correlator does not receive a MarketClosed event, `not MarketClosed()` evaluates to true. While `not MarketClosed()` evaluates to true, each time the correlator receives a Price event for IBM stock at a price that is greater than `targetPrice`, this event expression evaluates to true and triggers its listener action. When the correlator receives a MarketClosed event, `MarketClosed()` evaluates to true and so `not MarketClosed()` evaluates to false. At that point, the event expression can no

longer evaluate to true. When the correlator recognizes an event listener that can never trigger, it terminates it. In other words, after the market is closed the event listener terminates.

Typically, the stop condition is a condition that applies to multiple entities. In the previous example, the condition applies to only IBM stock, but it could easily be rewritten to apply to all stocks.

Pausing event listeners

You can also specify `and not` when you want to listen for a pattern, pause when a particular condition occurs, and resume listening for that pattern when some other condition occurs. Consider the example that terminates the event listener after the market closes. Suppose instead that you want to listen for increases in stock prices only when there is no auction. When the correlator receives an `InAuction` event, you want to pause the event listener and when the correlator receives an `AuctionClosed` event you want the event listener to become active again. To do this, you can write something like the following:

```
action initialize() {
    on EndAuction() and not BeginAuction() notInAuctionLogic();
    on BeginAuction() and not EndAuction() inAuctionLogic();
    route RequestAuctionPhase();
}

action inAuctionLogic() {
    on EndAuction() notInAuctionLogic();
}

action notInAuctionLogic() {
    on all Price("IBM",>targetPrice) as p and not BeginAuction()
        sellStock();
    on BeginAuction() inAuctionLogic();
}
```

The `initialize()` action sets up two event listeners that determine whether to start with the `inAuctionLogic()` action or the `notInAuctionLogic()` action. The response to the routed `RequestAuctionPhase` event is an `EndAuction` event or a `BeginAuction` event. As soon as one of these events arrive, both event listeners terminate. For example, if an `EndAuction` event arrives, the first event listener terminates because its `EndAuction()` event template evaluates to true and its `not BeginAuction()` event template also evaluates to true. The second event listener terminates because its `not EndAuction()` event template evaluates to false and so the event expression can never evaluate to true.

Choosing which action to execute

Another situation in which `and not` logic can help terminate event listeners is when you want to specify a choice of one or more actions and terminate the event listeners after one is chosen. An example of this appears below. This is the CEP equivalent of a case statement.

```
on Pattern_1() and not PatternMatched() processCase1();
on Pattern_2() and not PatternMatched() processCase2();
on Pattern_3() and not PatternMatched() processCase3();
on Pattern_1() or Pattern_2() or Pattern_3()
{
    route PatternMatched();
}
```

When you inject a monitor that contains this type of code the correlator immediately sets up multiple event listeners. For the example in [“Pausing event listeners” on page 165](#), the event listeners would be watching for these events:

- Pattern_1
- PatternMatched
- Pattern_2
- Pattern_3

Initially, all `and` and `not` event templates evaluate to `true`. Suppose `Pattern_2` arrives. This causes these two event listeners to trigger:

```
on Pattern_2() and not PatternMatched() processCase2();  
on Pattern_1() or Pattern_2() or Pattern_3()
```

It is unknown which event listener action the correlator executes first, but the order does not matter. The correlator does all of the following:

- The correlator executes the `processCase2()` action.
- The correlator terminates the event listener that specifies `processCase2()` because it has found its match and it does not specify `all`.
- The correlator routes a `PatternMatched` event to the front of the context's input queue.

When the correlator processes the `PatternMatched()` event, the two event templates that are still watching for `and not PatternMatched` become `false`. Consequently, those event listeners will never trigger and the correlator terminates them.

Following is another example of specifying `and not` to make a choice:

```
on Ack() and not Nack()  
{  
    processAck();  
}  
on Nack() and not Ack()  
{  
    processNack();  
}
```

Specifying 'and not' logic to detect when events are missing

Using `and not` logic with a time-based listener is useful for detecting the absence of an event that is expected.

For example, consider an application that monitors the processing of customer orders. The application listens for `OrderCreate` events, which indicate that a customer has placed an order. After an `OrderCreate` event is found, the application listens for an `OrderStepComplete` event that has an `instanceid` value that matches the `instanceid` value in the `OrderCreate` event and that has a `step` field value of `Order Shipped`. If the application does not find a matching `OrderStepComplete` event within an hour (3600 seconds), the listener triggers and the application generates an alert to indicate that the order was not shipped.

Following is code that shows the listener definition.

```
on all OrderCreate() as oc {
  on wait(3600.00) and not OrderStepComplete(
    instanceid=oc.instanceid,step="Order Shipped") as os {
    // Raise an alert.
  }
}
```

This listener triggers when the event templates on both sides of the `and` operator evaluate to true. The event template `before and` evaluates to true after an hour has elapsed. The event template `after and` evaluates to true in the absence of a matching `OrderStepComplete` event. If the application finds a matching `OrderStepComplete` event within an hour then the second event template evaluates to false and the correlator terminates the listener because it can never trigger.

In the following example, when a `FileReceived` event is found, the application starts to listen for a `FileProcessed` event. If a `FileProcessed` event is not found within 30 seconds of receiving the `FileReceived` event, the application generates an alert.

```
monitor SimpleFileSearch {
  action onload() {
    on all FileReceived() as f {
      on wait(30.0) and not FileProcessed(id=f.id) {
        // Send alert that file was not processed.
      }
      on FileProcessed(id=f.id) within(30.0) {
        // Send confirmation that the file was processed.
      }
    }
  }
}
```

How the correlator executes event listeners

An understanding of how the correlator executes event listeners can help you correctly define event listeners. The topics below provide the needed background.

How the correlator evaluates event expressions

When the correlator processes an injection request, it executes the monitor's `onload()` statement, which typically defines an event listener. To understand how the correlator evaluates the event expression in the event listener, consider the following `on` statement:

```
on A()->B() and C()->D() processOrder();
```

The event expression consists of four templates and three operators. The operators are:

```
->
and
->
```

The correlator does not evaluate the right operand of a followed by operator until after its left operand has evaluated to true. Hence, `B` and `D` are not evaluated initially but will only be evaluated

after A and C, respectively, have become `true`. Initially, the correlator evaluates the A and C event templates.

Suppose a C event arrives first. The C part of the event expression is now `true` and the correlator now evaluates the A and D event templates. Now suppose an A event arrives next. The correlator evaluates the B and D event templates. When a B event arrives the first term, `A()->B()`, of the event expression becomes `true`. Finally a D event arrives and the second term, `B()->D()` becomes `true` and so the expression as a whole evaluates to `true`. The event listener triggers.

As mentioned before, when the correlator instantiates an event listener each event template in the event listener is initially `false`. An event template changes to `true` when the correlator finds a matching event. In a given context, the correlator cannot find a matching event while it is setting up an event listener because the correlator processes only one thing at a time in each context. Everything happens in order and no two things happen simultaneously in a given context.

Of course, events are always coming into the correlator. These events go on the input queue of each public context to wait their turn for processing. So while a matching event might arrive while the correlator is setting up an event listener, as far as correlator processing is concerned, the event arrives later. See [“Understanding time in the correlator” on page 179](#).

Avoiding event listeners that trigger upon instantiation

Because all event templates are initially `false`, it is important to think carefully before specifying not in an event expression. It is easy to inadvertently specify the `not` operator in such a way that the expression evaluates to `true` immediately upon instantiation. Since this triggers the specified action without any events occurring, it is unlikely to be what you intended and you want to avoid doing this. Consider the following example:

```
on ( A() -> B() ) or not C() myAction();
```

Assuming that A, B and C represent event templates, the value of each starts as being `false`. This means that `not C` is immediately `true`, and hence the whole expression is immediately `true`, which triggers the specified action. If any of A, B or C is a nested event expression the same logic applies for its evaluation. Typically, the `not` operator is used in conjunction with the `and` operator. See [“Choosing which action to execute” on page 165](#).

When an event listener triggers the correlator sends a request to the front of the context's input queue to execute the event listener action. For example:

```
route D();
on not E() {
    print "not E";
}
route F();
```

The `route` keyword sends the specified event to the front of the context's input queue. The correlator processes this code in the following order:

1. The correlator processes event D.
2. The correlator prints "not E".
3. The correlator processes event F.

When the correlator terminates event listeners

The correlator terminates event listeners in the following situations:

- The event listener's event expression evaluates to `true`, and does not specify the `all` keyword. The correlator executes the specified action. Since the single defined match was found, the correlator terminates the event listener.
- The correlator recognizes that an event listener's event expression can never evaluate to `true`. For example:

```
on ( A() -> B() ) and not C()
```

The event listener starts by seeking both `A() -> B()` and `not C()` concurrently. If an event matching `C` is received before one matching `B`, the `C` clause evaluates to `true`, and hence `not C` becomes `false`. This means that `(A() -> B()) and not C()` can never evaluate to `true`, and hence this event listener will never trigger its action. The correlator terminates these zombie event listeners periodically.

- You obtain a handle to an event listener and call the `quit()` method on that event listener. See [“Terminating and changing event listeners” on page 153](#).

How the correlator evaluates event listeners for a series of events

Suppose there are seven event templates defined, which are represented as `A`, `B`, `C`, `D`, `E`, `F` and `G`. Now, consider a series of incoming events, where x_n indicates an event instance that matches the event template x . Likewise, x_{n+1} indicates another event instance that matches against x , but which need not necessarily be identical to x_n .

Consider the following pattern of incoming events:

```
C1 A1 F1 A2 C2 B1 D1 E1 B2 A3 G1 B3
```

Given the above event pattern, what should the event expression `A() -> B()` match upon?

In theory the combinations of events that correspond to “an `A` followed by a `B`” are `{A1, B1}`, `{A1, B2}`, `{A1, B3}`, `{A2, B1}`, `{A2, B2}`, `{A2, B3}` and `{A3, B3}`. In practice it is unlikely that you want your event listener to match seven times on the above example pattern, and it is uncommon for all the combinations to be useful.

In fact, within EPL, `on A() -> B()` will only match on the first instance that matched the template. Given the above event pattern the event listener will trigger only on `{A1, B1}`, execute the associated action and then terminate.

Evaluating event listeners for all A-events followed by B-events

You might want to alter the behavior described in the previous topic, and have the event listener match on more of the combinations. To do this, specify the `all` operator in the event expression.

If the event listener's specification was rewritten to read:

```
on all A() -> B() success();
```

the event listener would match on every A and the first B that follows it.

The way this works is that upon encountering an A, the correlator creates a second event listener to seek the next A. Both event listeners would be active concurrently; one looking for a B to successfully match the pattern specified, the other initially looking for an A. If more As are encountered the procedure is repeated; this behavior continues until either the monitor or the event listener are explicitly killed.

Therefore `on all A() -> B()` would return {A1, B1}, {A2, B1} and {A3, B3}.

Note that `all` is a unary operator and has higher precedence than `->`, `or` and `and`. Therefore

```
on all A() -> B()
```

is the same as both of the following:

```
on (all A()) -> B()  
on ( ( all A() ) -> B() )
```

The following table illustrates how the execution of `on all A() -> B()` proceeds over time as the pattern of input events is processed by the correlator. The timeline is from left to right, and each stage is labeled with a time t_n , where t_{n+1} occurs after t_n . To the left are listed the event listeners, and next to each one (after the ?) is shown what event template that event listener is looking for at that point in time. In the example, assuming L was the initial event listener, L', L'' and L''' are other sub-event-listeners that are created as a result of the `all` operator.

Guide to the symbols used:

This symbol	indicates the following
-------------	-------------------------



A specific point in time when a particular event is received.



No match was found at that time.



The listener has successfully located an event that matches its current active template.

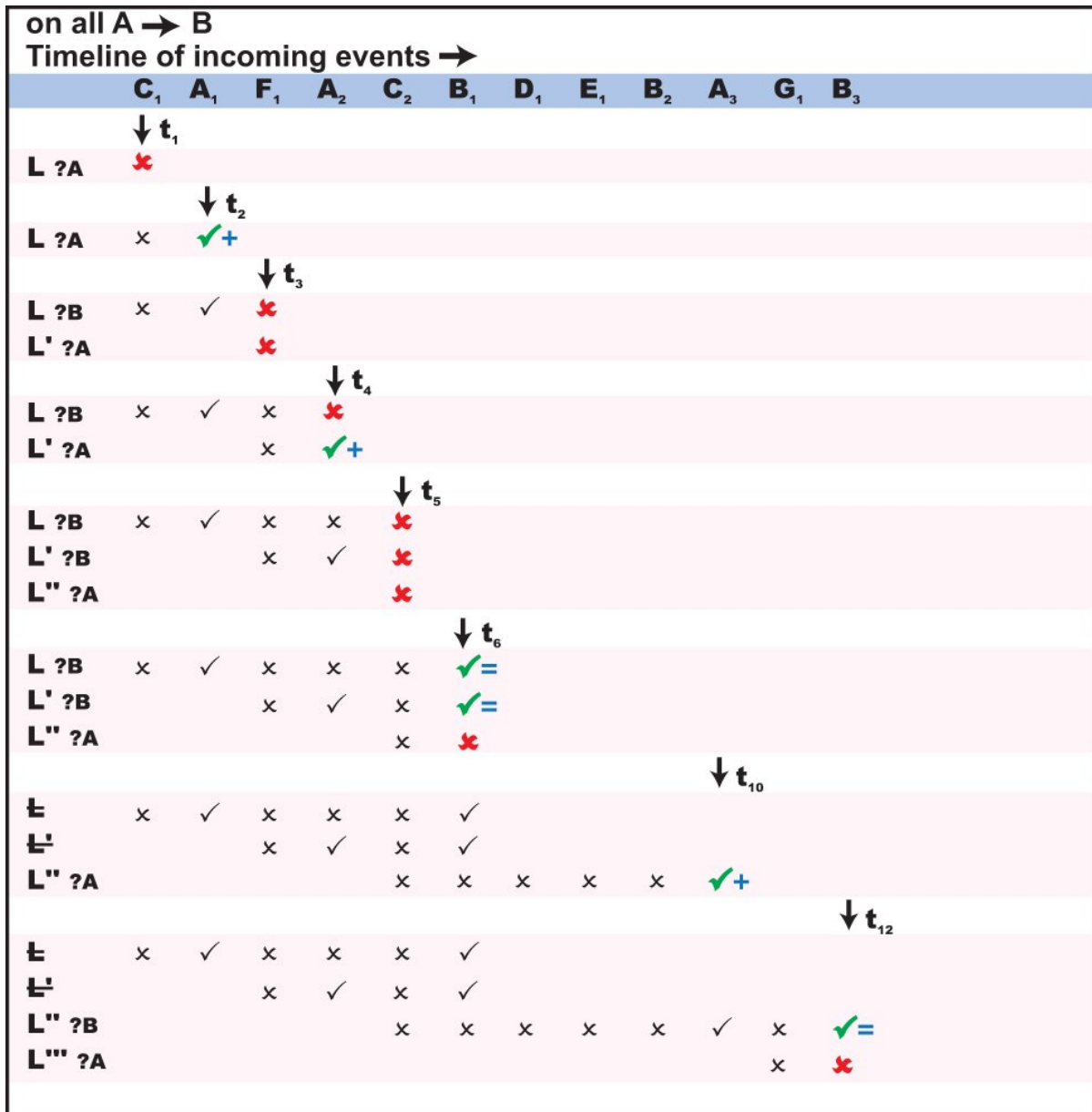


A listener has successfully triggered.



A new listener is going to be created.

The parent event listener denoted by `on all A() -> B()` will never terminate as there will always be a sub-event-listener active that is looking for an A.



Evaluating event listeners for an A-event followed by all B-events

Consider an event listener defined as follows:

```
on A() -> all B() success();
```

The monitor would now match on all the patterns consisting of the first A and each possible following B.

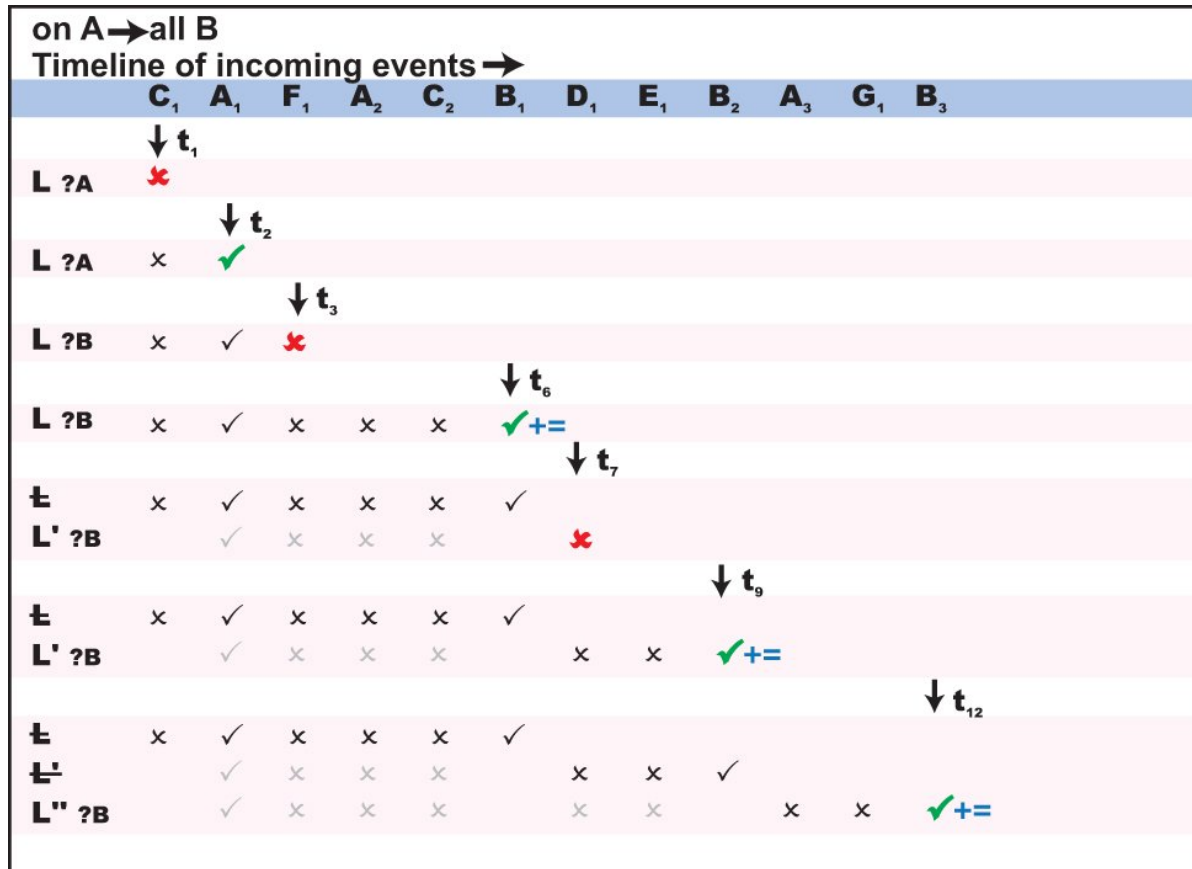
For clarity this is the same as:

```
on ( A() -> ( all B() ) ) success();
```

The way this works is that the correlator creates a second event listener after finding a matching B. The second event listener watches for the next B, and so on repeatedly until the monitor is explicitly killed.

Therefore on $A() \rightarrow \text{all } B()$ would match $\{A_1, B_1\}$, $\{A_1, B_2\}$ and $\{A_1, B_3\}$.

Graphically this would now look as follows:



The table shows the early states of L' and L'' in light color because those event listeners actually never really went through those states themselves. However, since they were created as a clone of another event listener, it is as though they were.

The parent event listener denoted by $\text{on } A() \rightarrow \text{all } B()$ will never terminate as there will always be a sub-event-listener looking for a B.

Evaluating event listeners for all A-events followed by all B-events

Consider the following event listener definition:

```
on all A() -> all B() success();
```

or

```
on ( ( all A() ) -> ( all B() ) ) success();
```

Now the monitor would match on an A and create another event listener to look for further As. Each of these event listeners will go on to search for a B after it encounters an A. However, in this instance all event listeners are duplicated once more after matching against a B.

The effect of this would be that `on all A -> all B` would match {A1, B1}, {A1, B2}, {A1, B3}, {A2, B1}, {A2, B2}, {A2, B3} and {A3, B3}. That is, all the possible permutations. This could cause a very large number of sub-event-listeners to be created.

Note:

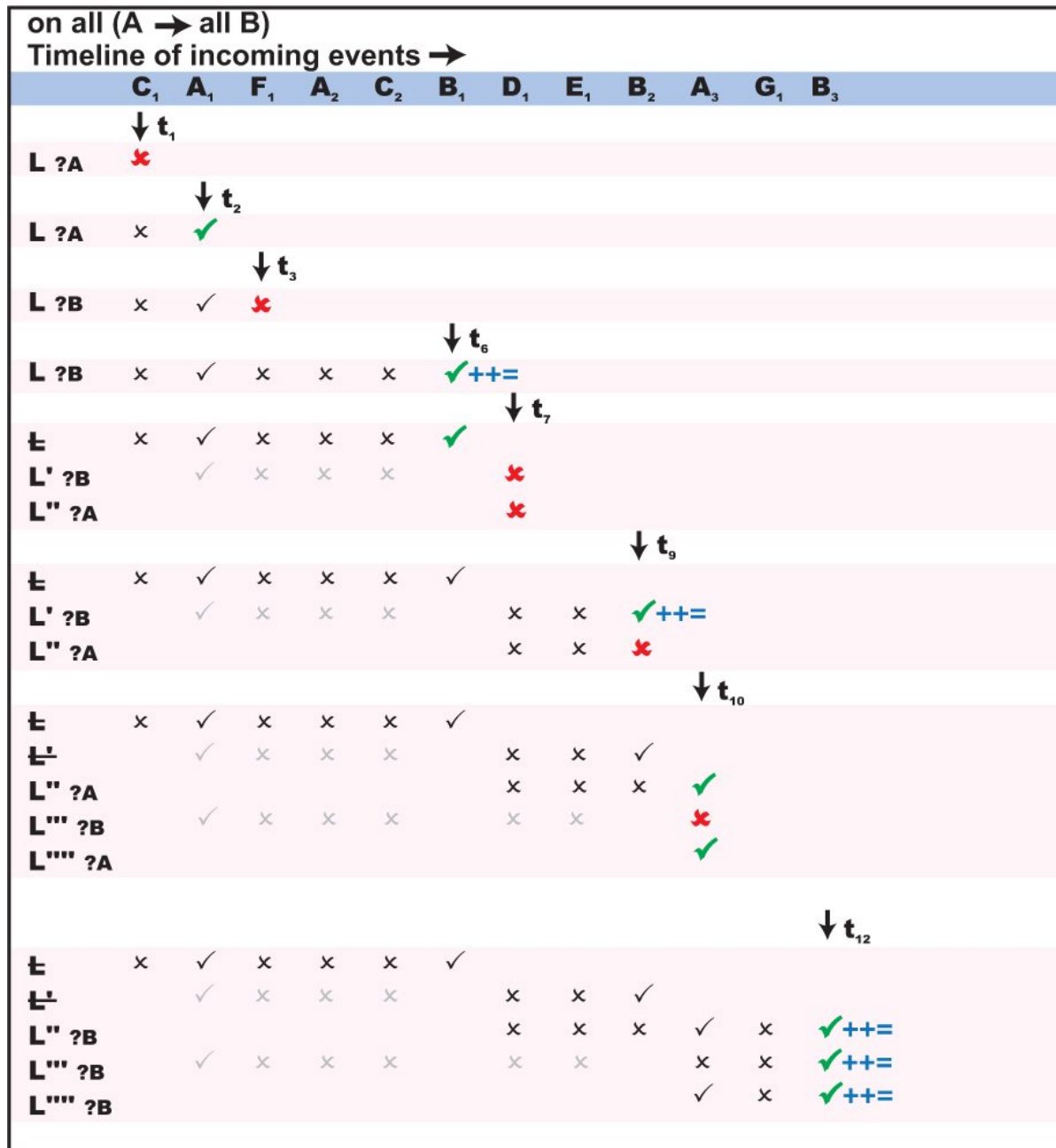
The `all` operator must be used with caution as it can create a very large number of sub-event-listeners, all looking for concurrent patterns. This is particularly applicable if multiple `all` operators are nested within each other. This can have an adverse impact on performance.

Now consider the example,

```
on all ( A() -> all B() ) success();
```

This will match the first A followed by all subsequent Bs. However, as on every match of an A followed by B, `(A() -> all B())` becomes `true`, then a new search for the "next" A followed by all subsequent Bs will start. This will repeat itself recursively, and eventually there could be several concurrent sub-event-listeners that might match on the same patterns, thus causing duplicate triggering.

Give the same event pattern as described in [“Evaluating event listeners for all A-events followed by B-events” on page 169](#), this would be evaluated as follows:



Thus matching against $\{A_1, B_1\}$, $\{A_1, B_2\}$, $\{A_1, B_3\}$, and twice against $\{A_3, B_3\}$. Notice how the number of active event listeners is progressively increasing, until after t_{12} there would actually be six active event listeners, three looking for a B and three looking for an A.

Defining event listeners with temporal constraints

So far this section has shown how to use event expressions to define interesting patterns of events to look for, where the events of interest depend not only on their type and content, but also on their temporal relationship to (whether they occur before or after) other events.

Being able to define temporal relationships can be useful, but typically it also needs to be constrained over some temporal interval.

Listening for event patterns within a set time

Consider this earlier example:

```
event StockTick {
    string name;
    float price;
}

event NewsItem {
    string subject;
    string newsHeading;
}

monitor NewsSharePriceSequence_ACME {
    // Look for a NewsItem followed by a StockTick
    //
    action onload() {
        on NewsItem("ACME",*) -> StockTick("ACME",*)
        notifyUser();
    }

    // Print a message, event sequence detected
    //
    action notifyUser() {
        log "Event sequence detected.";
    }
}
```

This will look for the event pattern of a news item about a company followed by a stock price tick about that company. Once improved this could be used to detect the beginning of a rise (or fall) in the value of shares of a company following the release of a relevant news headline.

However, unless a temporal constraint is put in place, the monitor is not going to be that pertinent, as it might trigger on an event pattern where the price change occurs weeks after the news item. That would clearly not be so useful to a trader, as the two events were most likely unrelated and hence not indicative of a possible trend.

If the event listener above is rewritten as follows,

```
on NewsItem("ACME",*) -> StockTick("ACME",*) within(30.0)
    notifyUser();
```

the `StockTick` event would now need to occur within 30 seconds of `NewsItem` for the event listener to trigger.

The `within(float)` operator is a postfix unary operator that can be applied to an event template (the `StockTick` event template in the above example). Think of it like a stopwatch. The clock starts ticking as soon as the event listener starts looking for the event template that the `within` operator is attached to. If the stopwatch reaches the specified figure before the event template evaluates to true, then the event template becomes permanently false.

In the above code, the timer is only activated once a suitable `NewItem` is encountered. Unless an adequate `StockTick` then occurs within 30 seconds and makes the expression evaluate to `true`, the timer will fire and fail the whole event listener.

You can apply the `within` operator to any event template. For example:

```
on A() within(10.0) listenerAction();
```

After the correlator sets up this event listener, the event listener must detect an `A` event within 10 seconds. If no `A` event is detected within 10 seconds, the event expression becomes permanently `false` and the correlator subsequently terminates the event listener.

Waiting within an event listener

The second timer operator available for use within event expressions is `wait(float)`.

The `wait` operator lets you insert a 'temporal pause' within an event expression. Once activated, a `wait` expression becomes `true` automatically once the specified amount of time passes. For example:

```
on A() -> wait(10.0) -> C() success();
```

Execution of this event listener proceeds as follows:

1. Set up an event template to watch for an `A` event.
2. After detecting an `A` event, wait 10 seconds. Set up an event template to watch for a `C` event.

In addition to being part of an event expression, `wait` can also be used on its own.

```
on wait(20.0) success();
```

When the correlator instantiates this event listener the event listener just waits for the number of seconds specified (here being 20), then it evaluates to `true`, triggers, and causes the correlator to execute the `success()` action.

Therefore a `wait` clause starts off being `false`, and then turns to `true` once its time period expires. This behavior can be inverted through use of `not`. The expression `not wait (20.0)` would start off being `true`, and stay `true` for 20 seconds before becoming `false`.

Consider the following example:

```
on B() and not wait(20.0) success();
```

This event listener triggers only if a `B` event is detected within 20 seconds after the correlator sets up the event template that watches for `B` events. After 20 seconds, the `not wait(20.0)` clause would become `false` and prevent the event listener from ever triggering. This would therefore be the same as

```
on B within(20.0) success();
```

By using `all` with `wait`, you can easily implement a periodic repeating timer,

```
on all wait(5.0) success();
```


This event listener triggers every 5 seconds and causes the correlator to execute the `success()` action each time.

See also [“Specifying 'and not' logic to detect when events are missing” on page 166](#).

Triggering event listeners at specific times

The `at` temporal operator lets you express temporal activity with regards to absolute time. The `at` operator allows triggering of a timer:

- **At a specific time**, for example, 12:30pm on the 5th April.
- **Repeatedly** with regards to the calendar when used in conjunction with the `all` operator, across seconds, minutes, hours, days of the week, days of the month, and months, for example, on every hour, or on the first day of the month, or every 10 minutes past the hour and every 40 minutes past the hour.

Important:

Triggering using the `at` operator always uses the time zone in which the correlator is running. If the time zone contains time changes (for example, Daylight Saving Time), then a listener which would trigger during a period of time which is skipped (when the clocks go forward) will not trigger, since that time did not occur. Listeners which would trigger during a repeated section of time (when the clocks go back) will trigger for both the first time and the second time, since that time occurred twice. This is true for all patterns using the `at` operator.

The syntax of the `at` operator is as follows:

```
at(minutes, hours, days_of_month, months, days_of_week [ ,seconds])
```

where the last operand, *seconds*, is optional.

Valid values for each operand are as follows:

Operand	Values
<i>minutes</i>	0 to 59, indicating minutes past the hour.
<i>hours</i>	0 to 23, indicating the hours of the day.
<i>days_of_month</i>	1 to 31, indicating days of the month. For some months only 1 to 28, 1 to 29 or 1 to 30 are valid ranges.
<i>months</i>	1 to 12, indicating months of the year, with 1 corresponding to January
<i>days_of_week</i>	0 to 6, indicating days of the week, where 0 corresponds to Sunday.
<i>seconds</i>	0 to 59, indicating seconds past the minute.

The `at` operator can be embedded within an event expression in a manner similar to the `wait` operator. If used outside the scope of an `all` operator it will trigger only once, at the next valid time as expressed within its elements. In conjunction with an `all` operator, it will trigger at every valid time.

The wildcard symbol (*) can be specified to indicate that all values are valid, for example:

```
on at(5, *, *, *, *) success();
```

would trigger at the next “five minutes past the hour”, while

```
on all at(5, *, *, *, *) success();
```

would trigger at five minutes past each hour (that is, every day, every month).

Whereas,

```
on all at(5, 9, *, *, *) success();
```

would trigger at 9:05am every day. However,

```
on all at(5, 9, *, *, 1) success();
```

would trigger at 9:05am only on Mondays, and never on any other week day. This is because the effect of the wildcard operator is different when applied to the *days_of_week* and the *days_of_month* operands. This is due to the fact that both specify the same entity. The rule is therefore as follows:

- As long as both elements are set to wildcard, then each day is valid.
- If either of the *days_of_week* or the *days_of_month* operand is not a wildcard, then only the days that match that element will be valid. The wildcard in the other element is effectively ignored.
- If both the *days_of_week* and the *days_of_month* operands are not wildcards, then the days valid will be the days which match either. That is, the two criteria are ‘or’ed, not ‘and’ed.

A range operator (:) can be used with each element to define a range of valid values. For example:

```
on all at(5:15, *, *, *, *) success();
```

would trigger every minute from 5 minutes past the hour till 15 minutes past the hour.

A divisor operator (/integer, x) can be used to specify that every x'th value is valid. Therefore

```
on all at(* / 10, *, *, *, *) success();
```

would trigger every ten minutes, that is, at 0, 10, 20, 30, 40 and 50 minutes past every hour.

If you wish to specify a combination of the above operators you must enclose the element in square braces ([]), and separate the value definitions with a comma (.). For example

```
on all at([* / 10, 30:35, 22], *, *, *, *) success();
```

indicates the following values for minutes to trigger on; 0, 10, 20, 30, 40 and 50, from 30 to 35, and specifically the value 22.

A further example,

```
on all at(* / 30, 9:17, [* / 2, 1], *, *) success();
```

would trigger every 30 minutes from 9am to 5pm on even numbered days of the month as well as specifically the first day of the month.

Using variables to specify times

If you wish to programmatically parameterize usage of the `at` operator, you have to use variables in conjunction with it. You can replace any of the parameters to the `at` operator with a string variable or with a sequence of integer variables.

The first alternative, using a string variable, allows you to define the matching criteria within a string variable and then specify the variable within the `at` call. The following example shows how this can be done. Each of the parameters can be replaced with a string variable in this way.

```
string minutes = "*/30";
on all at(minutes,9:17,[*/2,1],*,*) success();
```

The other alternative is to use a sequence of integer variable. This is only useful when you want to specify a selection of valid values for the parameter.

```
sequence<integer> days = new sequence<integer>;
days.append(1); // Monday is ok
days.append(3); // and so is Wednesday
on all at(*,*,*,*,days) success();
```

See also the description of the sequence type in the *API Reference for EPL (ApamaDoc)*.

Understanding time in the correlator

An understanding of how the correlator handles time is essential to writing Apama applications. The topics below discuss time in the correlator.

See also "System clock" in *Installing Apama*.

Correlator timestamps and real time

When the correlator receives an event, it gives the event a timestamp that indicates the time that the correlator received the event. The correlator then places the event on the input queue of each public context. The correlator processes events in the order in which they appear on input queues.

An input queue can grow considerably. In extreme cases, this might mean that a few seconds pass between the time an event arrives and the time the correlator processes it. As you can imagine, this has implications for whether the correlator triggers listeners. However, the correlator uses event timestamps, and not real time, to determine when to trigger listeners.

As an extreme example, suppose a monitor is looking for `A -> B within(2.0)`. The correlator receives event A. However, the queue has grown to a huge size and the correlator processes event A three seconds after event A arrives. The correlator receives event B one second after it receives event A. Some events in the queue before event B cause a lot of computation in the correlator. The result is that the correlator processes event B five seconds after event B arrives. In short, event B arrives one second after event A, but the correlator processes event B three seconds after it processes event A.

If the correlator used real time, `A -> B within(2.0)` would not be triggered by this pattern. This is because the correlator processes event B more than two seconds after processing event A. However, the correlator uses the timestamp to determine whether to trigger actions. Consequently, `A -> B within(2.0)` does trigger, because the correlator received event B one second after event A, and so their timestamps are within 2 seconds of each other.

As you can see, the number of events on an input queue never affects temporal comparisons.

Event arrival time

As mentioned before, when an event arrives, the correlator assigns a timestamp to the event. The timestamp indicates the time that the event arrived at the correlator. If you coassign an event to a variable, the correlator sets the timestamp of the event to the current time in the context in which the coassignment occurs. For JMon applications, this is always the current time in the main context.

The correlator uses clock ticks to specify the value of each timestamp. The correlator generates a clock tick every tenth of a second. The value of an event's timestamp is the value of the last clock tick before the event arrived.

When you start the correlator, you can specify the `--frequency hz` option if you want the correlator to generate clock ticks at an interval other than every tenth of a second. Instead, the correlator generates clock ticks at a frequency of `hz` per second. Be aware that there is no value in increasing `hz` above the rate at which your operating system can generate its own clock ticks internally. On UNIX and some Windows machines, this is 100 Hz and on other Windows machines it is 64 Hz.

When you start the correlator, you can specify the `-xclock` option to disable the correlator's internal clock and replace it with externally generated time events. See [“Externally generating events that keep time \(&TIME events\)” on page 183](#).

About timers and their trigger times

In an event expression, when you specify the `within`, `wait`, or `at` operator you are specifying a timer. Every timer has a trigger time. The trigger time is when you want the timer to fire.

- When you use the `within` operator, the trigger time is when the specified length of time elapses. If a `within` timer fires, the event listener fails. In the following event listener, the trigger time is 30 seconds after A becomes true.

```
on A -> B within(30.0) notifyUser();
```

If B becomes true within 30 seconds after the event listener detects an A, the trigger time is not reached, the timer does not fire, and the monitor calls the `notifyUser()` action. If B does not become true within 30 seconds after the event listener detects an A, the trigger time is reached, the timer fires, and the event listener fails. The monitor does not call `notifyUser()`. The correlator subsequently terminates the event listener since it can never trigger.

- When you use the `wait` operator, the trigger time is when the specified pause during processing of the event expression has elapsed. When a `wait` timer fires, processing continues. In the following expression, the trigger time is 20 seconds after A becomes true. When the trigger time is reached, the timer fires. The event listener then starts watching for B. When B is true, the monitor calls the success action.

```
on A -> wait(20.0) -> B success();
```

- When you use the `at` operator, the trigger time is one or more specific times. An `at` timer fires at the specified times. In the following expression, the trigger time is five minutes past each hour every day. This timer fires 24 times each day. When the timer fires, the monitor calls the `success` action.

```
on all at(5, *, *, *, *) success();
```

Important:

Triggering using the `at` operator always uses the time zone in which the correlator is running.

At each clock tick, the correlator evaluates each timer to determine whether that timer's trigger time has been reached. If a timer's trigger time has been reached, the correlator fires that timer. When a timer's trigger time is exactly at the same time as a clock tick, the timer fires at its exact trigger time. When a timer's trigger time is not exactly at the same time as a clock tick, the timer fires at the next clock tick. This means that if a timer's trigger time is .01 seconds after a clock tick, that timer does not fire until .09 seconds later.

When a timer fires, the current time is always the trigger time of the timer. This is regardless of whether the timer fired at its trigger time or at the first clock tick after its trigger time. In other words, the current time is equal to the value of the `currentTime` variable when the timer was started plus the elapsed wait time. For example:

```
float listenerSetupTime := currentTime;
on wait(1.23) {
    //When the timer fires, currentTime = (listenerSetupTime + 1.23)
}
```

A single clock tick can make a repeating timer fire multiple times. For example, if you specify `on all wait(0.01)`, this timer fires 10 times every tenth of a second.

Because of rounding constraints,

- A timer such as `on all wait(0.1)` drifts away from firing every tenth of a second. The drift is of the order of milliseconds per century, but you can notice the drift if you convert the value of the `currentTime` variable to a string.
- Two timers that you might expect to fire at the same instant might fire at different, though very close, times.

The rounding constraint is that you cannot accurately express 0.1 seconds as a float because you cannot represent it in binary notation. For example, the `on wait(0.1)` event listener waits for 0.10000000000000000555 seconds.

To specify a timer that fires exactly 10 times per second, calculate the length of time to wait by using a method that does not accumulate rounding errors. For example, calculate a whole part and a fractional part:

```
monitor TenTimesPerSecondMonitor {
    // Use integers to keep track of the next timer fire time.
    // This ensures that the value of the currentTime variable increases
    // by exactly 1.0 after every 10 tenths of a second.
    integer nextFireTimeInteger;
```

```
integer nextFireTimeFraction;
action onload() {
    nextFireTimeInteger := currentTime.ceil();
    nextFireTimeFraction := (10.0 *
        (currentTime-nextFireTimeInteger.toFloat() ) ).ceil();
    setupTimeListener();
}

action setupTimeListener() {
    nextFireTimeFraction := nextFireTimeFraction + 1;
    if(nextFireTimeFraction = 10) {
        nextFireTimeFraction := 0;
        nextFireTimeInteger := nextFireTimeInteger+1;
    }
    on wait( (nextFireTimeInteger.toFloat() +
        (nextFireTimeFraction.toFloat()/10.0) ) - currentTime )
    {
        setupTimeListener();
        doWork();
    }
}

action doWork()
{
    // This is called 10 times every second.
    log currentTime.toString();
    // ...
}
}
```

When a timer fires, the correlator processes items in the following order. The correlator:

1. Triggers all event listeners that trigger at the same time.
2. Routes any events, and routes any events that those events route, and so on.
3. Fires any timers at the next trigger time.

Disabling the correlator's internal clock

By default, the correlator keeps time by generating clock ticks every tenth of a second. If you specify the `-xclock` option when you start a correlator, the correlator disables its internal clock. This means the correlator does not generate clock ticks and does not assign timestamps based on clock ticks to incoming events.

Instead, it is up to you to send `&TIME` events into the correlator to externally keep time. This gives you the ability to artificially control how the correlator keeps time.

Time flows in all contexts, including private contexts. Also, different contexts can have different internal times. This happens when one context is still processing events that arrived at an earlier time while another is processing more recent events. The "currentTime" is always the time of the events being processed. (As opposed to wall-clock time, which can be obtained from the Time Manager EPL plug-in.)

Externally generating events that keep time (&TIME events)

A &TIME event can have one of the following formats:

- It can contain a number of seconds:

```
&TIME(float seconds)
```

The *seconds* parameter represents the number of seconds since the epoch, 1st January 1970.

The maximum value for *seconds* that the correlator can accept is 10^{12} , which equates to roughly 33658 AD, and should be enough for anyone. However, most time formatting libraries cannot produce a date for numbers anywhere near that large.

- Or it can contain a time string:

```
&TIME(string time)
```

The *time* is a string in extended ISO8601 form, with fractional seconds. For example:

```
&TIME("2015-04-20T23:32:41.032+01:00")
```

```
&TIME("2015-04-20T22:32:41.032+00:00")
```

```
&TIME("2015-04-20T22:32:41.032Z")
```

```
&TIME("2015-04-20T22:32:41.032")
```

These all refer to the same time. Note that the first example shows the time in a different timezone with an offset of 1 hour.

When the correlator processes an &TIME event by taking it off an input queue, the correlator sets its internal time (the current time) in that context to the value encoded in the event. Every event that the correlator processes after an &TIME event and before the next &TIME event has the same timestamp. That is, they have the timestamp indicated by the value of the previous &TIME event. For example:

```
&TIME(1)
A()
B()
&TIME(2)
C()
```

Events A and B each have a timestamp of 1. Event C has a timestamp of 2.

If you specify the `-xclock` option, and you do not send &TIME events to the correlator, it is as if time has stopped in the correlator. Every event receives the exact same timestamp. While not sending time events is not strictly incorrect, it does mean that time stands still.

You must use great care when implementing this facility. There are EPL operations that rely on correct time-keeping. For example, all timer operations rely on time progressing forwards. Timers will fail to fire if time remains at a standstill, or worse, moves backwards. There is a warning message in the correlator log if you send a time event that moves time backwards.

When sending &TIME events into a multi-context application, the time ticks are delivered directly to all contexts. This can be different than the way in which events in the `.evt` file are sent in to the

correlator and then sent between contexts in an application. This difference can result in processing events at an incorrect simulated time. In these cases, sending `&FLUSHING(1)`, for example, before sending time ticks and events can result in more predictable and reliable behavior.

For more information, see "Event timing" in the correlator utilities section of *Deploying and Managing Apama Applications*.

About repeating timers and `&TIME` events

You are not required to send `&TIME` events every tenth of a second. You can send them at larger intervals and timers will behave as they would when the correlator generates clock ticks. For a repeating timer, a single `&TIME` event can make it fire multiple times. Consequently, sending an `&TIME` event can have a lot of overhead if it is a large time jump and there are repeating timers. For example, consider the following pattern:

1. You start the correlator and specify the `-xclock` option, which sets the time to 0.
2. You inject a timer into the correlator, for example, on `all wait(0.1)`.
3. You send an `&TIME` event to the correlator and this event has a relatively large value, for example, 1185898806.

The result of this pattern is that the timer fires many times because the `&TIME` event causes each intermediate, repeating timer to fire. (Intermediate timers are timers that are set to fire between the last-received time and the next-received time.) For the example given, the timer fires 10^{10} times, which can take a while to process. You can avoid this problem by doing any one of the following:

- Send the correlator an `&TIME` event and specify a sensible time before you set up any timers. This is likely to be your best alternative.
- Send the correlator an `&TIME` event and specify a sensible time before you inject any monitors.
- Send the correlator an `&SETTIME` event before you send the `&TIME` event. See ["Setting the time in the correlator \(&SETTIME event\)" on page 184](#).

Setting the time in the correlator (`&SETTIME` event)

A `&SETTIME` event can have one of the following formats:

- It can contain a number of seconds:

```
&SETTIME(float seconds)
```

The *seconds* parameter represents the number of seconds since the epoch, 1st January 1970. For example:

`&SETTIME(0)` sets the time to "Thu Jan 1 00:00:00.0 BST 1970".

`&SETTIME(1185874846.3)` sets the time to "Tue Jul 31 09:40:46.3 BST 2007".

- Or it can contain a time string:

```
&SETTIME(string time)
```


The *time* is a string in extended ISO8601 form, with fractional seconds. For example:

```
&SETTIME("2015-04-20T23:32:41.032+01:00")
```

```
&SETTIME("2015-04-20T22:32:41.032+00:00")
```

```
&SETTIME("2015-04-20T22:32:41.032Z")
```

```
&SETTIME("2015-04-20T22:32:41.032")
```

These all refer to the same time. Note that the first example shows the time in a different timezone with an offset of 1 hour.

Normally, you do not need to send `&SETTIME` events. You would just send `&TIME` events. An `&SETTIME` event is useful only to avoid the problem pattern described above. The only difference between an `&SETTIME` event and an `&TIME` event is that the `&SETTIME` event causes an intermediate, repeating timer to fire only once while the `&TIME` event causes intermediate, repeating timers to fire repeatedly. For example, on `all wait(0.1)` fires ten times for every second in the difference between consecutive `&TIME` events. However, it fires only once when the correlator receives an `&SETTIME` event.

If you decide to send an `&SETTIME` event before an `&TIME` event, you typically want to send the `&SETTIME` event only before the first `&TIME` event. You should not send an `&SETTIME` event before subsequent `&TIME` events. Doing so causes a jumpy quality in the behavior of time. There is a warning message in the correlator log if you set a time that moves time backwards.

For information about when you might want to use external time events, see "Determining whether to disconnect slow receivers" in *Deploying and Managing Apama Applications*.

Out of band connection notifications

Apama applications running in the correlator can make use of Apama *out of band notifications*. Out of band notifications are events that are automatically sent to all public contexts in a correlator whenever any component (an IAF adapter, dashboard, another correlator, or a client built using the Apama SDKs) connects or disconnects from the correlator.

For example, consider an environment where correlator A and correlator B both have out of band notifications enabled and are connected so that events from correlator A are sent to correlator B. In this case, correlator A will receive a `ReceiverConnected` event and correlator B will receive a `SenderConnected` event. The Apama application running in correlator A and B can listen for those events and execute some application logic. Note that clients such as dashboards and IAF adapters typically connect as both receiver and a sender together and, therefore, two events would be sent in quick succession.

Out of band events are defined in the `com.apama.oob` package and consist of:

- `OutOfBandConnections`
- `ReceiverConnected`
- `SenderConnected`
- `ReceiverDisconnected`

■ SenderDisconnected

`OutOfBandConnections` contains helper functions to get currently connected senders and receivers synchronously. These functions return a sequence of `ReceiverConnected` events for connected receivers and sequence of `SenderConnected` events for connected senders. Your application can call these functions at any time and can consume the `ReceiverConnected` and `SenderConnected` events in the same way as it consumes asynchronous out of band events. This is particularly useful for getting information about connected senders and receivers which were already connected before the application was injected and whose `ReceiverConnected` and `SenderConnected` events were missed by the application. See the "API Reference for EPL (ApamaDoc)" for more information about the event and helper functions provided.

The `ReceiverConnected` and `SenderConnected` events contain the name of the component that is connecting. When correlators and IAF adapters send a notification event, the format of the string that contains the component name is as follows:

```
"name"
```

If no name is provided, however, the component name is as follows:

```
"name (on port port_number)"
```

The *name* is the name that was specified when the component was started. For correlators and IAF adapters, you can specify a name with the `--name` option when you start the component (see "Starting the correlator" in *Deploying and Managing Apama Applications* and "IAF command-line options" in *Connecting Apama Applications to External Components*). The name defaults to `correlator` or `iaf` according to the type of component. The *port_number* is the port that the connecting receiver or sender is running on.

Out of band events make it possible for developers of Apama components to add appropriate actions for the component to take when it receives notice that another component of interest has connected or disconnected. For example, an adapter can cancel outstanding orders or send a notification to an external system.

To enable out of band notifications in your Apama applications, you add the **Out of Band Event Notifications** bundle to your project. This bundle contains the event definitions and the monitor that enables the notifications. See "Adding bundles to projects" in *Using Apama with Software AG Designer* or "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications* for further information. In your Apama application, you have to create a listener for out of band events specific to the components in which you are interested.

Note:

You can also enable out of band notifications for a correlator with the `engine_management` tool and its `-r set00B` option. Be sure to inject the event definitions before running the tool with that option. For more information about using the `engine_management` tool, see "Shutting down and managing components" in *Deploying and Managing Apama Applications*.

5 Working with Streams and Stream Queries

■ Introduction to streams and stream networks	188
■ Defining streams	189
■ Using output from streams	190
■ Defining stream queries	193
■ Defining custom aggregate functions	221
■ Working with lots that contain multiple items	225
■ Stream network lifetime	229
■ Using dynamic expressions in stream queries	232
■ Troubleshooting and stream query coding guidelines	239

EPL lets you create two kinds of queries:

- Self-contained queries are processing elements that communicate with other self-contained queries, and with their environment, by receiving and sending events. Self-contained queries are designed to be multithreaded and to scale across machines. A self-contained query is sometimes referred to as an Apama query. This kind of query is defined in a `.qry` file, which cannot contain a monitor. See [“Defining Queries” on page 59](#).
- Stream queries operate on streams of items to generate more valuable streams that contain derived items. Stream queries are defined in monitors. The following topics provide information about stream queries.

In stream queries, derived items can be of any EPL type. You can use standard relational operations, such as filters, joins, aggregation, and projection, to generate items. For example, you can define a query that converts a stream of raw tick data into a stream of volume-weighted average price (VWAP) items.

Stream-based language elements allow operations that refine events to be expressed more clearly and concisely than when using procedural language constructs such as event listeners. In particular, applications that need to calculate one value based on multiple items from an input stream are simpler and more efficient when written with stream queries.

Apama provides sample code that uses streams and stream queries in the `samples\ep1` directory of your Apama installation directory.

Introduction to streams and stream networks

A stream query is part of a *stream network*. A stream network starts with one or more *stream source templates* (see [“Creating streams from event templates” on page 189](#)). A stream source template collects matching events received by the monitor instance and places them as items in a stream. Stream queries (see [“Defining stream queries” on page 193](#)) take existing streams (a stream created by a stream source template or by another stream query) and generate added-value streams that contain derived items. Finally, *stream listeners* (see [“Using output from streams” on page 190](#)) bring items out of the stream network and into procedural code. In a given stream network, upstream elements feed into downstream elements to generate derived items.

When a monitor instance receives an event that matches a stream source template, the correlator activates the stream network. The passage of time can also cause the correlator to activate a stream network. If, for example, a stream query operates on the items received within the last 5.0 seconds, then 5.0 seconds after an item arrives the correlator will again activate the stream network (see [“Adding window definitions to from and join clauses” on page 199](#)).

In a given stream network activation, not all stream queries and not all stream listeners necessarily receive items. Which queries and stream listeners receive items depends on the definitions of the stream queries and stream listeners. However, in a given stream network activation, the correlator passes items through all queries and stream listeners in the network that receive items. A query or stream listener that receives an item is considered to be activated. Only when processing of all activated queries and stream listeners is complete does the correlator process the next event on the context's input queue.

In a given stream network activation, various queries can produce multiple items on their output streams. The items in a particular stream during a particular stream network activation are called a *lot*. If a stream query or stream listener receives a lot that contains multiple items, it processes all items as part of a single stream network activation (see [“Working with lots that contain multiple items” on page 225](#), and [“Coassigning to sequences in stream listeners” on page 192](#)).

The items in a lot are always ordered, and the lots themselves are always ordered.

Defining streams

You can use a stream variable to reference a stream. A stream variable declaration has the following form:

```
stream<type> name
```

Replace *type* with the type of the items in the stream. This can be any Apama type.

Replace *name* with an identifier for the stream. For example:

```
stream<Tick> ticks;
```

A stream variable can be a field in an event. However, you cannot route, enqueue, or send an event that contains a stream variable field.

There are two ways to create a stream:

- From an event template. See [“Creating streams from event templates” on page 189](#).
- From the result of a stream query on some other stream. See [“Defining stream queries” on page 193](#).

To obtain a reference to an existing stream, you must assign from or clone another stream value.

An inert stream never generates any output. There are a number of ways to create an inert stream including, but not limited to, the following:

- Calling `new` on a stream type or a type that contains a stream
- Declaring a global variable of stream type, or a type that contains a stream
- Spawning a monitor instance that contains a stream value

Creating streams from event templates

A stream can be created from an event template using the `all` keyword, including `all any()` to listen for events of all types. This is referred to as a stream source template.

Example:

```
stream<Tick> ticks := all Tick(symbol="APMA");
```

This creates a stream that contains all subsequent `Tick` events that have the symbol `APMA`. You can use any single event template this way, however, you must specify the `all` keyword and you

cannot use operators such as `and` or `followed-by` to combine several event templates. See also [“Stream network lifetime” on page 229](#).

Terminating streams

If a stream goes out of scope, it continues to exist until the monitor instance terminates or the stream is explicitly terminated in some fashion. Streams are not garbage-collected. This means it is possible to leak streams, thereby consuming memory and potentially performing unnecessary computation, if you do not explicitly terminate streams.

To terminate a stream, call the `quit()` method on a stream variable that refers to the stream you want to terminate. For example:

```
stream<integer> foo := all A();  
...  
foo.quit();
```

This might also terminate connected streams. See [“Stream network lifetime” on page 229](#). It is also possible to terminate connected streams by quitting a stream listener.

Using output from streams

A stream listener passes output items from a stream to procedural code. You use a `from` statement to create a stream listener. The `from` statement has two forms.

The first form of the `from` statement creates a stream listener that takes items from an existing stream. For example:

```
from sA as a {  
    /* Code here executes whenever an item is available from sA. */  
}
```

The second form of the `from` statement contains a stream query definition, which creates a new stream query. The stream listener takes items from the output stream of the query. For example:

```
from a in sA select a as a {  
    /* Code here executes whenever the query produces output. */  
}
```

The syntax for the first form of the `from` statement is as follows:

```
[listener := ] from streamExpr coassignment block
```

Syntax Element	Description
<i>listener</i>	Optional. You can specify a <code>listener</code> variable to refer to the stream listener that the <code>from</code> statement creates. You can declare a new <code>listener</code> variable or use an existing <code>listener</code> variable.
<i>streamExpr</i>	Specifies any expression of type <code>stream</code> except a stream query. This can be, for example, a <code>stream</code> variable or a

Syntax Element	Description
	stream source template. If you want to specify a stream query, use the other form of the <code>from</code> statement.
<i>coassignment</i>	<p>You must coassign the stream output into a variable. You can either use the <code>as</code> operator to implicitly declare the variable in the scope of the following statement or the <code>:</code> assignment operator to coassign to a local or global variable of the same type as the stream output that has already been declared.</p> <p>For details about the characters you can specify, see “Identifiers” on page 687.</p> <p>The output from a stream is referred to as a <i>lot</i>. Like an auction lot, a stream output lot can contain one or more items. If the stream output is a lot that contains more than one item, the <code>from</code> statement coassigns each item, in turn, to the variable. See “Working with lots that contain multiple items” on page 225.</p> <p>A <code>from</code> statement cannot specify multiple coassignments.</p>
<i>block</i>	<p>Specifies a block of EPL statements, enclosed in braces. The <code>from</code> statement coassigns each stream output item to the specified variable and executes the block once for each output item.</p> <p>If the stream output is a lot that contains more than one item, and you want to execute the block just once for the lot rather than once for each item in the lot, coassign the result to a sequence. See “Coassigning to sequences in stream listeners” on page 192.</p>

The syntax for the second form of the `from` statement is as follows:

```
[listener:=] StreamQueryDefinition coassignment block
```

Syntax Element	Description
<i>listener</i>	Optional. You can specify a <code>listener</code> variable to refer to the stream listener that the <code>from</code> statement creates. You can declare a new <code>listener</code> variable or use an existing <code>listener</code> variable.
<i>StreamQueryDefinition</i>	Specifies a stream query. See “Defining stream queries” on page 193 .
<i>coassignment</i>	You must coassign the stream output into a variable. You can either use the <code>as</code> operator to implicitly declare

Syntax Element	Description
	<p>the variable in the scope of the following statement or the <code>:</code> assignment operator to coassign to a local or global variable of the same type as the stream output that has already been declared.</p> <p>For details about the characters you can specify, see “Identifiers” on page 687.</p> <p>If the query outputs lots that contain more than one item, the <code>from</code> statement coassigns each item in the lot, in turn, to the variable. See “Working with lots that contain multiple items” on page 225.</p> <p>A <code>from</code> statement cannot specify multiple coassignments.</p>
<i>block</i>	<p>Specifies a block of EPL statements, enclosed in braces. The <code>from</code> statement coassigns each stream output item to the specified variable and executes the block once for each output item.</p> <p>If the stream output is a lot that contains more than one item, and you want to execute the block just once for the lot rather than once for each item in the lot, coassign the result to a sequence. See “Coassigning to sequences in stream listeners” on page 192.</p>

Listener variables and streams

Like event listeners, you can assign a stream listener to a `listener` variable. A stream listener exists until one of the following happens:

- The monitor instance that contains the stream listener is terminated.
- The stream or streams the listener refers to are terminated.

If you do not want to wait for one of the above to occur, you can stop a stream listener by calling the `quit()` method on a `listener` variable that refers to it. Note that in many cases this will also terminate the stream that is feeding the stream listener. See [“Stream network lifetime” on page 229](#).

Coassigning to sequences in stream listeners

Unlike event listeners, a stream query might generate multiple items for each external or routed event. This is usually due to a batched window (a window that is updated after every p seconds or after every m items arrive) or to a join operation on two streams. In this case, the correlator executes a stream listener action multiple times, once for each generated item.

In a stream query definition, a window defines the set of items from the input stream that the query operates on. See [“Adding window definitions to from and join clauses” on page 199](#).

To execute the stream listener action only once, and coassign all generated items at once, specify a stream listener that coassigns to a sequence variable. The sequence must contain items of the same type as the stream. For example:

```
sequence<A> seqA;
from batchedEvents: seqA {
    /* seqA contains all events that arrive in this batch */
}
```

Defining stream queries

A stream query operates on one or two streams to transform their contents into a single output stream. A stream query definition declares an identifier for the items in the stream so that the item can be referred to by the operators in the stream query. Here is a simple stream query definition:

```
stream<integer> ints := from a in sA select a.i;
```

When the correlator executes a statement that contains a stream query definition, the correlator creates a new stream query. Each stream query has an output stream (the type of which might differ from that of the input stream).

A stream query definition is an expression that evaluates to a stream value. The value is a reference to the output stream of the generated query.

Following is an example of a simple stream query in a stream listener:

```
from a in sA select a.b as b {
    doSomethingWith(b);
}
```

The following table describes the user-defined parts of this stream listener. It is important to understand the distinctive role each one serves.

Syntax Element	Description
a	This is an identifier that represents the current item in the stream being queried. See “Specifying input streams in from clauses” on page 197 .
sA	This variable represents the stream being queried.
a.b	This expression describes what each query result looks like. In this example, the query produces outputs from the b field of the events in the stream.
b	This is the variable that you coassign the query results to so that the correlator can use the query result in the stream listener's code block.

Linking stream queries together

A stream query definition is an expression and its result is a stream. Consequently, with one exception described below, you can use a stream query definition anywhere that you can use a stream value. For example, you can assign the resulting value to a stream variable:

```
stream <float> values := from a in sA select a.value;
```

Alternatively, you can use a stream query definition as the return value from an action. For example:

```
action createPriceStream (stream<Tick> ticks) returns stream<float> {  
    return from t in ticks select t.price;  
}
```

Another option is to embed a stream query within another stream query. For example:

```
from p in (from t in ticks where t.price > threshold select t.price)  
within period  
select wavg(t.price,t.volume) as vwap {  
    processVwap(vwap);  
}
```

You can use stream variables to link stream queries together, as detailed in the next section.

The exception is that you cannot use a stream query immediately after the `from` keyword in the first form of the `from` statement. For example, the following is not a valid statement:

```
from from t in ticks select t.price as tickPrice {  
    print tickPrice.toString();  
}
```

Instead, use the second form of the `from` statement and specify a stream variable or a stream source template. The following example specifies a stream variable:

```
from t in ticks select t.price as tickPrice{  
    print tick.price.toString();  
}
```

For more information on the different forms of the `from` statement, see [“Using output from streams” on page 190](#).

Simple example of a stream network

Sometimes a single `from` statement is all that is required to achieve your goal. For example, to obtain a volume-weighted average price (VWAP) for a stock, you can add the following `from` statement to a monitor:

```
from t in all Tick(symbol="APMA")  
within period  
select wavg(t.price,t.volume) as vwap {  
    processNewVwap(vwap); }
```

Often, however, you want to use the output from one query as the input to another query. For example, here is an extract from the Statistical Arbitrage demo, which is available from the Welcome page:

```
spreads :=
  from a in all com.apama.demo.marketdata.Depth(symbol=order.Instrument_1)
    retain 1
  from b in all com.apama.demo.marketdata.Depth(symbol=order.Instrument_2)
    retain 1
  select (a.midPrices[0] - b.midPrices[0]);
stream<MeanSd> meanSds :=
  from s in spreads within 20.0 select MeanSd( mean(s), stddev(s) );
stream<integer> comparison :=
  from s in spreads from m in meanSds select
    compareSpreadAndBands(s, m.mean, m.sd, order.Std_Dev_Multiplier);
stream<integer> prevComparison :=
  from c in comparison retain 1 select rstream c;
from c in comparison from p in prevComparison
  where c!=p select c as instruction {
  if state = WAIT_FOR_SPREAD and instruction = HOLD {
    monitorState();
  }
  if state = MONITOR and instruction != HOLD {
    waitForOrders(instruction);
  }
}
```

When queries are connected like this, the set of connected queries is referred to as a stream network.

A stream network is strictly within a monitor instance. Routing an event takes that event entirely out of the stream network since the event would not be received in the same network activation even if it is received by the same monitor. Spawning a monitor makes any stream variables point to inert streams, so it is not possible to refer to a stream network from a different monitor instance.

Stream query definition syntax

A stream query definition contains several elements, some of which are optional and some of which are required. These elements, and their constituent parts, are described in the following sections. The elements appear in a stream query in this order:

```
FromClause [ FromClause | JoinClause ] [ WhereClause ] ProjectionDefinition
```

Element	Required or Optional	Description
<i>FromClause</i>	Required	Specifies the input stream for the query. See “Specifying input streams in from clauses” on page 197. A from clause can also specify which items from the input stream the query should operate on. See “Adding window definitions to from and join clauses” on page 199.

Element	Required or Optional	Description
		If a second <code>from</code> clause appears, the correlator performs a cross-join to combine items from the two streams. See “Defining cross-joins with two from clauses” on page 211.
<i>JoinClause</i>	Optional	Specifies a second stream for the query to operate on. The correlator performs an equi-join to combine items from the two streams. See “Defining equi-joins with the join clause” on page 213. A join clause can also specify which items from the input stream the query should operate on. See “Adding window definitions to from and join clauses” on page 199.
<i>WhereClause</i>	Optional	Applies a filtering criterion to the items in the window or the items produced by the join operation. See “Filtering items before projection” on page 215.
<i>ProjectionDefinition</i>	Required	Defines how the query generates output items. See “Generating query results” on page 216.

Identifier scope in stream queries

Consider the following code fragment:

```
integer a;  
stream<float> prices := from a in ticks select a.price;
```

In this example, the `a` in the query refers to the current `Tick` item in the stream and not to the `a` integer variable. In a stream query, you can use an identifier that you have not previously declared. If there is a variable in a containing scope that has the same name as an identifier in the query, then for expressions in the query the identifier in the query hides the variable in the containing scope.

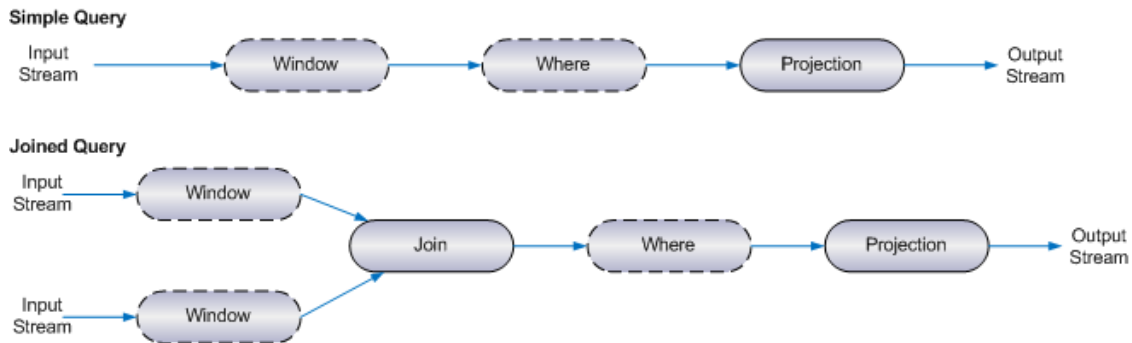
Following is another example of how scope works with stream queries:

```
integer a := 42;  
from a in ticks select a.price as p {  
    print a.toString(); // Prints "42" rather than one of the ticks. }  
}
```

The previous code fragment illustrates that identifiers in the listener action can have the same names as identifiers in the stream query. While this is not good practice, it is important to recognize that the listener action is not part of the stream query. Consequently, an identifier in a stream query is out-of-scope in the stream query's listener action.

Stream query processing flow

Each element of the stream query operates on the output of the previous part. To correctly define stream queries, it can be helpful to understand that items flow through the query and the correlator processes the parts of the query in the order shown in the following figure. In the figure, the dashed outlines indicate optional elements.



As items arrive on the input stream(s) and time elapses, the window definition for each stream identifies which items from that stream the query should be processing at any given moment. This includes partitioning, if it is specified. See [“Adding window definitions to from and join clauses” on page 199](#).

In queries with two input streams, the correlator combines items from the two streams by means of a cross-join operation (a second `from` clause) or an equi-join operation (a `join` clause). See [“Joining two streams” on page 211](#).

The `where` clause, if there is one, filters items. See [“Filtering items before projection” on page 215](#).

The projection definition defines how the query generates output items. This includes the `select` clause, which has appeared in examples such as [“Simple example of a stream network” on page 194](#). See [“Generating query results” on page 216](#).

Specifying input streams in from clauses

In a stream query, each `from` clause specifies a stream that the query is operating on. The syntax of the `from` clause is as follows:

```
from itemIdentifier in streamExpr [WindowDefinition]
```

Syntax Element	Description
<i>itemIdentifier</i>	Specify an identifier that you want to use to represent the current item in the stream you are querying. You use this identifier in subsequent clauses in the query. For details about the characters you can specify, see “Identifiers” on page 687 .

Syntax Element	Description
	<p>The type of the identifier is the same as the type of the items that are in the stream you are querying.</p> <p>There is no link between an item identifier in a query and a variable that you might define elsewhere in your code. In other words, it is okay for an in-scope variable to have the same name as an item identifier in a query. Inside the query, the item identifier hides that variable. See the second example below.</p>
<i>streamExpr</i>	Specify an expression that returns a <code>stream</code> type. This is the stream that you want to query.
<i>WindowDefinition</i>	Define which portion of the stream to query. See “Adding window definitions to from and join clauses” on page 199 .

Examples

The query below generates a stream of `float` items. The item identifier is `a`. The stream variable, `ticks`, refers to a stream of `Tick` events. The `select` clause specifies that each query result item contains only the price value from the `Tick` event. Details about the `select` clause are in [“Generating query results” on page 216](#).

```
stream<float> prices := from a in ticks select a.price;
```

The `all` keyword followed by an event template is an expression of type `stream` referred to as a stream source template. Consequently, you can use this in a `from` clause. For example, you can modify the previous example to use the stream source template directly within the stream query:

```
stream<float> prices :=  
  from a in all Tick(symbol="APMA") select a.price;
```

Notes

A stream query is an expression of type `stream` and so anywhere that you can specify a stream expression you can use a stream query in its place. (There is one exception to this. See [“Linking stream queries together” on page 194](#).) This means you can nest stream queries to create a compound stream query. For example, consider the following non-nested stream queries:

```
stream<A> sA := all A();  
  
stream<integer> derived :=  
  from a in sA retain 2 select mean(a.x);  
  
stream<B> sB :=  
  from a in derived within 10.0 select B(stddev(a));
```

An equivalent way to write this is as follows:

```
stream<B> sB :=
```

```

from b in
  from a in all A() retain 2 select mean(a.x)
within 10.0
select B(stddev(b));

```

The compiler generates the same stream network in both cases, so the performance is exactly the same. However, nesting stream queries beyond one level can make the compound stream query hard to understand.

To define a query that operates on two streams, specify two consecutive `from` clauses or specify a `from` clause followed by a `join` clause. See [“Joining two streams” on page 211](#).

Adding window definitions to `from` and `join` clauses

The items flowing through a stream are ordered. In any given activation, there are zero or more items that are current. By default, the stream query operates on those current items.

Alternatively, a window may be defined. Window definitions specify which items the query should operate on in each activation, based on (but not limited to) the following:

- The items within a given time period
- A maximum number of items
- The content of the items

As the window contents change, the items in the query projection will also change: new items will be inserted and old ones removed. The output from a query is a stream of items.

If the projection is an aggregate projection then the query output is the result of evaluation of the `select` clause when the window contents change. See [“Aggregating items in projections” on page 218](#).

If the projection is a simple, non-aggregate projection, the default output is the insert stream (or *istream* for short) of new projected items. Alternatively, if the `rstream` keyword is specified in the `select` clause, the output is the remove stream (or *rstream*) of items that have become obsolete. See also [“Obtaining the query's remove stream \(rstream\)” on page 219](#).

Window definition syntax

There are a number of different formats and keywords that you can use to define a window on a stream. Following are the alternatives you can choose from. See the subsequent topics for details.

```

[partition by partitionByExpr[, partitionByExpr]...]

(
  within windowDurationExpr[every batchPeriodExpr]
    [retain windowSizeExpr] [with unique keyExpr]
  | retain windowSizeExpr [every batchSizeExpr] [with unique keyExpr]
)

| retain all

```

Every window definition specifies `retain`, `within` or `both`.

Syntax Element	Description
<i>partitionByExpr</i>	Optionally specifies an EPL expression that should involve the input item in some way and that returns a comparable type. A <code>partition by</code> clause effectively creates a separate window for each encountered distinct value of <i>partitionByExpr</i> .
<i>windowDurationExpr</i>	Specifies a float expression that indicates a duration of a number of seconds. The window contains the items received within the last <i>windowDurationExpr</i> seconds. See “Defining time-based windows” on page 201 .
<i>batchPeriodExpr</i>	Specifies a float expression that indicates an interval period of a number of seconds. The window updates its contents every <i>batchPeriodExpr</i> seconds. See “Defining batched windows” on page 205 .
<i>windowSizeExpr</i>	Specifies an integer expression that indicates the number of items you want to retain in the window. The window contains the most recent <i>windowSizeExpr</i> items. See “Defining size-based windows” on page 202 .
<i>keyExpr</i>	<p>Specifies an EPL expression that must contain at least one reference to the input item and must return a comparable type. See “Comparable types” on page 593.</p> <p>If you add a <code>with unique</code> clause, if there is more than one item in the window that has the same value for the key identified by <i>keyExpr</i>, only the most recently received item is considered to be in the window. See “Defining content-dependent windows” on page 209.</p>
<i>batchSizeExpr</i>	Specifies an integer expression that indicates a number of items. The window updates its contents after every <i>batchSizeExpr</i> items that match the query are found. See “Defining batched windows” on page 205 .

Omitting the window definition

The window definition is optional in a stream query. If you do not specify any window then, for any given activation of the stream query, the stream query operates on only the items that are current for that activation. Typically, this is a single event. However, if the source for this query is, for example, a stream query with a batched window, then the items in each batch will be processed together as in the following example:

```
stream<A> sA := from a in all A() retain 4 every 4 select a;
from a in sA select count() as c { ... }
```


The second query receives batches of four A events and will generate a single aggregate value for each batch. For more details see [“Stream queries that generate lots” on page 226](#).

Retaining all items

The simplest window is one that contains all items that have ever been in the stream. The corresponding window definition is `retain all`. Conceptually, once an item enters a `retain all` window, it remains in the window indefinitely (or until the stream query is terminated). The following query evaluates the running mean of all items that have ever been in the `values` stream:

```
stream <decimal> means := from v in values retain all select mean(v);
```

The `retain all` clause specifies an unbounded window. Unbounded windows have restrictions on their use:

- You cannot have a partitioned or batched unbounded window.
- You cannot perform a join operation on an unbounded window.
- You cannot specify an unbounded window when you use `rstream` in the `select` clause of a query.

When you use a custom (user-defined) aggregate function in a query that contains an unbounded window, you cannot also use a bounded aggregate function. You should also be aware that, if you use a badly implemented custom aggregate function in a query that contains an unbounded window, then this can result in uncontrolled memory usage. See [“Defining custom aggregate functions” on page 221](#).

Defining time-based windows

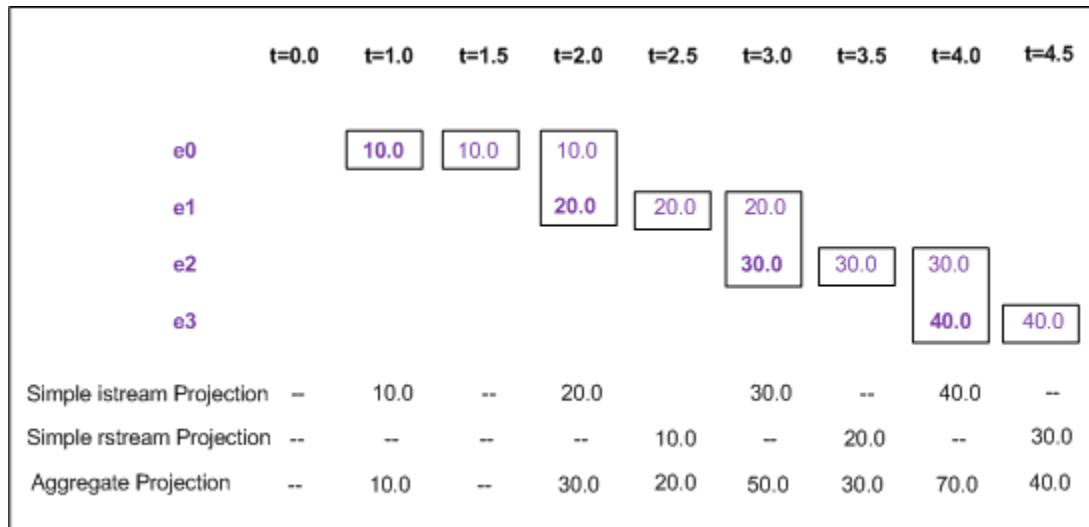
In a time-based window, the items are held in the window for a specific duration. The syntax for defining a time-based window is:

```
within windowDurationExpr
```

Replace `windowDurationExpr` with an expression that returns the number of seconds that items should remain in the window as a `float` value. For example, the following query calculates the sum of all items that arrived in a stream of `float` values during the last 1.5 seconds:

```
stream<float> sums := from v in values within 1.5 select sum(v);
```

The following diagram illustrates how this works in practice.



Each column represents a time when the query window contents change whereas each row represents the arrival and lifetime of each event. As an event arrives in the window, it appears in bold purple. At each given time, the current window contents is indicated by the items enclosed by boxes. Bold purple items are new and lighter purple items are old items still in the window. The numbers at the bottom give the contents of the stream of insertions to and removals from the window in the case where each value is being selected independently, or when the aggregate sum of the values in the set of items in the window is being calculated. The query before the diagram corresponds to the aggregate projection line. The queries shown here are:

- Simple istream projection:

```
from v in values within 1.5 select v
```

- Simple rstream projection:

```
from v in values within 1.5 select rstream v
```

- Aggregate projection:

```
from v in values within 1.5 select sum(v)
```

In a simple, non-aggregate projection, when an event arrives in the window, it appears in the istream of the projection. It remains for 1.5 seconds, at which point it appears on the rstream of the projection. The aggregate projection behaves differently. Whenever an item arrives in or is removed from the window, a new sum appears on the istream of the aggregate projection.

Defining size-based windows

As well as time, you can specify windows that contain only a certain number of items. In a size-based window, as each new item arrives, it is added to the window. After the number of items in the window reaches the window size limit specified in the query, the arrival of a new item causes the removal of the oldest item from the window.

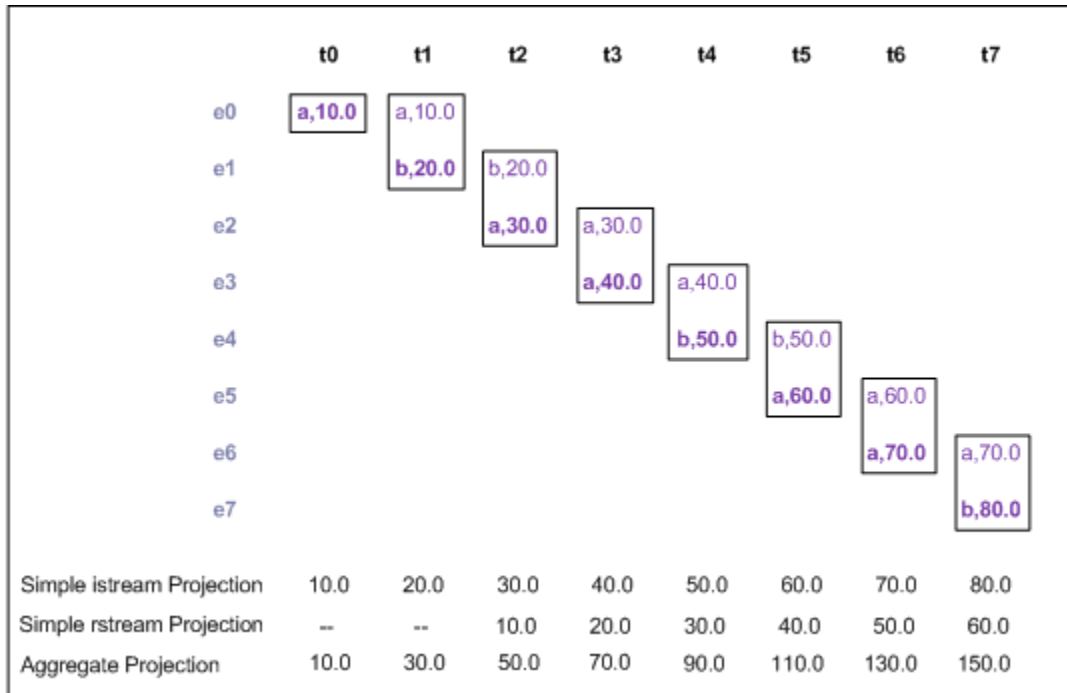
The syntax for defining a size-based window is as follows:

```
retain windowSizeExpr
```

Replace *windowSizeExpr* with an expression that returns how many items you want to retain in the window as an integer value. For example, the following query calculates the sum of the last two items in a stream of floats:

```
stream <float> sums := from v in values retain 2 select sum(v.number);
```

The following diagram, which uses the same notation as the previous section, illustrates how this works in practice.



The query before the diagram corresponds to the aggregate projection. The three queries shown here are:

- Simple istream projection:

```
from v in values retain 2 select v.number
```

- Simple rstream projection:

```
from v in values retain 2 select rstream v.number
```

- Aggregate projection:

```
from v in values retain 2 select sum(v.number)
```

When an event arrives in the window, it appears in the istream of a simple, non-aggregate projection. The first item remains in the window when a second item arrives. When a third item arrives, the first item is no longer in the window and it appears on the rstream of the simple, non-aggregate projection. Likewise, when the fourth item arrives in the window, it appears in the istream and the second item appears on the rstream of the simple projection, and so on. The behavior of the aggregate projection is that whenever an item arrives in or is removed from the window, a new sum appears on the istream of the aggregate projection.

Combining time-based and size-based windows

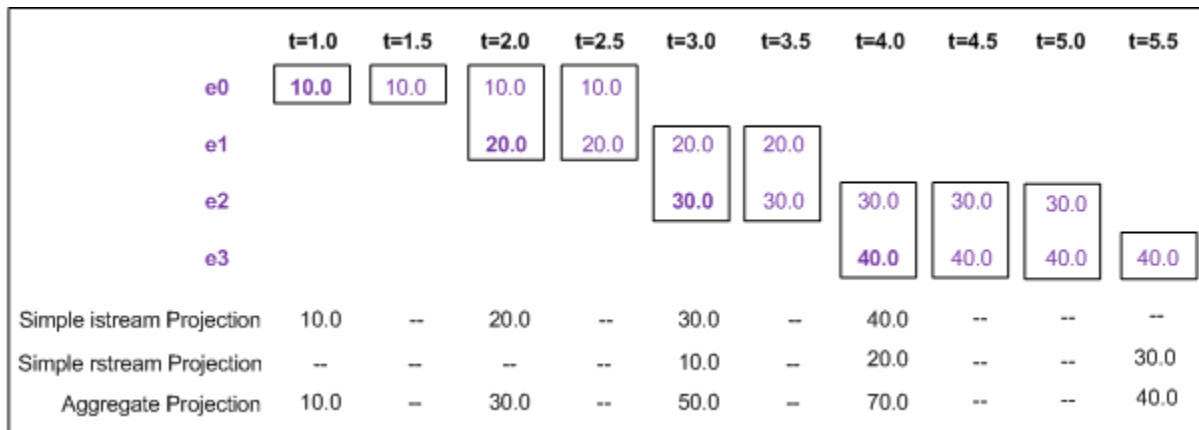
Sometimes you might want to focus on the last n items received in the last d seconds. To define a window that retains items based on both time and size, use the following format in the `from` clause:

```
within windowDurationExpr retain windowSizeExpr
```

The `within` keyword and expression must be first and the `retain` keyword and expression must be second. As with separate size-based and time-based windows, replace `windowDurationExpr` with an expression that returns a number of seconds, d , as a `float` value. Replace `windowSizeExpr` with an expression that indicates how many items you want to retain in the window, n , as an integer value. The window contains the last n items received in the last d seconds. If no items were received in the last d seconds, the window is empty. For example:

```
from v in values within 2.5 retain 2 select sum(v);
```

The following diagram, which uses the same notation as the previous section, illustrates how this works in practice.



The query before the diagram corresponds to the aggregate projection. The three queries shown here are:

- Simple istream projection:

```
from v in values within 2.5 retain 2 select v
```

- Simple rstream projection:

```
from v in values within 2.5 retain 2 select rstream v
```

- Aggregate projection:

```
from v in values within 2.5 retain 2 select sum(v);
```

The important point to note in this example is that some items drop out of the window before the 2.5 second period is passed. When `e2` arrives, `e0` and `e1` are already in the window. Even though `e0` has been there for only 2 seconds, it is removed because `e1` and `e2` are now the two most recent items received in the last 2.5 seconds.

Defining batched windows

The default behavior is that the contents of a window change upon the arrival of each item. The `every` keyword can be used to control when the contents of the window change: it causes the items to be added to the window in batches. Time-based windows can be controlled to update only every p seconds, and size-based windows can be controlled to update only after every m events.

The syntax for a batched window is one of the following:

```
within windowDurationExpr every batchPeriodExpr
| retain windowSizeExpr every batchSizeExpr
| within windowDurationExpr every batchPeriodExpr retain windowSizeExpr
```

Here, `windowDurationExpr` and `windowSizeExpr` retain their meaning from the previous sections. The `batchPeriodExpr` is an expression that returns the time, p , between updates as a float value. The `batchSizeExpr` is an expression that returns the number of events between updates, m , as an integer value.

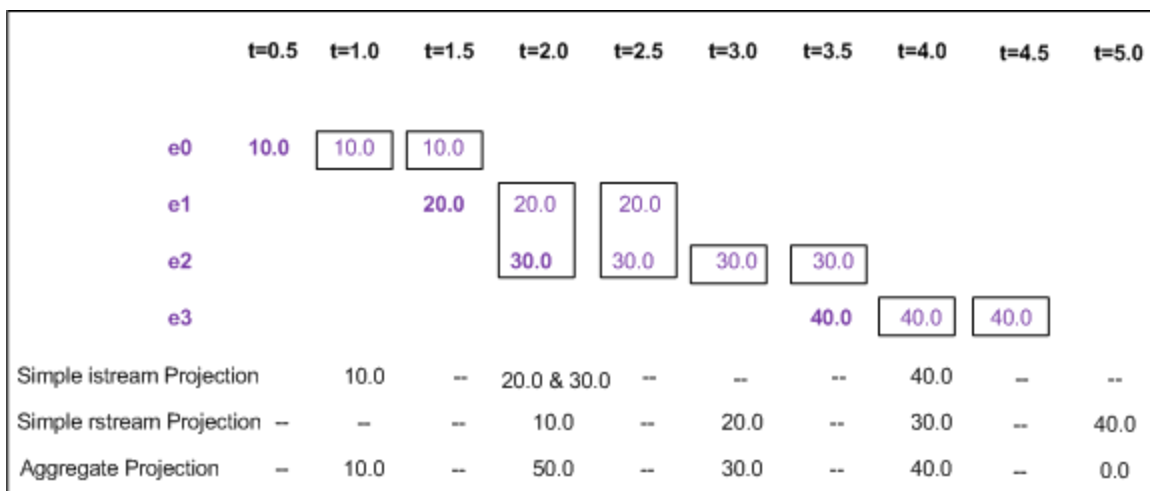
When you specify `within` followed by `every` followed by `retain`, the `every` keyword always indicates a number of seconds. That is, the window updates its content every p seconds.

If no items have arrived or expired since the previous window update, the window content is unchanged and consequently the query does not execute. The correlator executes the query only when the window content changes.

Here is an example of a stream query that defines a batched, time-based window. The correlator creates the query at $t=0.0$.

```
from v in values within 1.5 every 1.0 select sum(v)
```

The following diagram illustrates how this works in practice.



The query before the diagram corresponds to the aggregate projection. The three queries shown here are:

■ Simple istream projection:

```
from v in values within 1.5 every 1.0 select v
```

- Simple rstream projection:

```
from v in values within 1.5 every 1.0 select rstream v
```

- Aggregate projection:

```
from v in values within 1.5 every 1.0 select sum(v)
```

The important thing to note about the behavior of these queries is that the window content changes only every second. Nothing appears on any insert or remove stream between those points. This means that the items 10.0, 20.0 and 40.0 are not in the window at the moment they arrive, but are kept until the next multiple of 1.0 second. Item lifetimes are calculated from the item arrival time, not the point at which the batching allows the item into the window. Consequently, the lifetime of the items in the window is also affected by the batching. In these examples, you can see that the items that were delayed entering the window are only in the window for one second because they were already 0.5 seconds old at the point they entered the window. For contrast, the item with the value 30.0 remains in the window for 2.0 seconds because after 1.5 seconds the batching has not occurred, and so the window cannot change until the next multiple of 1.0 second.

In the examples given here, the batch period is smaller than the duration of the window. If the batch period is larger than the duration of the window, then some items can never enter the window, if they would have already expired by the time the next batch arrives in the window.

Batched size-based windows behave similarly to batched time-based windows, except that the batch criteria is waiting for a number of items to arrive. In that case, items always arrive in the window as a multiple of the batch size.

Batched windows produce multiple items at one time. A single group of items flowing between queries together is called a lot. A lot can contain one item or several items. A batched window is one way of producing a lot that contains several items.

Partitioning streams

The `partition by` clause splits a stream into partitions, based on one or more key values. The subsequent window operators are applied to the partitioned stream; the behavior is as if the window operators had been applied separately to each partition. The result of using `partition by` followed by a window operator is referred to as a partitioned window. You use a query with a partitioned window to retain particular items for each partition specified by the `partition by` clause.

Partitioning is introduced with the following syntax:

```
partition by partitionByExpr [, partitionByExpr] ...
```

The `partition by` clause precedes other window operators, so a complete query would be:

```
from a in sA partition by a.x retain 2 select sum(a.y);
```

Each *partitionByExpr* is an expression that should contain at least one reference to the input item and must return a comparable type. See [“Comparable types” on page 593](#). Some examples are in the following table. Assume that each `partition by` clause in the table starts with the following:

```
from a in all A() ...
```

Definition	Description
<code>partition by a.x</code>	Partition on a single primitive type field of the input event. This is likely to be the most common case.
<code>partition by a</code>	<p>Partition on an event's field values. The events that have identical values for all fields are in the same partition. For example:</p> <pre>from a in all A() partition by a retain 2 select a;</pre> <p>Given the following input events:</p> <pre>A(1,1) A(1,2) A(1,1)</pre> <p>The first and third events are in the same partition, the second is not. In this case, the event type <code>A</code> must itself be a comparable type.</p>
<code>partition by 1</code>	This is a valid partition expression, but it is not recommended. A partition expression should reference the input item in some way.
<code>partition by f(a)</code>	This is a valid partition expression if <code>f()</code> is a function that returns an appropriate type.
<code>partition by a.x*globaldict[a.y]</code>	Another valid partition expression.

Example:

```
from t in all Tick()
  partition by t.symbol retain 1
  select rstream t;
```

This query creates a separate partition for each new stock symbol it finds. Each partition contains the most recent `Tick` event for that symbol. The query output, for each encountered symbol, is the previous `Tick` event for that symbol. Note that it is possible for this query to consume a large quantity of memory.

Partitions and aggregate functions

The `partition by` clause creates several partitions within the window. However, a stream query has other parts in addition to the window. The other parts include the projection and optional join or where elements. These other parts of the query operate on a single window that contains all items from all partitions.

Likewise, when you partition a stream any specified aggregate functions aggregate over all partitions. If you want to generate separate aggregate values for different groups of events then you must specify a `group by` clause. See [“Grouping output items” on page 218](#). A common use case is to specify matching `partition by` and `group by` clauses.

Consider the following stream query:

```
from a in all A() partition by a.x retain 2 select sum(a.y);
```

The window definition is `retain 2`, and this is partitioned by `a.x`, where `x` is the first field in `A`. There is one `retain 2` partition for each value of `x`. Suppose this stream query receives the following input events:

```
A(1,1)
A(1,2)
A(2,1)
A(2,2)
A(1,3)
A(2,3)
```

After these events have all arrived, one partition contains `A(1,2)` and `A(1,3)` while a second partition contains `A(2,2)` and `A(2,3)`. However, the parts of the query following the window definition operate on the collection of all items in all partitions. In this example, the `sum()` aggregate function generates 10. It does not generate a lot that contains two values of 5. Now consider the following query:

```
from t in all Tick()
  partition by t.symbol retain 10
  group by t.symbol
  select mean(t.price)
```

This query returns one mean value per symbol, which is the mean of the last 10 ticks for that symbol. If you do not want all means for all symbols in one lot, you might prefer to spawn monitors so that you have an instance of the following query for each symbol:

```
from t in all Tick(symbol=X)
  retain 10
  select mean(t.price)
```

If you do want the averages for all the symbols in the same stream, then you can specify the group key in the select clause in order to later differentiate between the output events, as in the following example:

```
from t in all Tick()
  partition by t.symbol retain 10
  group by t.symbol
  select Output(t.symbol, mean(t.price))
```

As you can see, the `partition by` clause is often used in conjunction with the `group by` clause.

Tip:

In EPL, it is common to use `spawn` in a monitor to create separate monitor instances. For example, each monitor instance might process a separate stock symbol. Spawning separate monitor instances might be preferable to using a single monitor instance that specifies `partition by` in a stream query so that it, for example, processes all stock symbols. Spawning separate monitor instances can be more efficient because your application processes only the subset of symbols that are of interest. Also, the subset of symbols of interest can change through the day. Appropriate monitor instances and queries can be created as required.

See also [“IEEE special values in stream query expressions” on page 221](#).

Using multiple partition by expressions

To partition a window according to multiple criteria, you can insert multiple, comma-separated expressions. For example, you can refine a previous query to produce values for different volume bands as follows:

```
from t in all Tick()
  partition by t.symbol, t.volume.floor()/100 retain 1
  select rstream t;
```

In this example, the correlator applies `retain 1` to each set of ticks that share both the same symbol and the same volume (to within 100). As a result, an item is output only when a replacement tick arrives for an existing symbol in an existing volume band.

Partitioning time-based windows

If a window is purely time-based, then there is no benefit to partitioning the window. For example, consider the following two queries:

```
from t in all Tick() within 1.0 ...
from t in all Tick() partition by t.symbol within 1.0 ...
```

The first query outputs every `Tick` received in the last second. The second query organizes the stream of `Tick` events by their symbols, then gives you each one that arrived in the last second. This is still every `Tick` received in the last second. The correlator ignores a `partition by` statement if it is used only with a `within` window.

If your window includes a `retain` clause as well as a `within` clause then it can be helpful to use `partition by`, likewise if there is a `with` clause. See [“Defining content-dependent windows” on page 209](#). For example:

```
from t in all Tick() partition by t.symbol within 10.0 retain 5 ...
```

This window will contain at most 5 `Tick` events for each different symbol received within the last 10 seconds.

Defining content-dependent windows

The contents of the window can also depend on the content of individual items in the stream. Currently, the only content-dependent window operator is the `with unique` clause, which limits the window to containing only the most recent item for each key value. The `with unique` clause can be added to a `within` or a `retain` window by following it with:

```
with unique keyExpr
```

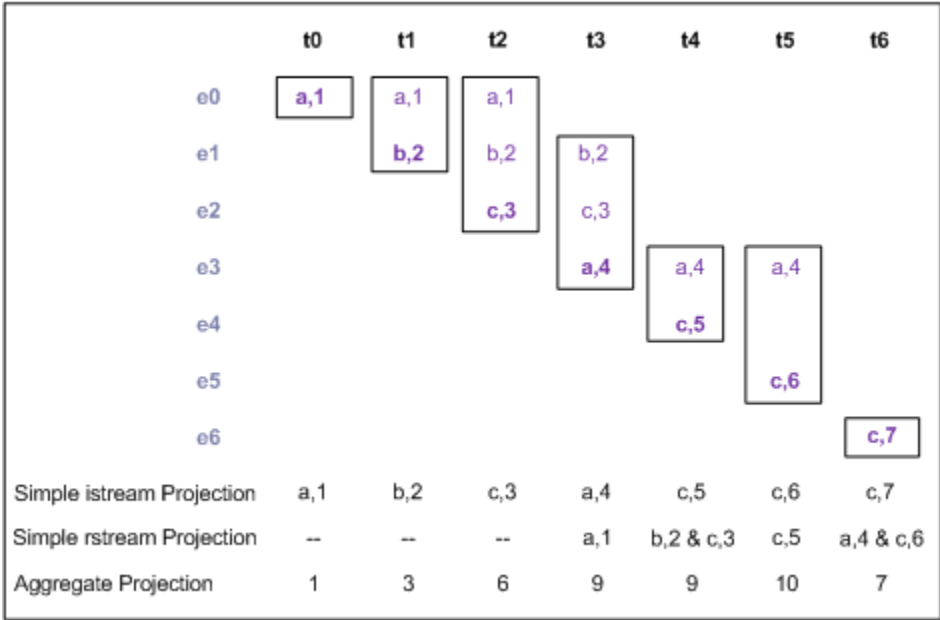
The *keyExpr* follows the same rules as a partition key expression. That is, it is an expression that should contain at least one reference to the input item and must return a comparable type. See [“Comparable types” on page 593](#).

If you add a `with unique` clause, if there is more than one item in the window that has the same value for the key identified by *keyExpr*, only the most recently received item is considered to be

in the window. It is important to note that the `with unique` clause processing happens after the rest of the window processing. Consider the following query:

```
from p in pairs retain 3 with unique p.letter select sum(p.number)
```

If the most recent two events have the same letter, there will be only two events over which the sum is calculated. This is illustrated in the following diagram:



The query before the diagram corresponds to the aggregate projection. The three queries shown here are:

- Simple istream projection:

```
from p in pairs retain 3 with unique p.letter select p
```
- Simple rstream projection:

```
from p in pairs retain 3 with unique p.letter select rstream p
```
- Aggregate projection:

```
from p in pairs retain 3 with unique p.letter select sum(p.number)
```

As you can see, when the last three items received all have a unique letter, the query behaves like a `retain 3` window. When the last three items received do not all have a unique letter, the duplicate that arrived first is removed from the window. In this example, the arrival of `c,5` causes the removal of `c,3` even though it was one of the last three items received. In other words, the `with unique` clause can cause an item to be removed from the window and the sum earlier than it would otherwise be removed.

The difference between a partitioned window and a window that is using a `with unique` clause can be described as:

- using `partition by` gives you the last three values for each key, and

- using `with unique` gives you one value of each key, from the last three.

You can combine both `partition by` and `with unique` if you are using different key expressions in each clause.

Note that you cannot specify `within` followed by `retain` followed by `with unique`.

See also “[IEEE special values in stream query expressions](#)” on page 221.

Joining two streams

When a stream query operates over two input streams, it is referred to as a join operation. There are two forms of join operation available in EPL:

- A cross-join joins every event from one stream's window with every event in the other stream's window.
- An equi-join joins events only when they have matching keys.

Each form takes two input streams and produces a single output stream of combined items.

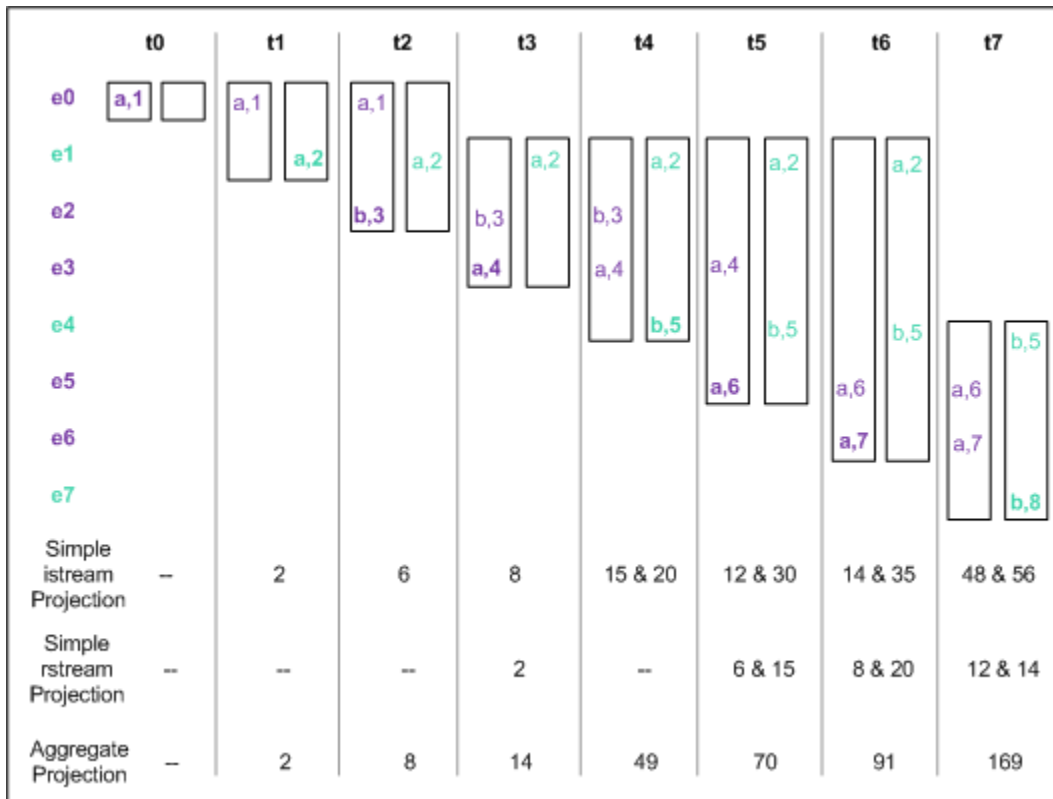
Join operations, particularly cross-joins, can create many more output events than input events, not just the same or fewer.

Defining cross-joins with two from clauses

A cross-join is defined with two `from` clauses, one for each stream, optionally including window definitions. A simple example of this is:

```
from p1 in leftPairs retain 2
  from p2 in rightPairs retain 2
  select sum(p1.num * p2.num);
```

This is illustrated in the following diagram, whose notation differs from the previous diagrams. Here, for each time point there are two columns, one for each side of the join. The first column, with purple events, represents the items from the first `from` clause and the second column, with cyan events represents the items from the second `from` clause. Events in bold arrived during this activation of the stream query and the boxes enclose the windows for each side. As in the previous diagrams, the output is given for each of the three kinds of projections.



The query before the diagram corresponds to the aggregate projection. The three queries shown here are:

■ Simple istream projection:

```
from p1 in leftPairs retain 2
  from p2 in rightPairs retain 2
  select p1.num * p2.num
```

■ Simple rstream projection:

```
from p1 in leftPairs retain 2
  from p2 in rightPairs retain 2
  select rstream p1.num * p2.num
```

■ Aggregate projection:

```
from p1 in leftPairs retain 2
  from p2 in rightPairs retain 2
  select sum(p1.num * p2.num);
```

As shown in the diagram, in a cross-join whenever an item arrives in a window, it is joined to every item in the other window to produce a separate output item for each combination.

Because the number of output items is the product of the size of the two windows, cross-joins are normally used for joins between at least one of:

- A window of size 1.
- A stream where you have omitted the window definition.

If both sides of the join omit the window definition, then for output to occur an item must arrive on each stream during the same activation of the query.

A more concrete example can be seen in the Statistical Arbitrage demo (available from the Welcome page), which includes the following statement:

```
spreads :=
  from a in all com.apama.demo.marketdata.Depth(symbol=order.Instrument_1)
    retain 1
  from b in all com.apama.demo.marketdata.Depth(symbol=order.Instrument_2)
    retain 1
  select (a.midPrices[0] - b.midPrices[0]);
```

This query generates the spread between the latest prices for the two identified stocks. In each from clause, the window contains one item. Whenever a new item arrives in one window, the query executes the calculation defined in the select clause and outputs the result.

To generate a running mean and a standard deviation for this spread value, you can define the following query:

```
stream<MeanSD> averages := from s in spreads within 20.0
  select MeansSD(mean(s),stddev(s));
```

Then, to obtain all three current values for the spread, the mean and the standard deviation, you can perform a join between the spreads stream and the averages stream:

```
stream<SpreadMeanSD> all := from s in spreads
  from a in averages
  select SpreadMeansSD(s, a.mean, a.stddev);
```

This query outputs a result only when there is an item currently in both spreads and averages.

In a cross-join, you cannot specify more than two from clauses.

CAUTION:

Be aware that cross-joins have the potential to generate a great quantity of output. It is preferable to use cross-joins only where the window size/duration of any window involved in the cross-join is small. For example, putting 8000 events through a 100x100 cross-join produces 1.6 million output events. You cannot specify a cross-join in a query that contains an unbounded window.

Defining equi-joins with the join clause

An equi-join has a key expression for each of the two streams that are being joined. Two items are joined into an output item only if the values of their key expressions are equal. The full syntax for an equi-join, consisting of a from clause followed by a join clause, is:

```
from itemIdentifier1 in streamExpr1 [windowDefinition1]
  join itemIdentifier2 in streamExpr2 [windowDefinition2]
  on joinKeyExpr1 equals joinKeyExpr2
```

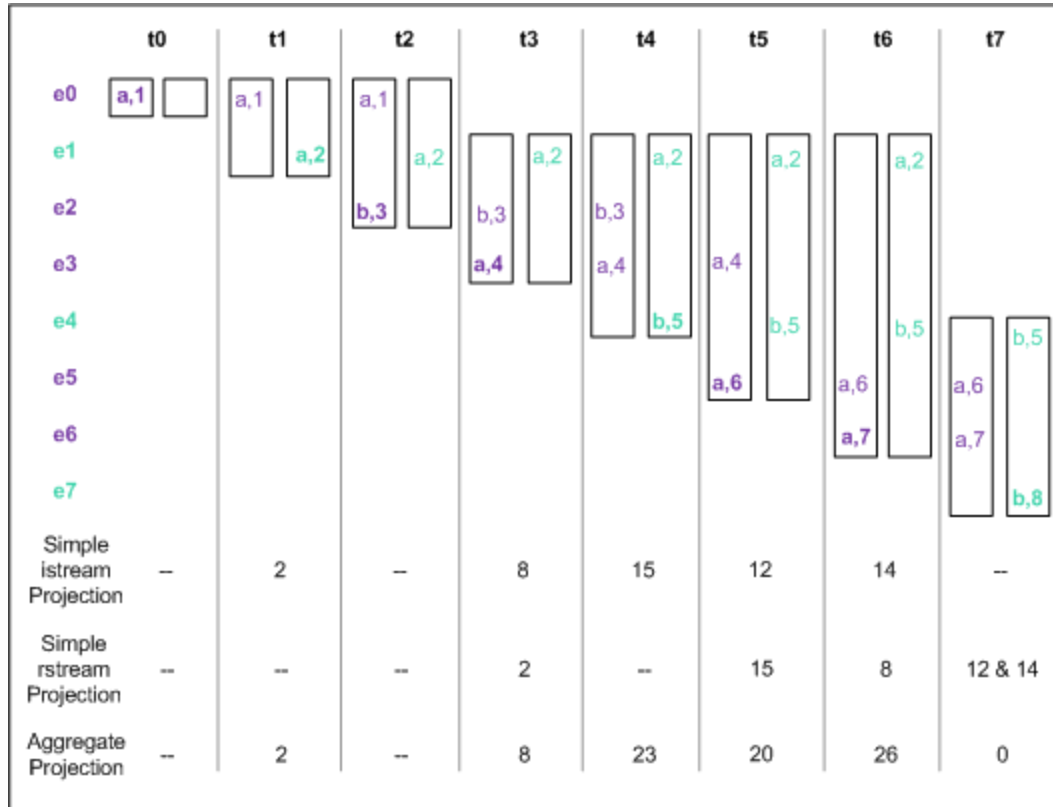
As with the partition and unique key expressions, each join key expression must return a comparable type (see [“Comparable types” on page 593](#)). Also, *joinKeyExpr1* must include a reference to *itemIdentifier1* and *joinKeyExpr2* must include a reference to *itemIdentifier2*. Each join key may not refer to the item from the other stream. An example of an equi-join is:

```

from p1 in leftPairs retain 2
  join p2 in rightPairs retain 2
    on p1.letter equals p2.letter
  select sum(p1.num * p2.num);

```

This is illustrated in the following diagram:



The query before the diagram corresponds to the aggregate projection. The three queries shown here are:

■ Simple istream projection:

```

from p1 in leftPairs retain 2
  join p2 in rightPairs retain 2
    on p1.letter equals p2.letter
  select p1.num * p2.num

```

■ Simple rstream projection:

```

from p1 in leftPairs retain 2
  join p2 in rightPairs retain 2
    on p1.letter equals p2.letter
  select rstream p1.num * p2.num

```

■ Aggregate projection:

```

from p1 in leftPairs retain 2
  join p2 in rightPairs retain 2
    on p1.letter equals p2.letter
  select sum(p1.num * p2.num);

```

This diagram shows the input that was used in the cross-join example, but with the join changed to be an equi-join. As you can see, only the items with matching letters appear in the output. The first event on the right side of the join has the same letter as the event on the left, so an output is produced as before. When the second event arrives on the left, however, no output is produced, because the letter does not match the other side. When a `b` event arrives on the right side of the join, that is joined with the `b` event on the left.

Finally, at the end of the table you can see that the join is empty because none of the events on the left match any of the events on the right.

Here is a more concrete example of an equi-join:

```
from r in priceRequest
  join p in prices partition by p.symbol retain 1
  on r.symbol equals p.symbol
  select p.price
```

For each new stock price request, this query generates the latest price for that stock/symbol. In an equi-join, whenever an item enters a window on one side, the correlator evaluates the join condition to determine if the item matches any of the items in the window on the other side. The correlator joins and outputs each matching pair when it finds one.

Typically, you want to create a derived event that is a function of the events on both sides of the join operation. Here is another example:

```
from latest in latestSensorReadings
  join average in averageSensorReadings
  on latest.sensorId equals average.sensorId
  select SensorAlert(latest.sensorId, latest.value, average.mean) as alert{
    send alert to "output";
  }
```

This query joins a stream of the most recent readings from all the sensors with a stream of averages of the same readings over some period. When a new reading appears it causes an event on the stream of averages at the same time. This causes them to be joined to create an alert that contains both the latest value and the latest average, which is then sent.

See also [“IEEE special values in stream query expressions” on page 221](#).

Filtering items before projection

In a stream query, after the window definition and any join clause, you can optionally specify a `where` clause to filter the items produced by the window or join. The `where` clause specifies an arbitrary EPL expression and can filter items based on any criteria available to EPL. The syntax of the `where` clause is as follows:

```
where booleanExpr
```

Replace *booleanExpr* with a Boolean expression. This expression is referred to as the `where` predicate. Only those items for which the `where` predicate evaluates to true are passed by the filter. For example:

```
from t in ticks retain 100
  where t.price*t.volume>threshold
```

```
select mean(t.price)
```

To calculate the mean price, this query operates on only the items whose value ($t.price * t.volume$) is greater than the specified threshold.

Performance

The filtering performed by the `where` clause happens after any window, with or join operations. In some cases, it is possible to rephrase the query to improve operational efficiency. For example:

```
from t in ticks within 60.0
  where t.price*t.volume>threshold
  select mean(t.price)
```

This query maintains a window of `Tick` items. Now consider this revision:

```
from p in
  (from t in ticks where t.price*t.volume>threshold select t.price)
  within 60.0
  select mean(p)
```

In the first example, the `within` window contains all `Tick` events received in the last minute. In the second example, the `where` clause is before the window definition so the filtering happens before items enter the window. Consequently, the window contains only `float` items for which the `where` predicate is true. These types of optimization are of particular benefit in queries that include both a `where` clause and a `join` operation (equi-join or cross-join). However, care must be taken when refactoring queries, particularly when size-based windows are involved. For example, consider the two queries below:

```
from t in ticks retain 100 where t.price*t.volume>threshold
  select mean(t.price)

from p in
  (from t in ticks where t.price*t.volume>threshold select t.price)
  retain 100 select mean(p)
```

These queries are not equivalent. The first query generates the mean of a subset of the last 100 items. The `where` predicate evaluated to true for only the items in the subset. The second query generates the mean of the last 100 items for which the `where` predicate evaluated to true.

Generating query results

The last component of a stream is the required projection definition, which specifies how to generate items for the query's output stream. A projection definition has the following syntax:

```
[group by groupByExpr[, groupByExpr]...] [having havingExpr]
select [rstream] selectExpr
```

Syntax Element	Description
<i>groupByExpr</i>	Each <i>groupByExpr</i> is an expression that returns a value of a comparable type. These expressions form the group key, which determines which group each output item is a part of. Any

Syntax Element	Description
	aggregate functions in the <code>having</code> or <code>select</code> expression operate over each group separately. See “Grouping output items” on page 218 .
<code>havingExpr</code>	The <code>havingExpr</code> expression filters output items. See “Filtering items in projections” on page 219 .
<code>selectExpr</code>	The value you specify for <code>selectExpr</code> defines the items that are the result of the query. The correlator evaluates <code>selectExpr</code> to generate each item that appears in the query's output stream. The type of <code>selectExpr</code> identifies the type of the query's output stream.

A projection can be one of the following kinds:

- A *simple* projection does not specify any aggregate functions, nor does it specify a `group by` or `having` clause. A simple projection can be a simple istream projection or a simple rstream projection.
- An *aggregate* projection specifies at least one aggregate function across the `having` and `select` expressions.

You can specify a `group by` clause as part of an aggregate projection. If there is a `group by` clause, the group key must be one or more expressions that take the input event and return a value of a comparable type.

You cannot specify `rstream` in an aggregate projection.

The following table describes the kinds of expressions that can appear in the `select` expression for each type of projection. In more complex expressions, the rules apply similarly to each sub-expression within that expression.

Kind of Expression	Valid in Projections	Description	Example
Non-item expression	Simple and aggregate	An external variable, constant, or method call. It does not refer to any of the input items.	<code>select currentTime;</code>
Item expression	Simple	A reference to the input item or a non-aggregate expression that contains at least one reference to the input item.	<code>select a.i;</code> <code>select sqrt(a.x)*5.0/a.y</code>

Kind of Expression	Valid in Projections	Description	Example
Group key expression	Aggregate	An expression that returns one of the group keys can also occur in the projection.	<code>group by a.i/10 select (a.i/10)*mean(a.x);</code>
Aggregate function expression	Aggregate	An expression that contains at least one aggregate function. Arguments to the aggregate function can include item expressions.	<code>select mean(a.i);</code>

Note:

An expression might not be syntactically equivalent to a `group by` expression even though it might appear to be equivalent. For example, if the `group by` expression is `a.i*10`, you cannot specify `10*a.i` as an equivalent expression. An equivalent `group by` expression must contain the exact sub-expression specified in the `group by` clause.

Aggregating items in projections

An aggregate function calculates a single value over a window. If a `select` expression contains any aggregate functions, then references to the input item can appear only in the arguments to those aggregate functions. Any EPL expression can appear in the arguments to the function, but other aggregate functions may not. EPL provides several built-in aggregate functions and you can define additional ones. See [“Defining custom aggregate functions” on page 221](#) and [“Built-in aggregate functions” on page 642](#).

Grouping output items

In a `select` clause, when you do not specify a `group by` clause any aggregate functions in the projection operate on all values in the window. This is true even if you partitioned the window. To group the items in the window into one or more separate groups and to calculate an aggregate value for each group of items, use the `group by` clause. The syntax of the `group by` clause is as follows:

```
group by groupByExpr[, groupByExpr]...
```

Each `groupByExpr` is an expression that returns a value of a comparable type. See [“Comparable types” on page 593](#).

These expressions form the group key, which determines which group each output item is a part of. Any aggregate functions in the `select` expression operate over each group separately.

In an aggregate projection, you can refer to any group key expressions anywhere in the `select` expression. However, you can refer to a query input item only in an aggregate function argument. For example:

```
from t in all Tick() within 30.0
  group by t.symbol select TickAverage(t.symbol, mean(t.price));
```

Whenever a lot arrives, this query updates one or more groups. Every group that is updated outputs a `TickAverage` event, and all `TickAverage` events are in the same lot. Each `TickAverage` event contains the symbol and the average price for that symbol over the last thirty seconds. If a group is not updated, it does not output a `TickAverage` event.

You typically use a `group by` clause in a stream query in conjunction with a `partition by` clause. In the following example, the window contains up to 10 events for each stock symbol. The aggregate projection calculates the average price separately for each symbol and each average is based on up to 10 events:

```
from t in ticks partition by t.symbol retain 10
  group by t.symbol select mean(t.price);
```

Obtaining the query's remove stream (rstream)

For each query, there are items that have been added to the window in a given query activation and items that have been removed (they were previously in the window, but are no longer in the window). By default, a simple, non-aggregate projection returns the items that have been added to the window. This is the insert stream (`istream`). To obtain the items that have been removed from the window, add the `rstream` keyword to the `select` clause.

For aggregate projections, obtaining the remove stream is not meaningful, and therefore the `rstream` keyword is not allowed in aggregate projections.

For examples of specifying `rstream`, see [“Defining time-based windows” on page 201](#), [“Defining size-based windows” on page 202](#), [“Defining cross-joins with two from clauses” on page 211](#) and [“Defining equi-joins with the join clause” on page 213](#).

When you specify `retain all`, you cannot specify `rstream`.

Filtering items in projections

In a stream query, as part of an aggregate projection definition, you can optionally specify a `having` clause to filter the items produced by the projection. The `having` clause specifies an arbitrary EPL expression and can filter items based on any criteria available to EPL. The syntax of the `having` clause is as follows:

```
having booleanExpr
```

Replace *booleanExpr* with a Boolean expression. This expression is referred to as the `having` predicate. The `having` predicate is evaluated for each lot that arrives. When the `having` predicate evaluates to `false`, the projection does not generate output.

Unlike the `where` clause, the `having` clause

- Is part of the projection

- Filters the output of the projection rather than what comes into the projection
- Cannot refer to individual items
- Can refer only to the group key or aggregates

A `having` clause can only be in an aggregate projection; it cannot be in a simple projection. Each aggregate projection must contain at least one aggregate in a `having` clause or in the `select` clause. Values for aggregates, whether in `having` expressions or `select` expressions, are always calculated over the same window(s). See [“Grouping output items” on page 218](#).

For example:

```
from t in all Temperature() within 60.0
  having count() > 10
  select mean(t.value)
```

This query calculates a rolling average of temperatures over the last minute. In this stream query, the `having` clause permits the average to be output only when it is a reliable measure. The `count()` aggregate function ensures that there are sufficient measurements (at least 10) in the previous 60 seconds to compensate for any noise or one-off errors in the readings.

Because the filtering occurs after the `select` expression has been processed, the average is still being calculated invisibly in the background, and can be output the very moment the measurement passes the reliability criterion. In the previous example, this means that after ten items have arrived, the average of all values in the last minute is output.

Filtering grouped aggregate projections

If you specify the `group by` clause, the `having` clause operates separately on each group, just as the `select` clause operates separately on each group. For example, the following code changes the previous code so that it outputs a reliable rolling average for each zone:

```
from t in all Temperature() within 60.0
  group by t.zone
  having count() > 10
  select ZoneAverage(t.zone, mean(t.value))
```

Just as a distinct mean is output for each group (each zone), the criterion for the `having` expression are applied separately to each group. A rolling average for a zone is output only when `count() > 10` is true for that zone.

Performance

It is possible for the stream network to avoid some calculations in a `select` clause when the `having` clause evaluates to `false`. Since maintaining aggregates can be expensive, this can be a useful optimization. When you know that a `having` clause can often evaluate to `false`, you can obtain better performance by specifying a `having` clause in the stream query as opposed to specifying a query like this:

```
from t in all Ticks(symbol="APMA") within 60.0 * 10.0
  select MeanStddev(mean(t.value), stddev(t.value)) as avg_sd {
    if(shouldOutput()) {
```

```

        send avg_sd to "output";
    }
}

```

This query computes a rolling average and standard deviation over the last ten minutes of a stock, and sends them to a dashboard or similar. Optionally, the output feed that sends out the rolling average and standard deviation can be turned off, and this is indicated by the return value of the `shouldOutput()` action. However, even when the output is turned off, `Tick` events still come in and the stream network still calculates the rolling average and standard deviation.

You can rewrite the code such that turning off the output terminates the query and turning on the output restarts the query. This option loses the state of the window and introduces a 10-minute lag before accurate output is available. A better option is to add a `having` clause so that turning off the output removes the performance penalty without losing state. For example:

```

from t in all Ticks(symbols="APMA") within 60.0 * 10.0
  having shouldOutput()
  select AvgStddev(mean(t.value), stddev(t.value)) as avg_sd {
    send avg_sd to "output";
  }

```

The `mean()` and `stddev()` aggregates continue to accumulate state when `shouldOutput()` returns `false`, but they do not fully calculate the rolling average and standard deviation for each incoming item.

IEEE special values in stream query expressions

The following information about IEEE special values applies to the following expressions:

- The key expression in a `with unique` clause
- A `partition by` expression
- The expressions that define the conditions in a `join` clause
- A `group by` expression

If one of these expressions is a `decimal` or `float` value, or a container that involves a `decimal` or `float` value, and the `decimal` or `float` value is an IEEE special value, then the following applies:

- **NaN** — This value is illegal as all or part of an expression and terminates the monitor instance.
- **Positive/negative infinity** — These values are legal and all positive infinities are treated as equal as are all negative infinities.

Defining custom aggregate functions

EPL provides a number of commonly used aggregate functions that you can specify in the `select` clause of a query. See [“Aggregating items in projections” on page 218](#). If none of these functions perform the operation you need, you can define a custom aggregate function. The format for defining a custom aggregate function is as follows:

```

aggregate [bounded|unbounded] aggregateName ([arglist])

```

```
returns retType { aggregateBody }
```

Element	Description
bounded unbounded	<p>Specify bounded when you are defining a custom aggregate function that will work with only a bounded window. That is, the query cannot specify <code>retain all</code>.</p> <p>Specify unbounded when you are defining a custom aggregate function that will work with only an unbounded window. That is, the query must specify <code>retain all</code>.</p> <p>Do not specify either bounded or unbounded when you are defining a custom aggregate function that will work with either a bounded or an unbounded window.</p> <p>If you do not specify bounded, you must define the custom aggregate function so that it can handle a window that never removes items. The function should not consume memory per item in the window.</p>
<i>aggregateName</i>	<p>Specify a name for your aggregate function. This is the name you will specify when you call the function in a <code>select</code> clause.</p> <p>For details about the characters you can specify, see “Identifiers” on page 687.</p>
<i>arglist</i>	<p>Optionally, specify one or more comma-separated type/name pairs. Each pair indicates the type and the name of an argument that you are passing to the function. For example, <code>(float price, integer quantity)</code>.</p>
<i>retType</i>	<p>Specify any EPL type. This is the type of the value that your function returns.</p>
<i>aggregateBody</i>	<p>The body of a custom aggregate function is similar to an event body. It can contain fields that are specific to one instance of the custom aggregate function and actions to operate on the state. The <code>init()</code>, <code>add()</code>, <code>remove()</code> and <code>value()</code> actions are special. They define how stream queries interact with custom aggregate functions.</p>

You define custom aggregate functions outside of an event or a monitor, and the function's scope is the package in which you declare it. To use custom aggregate functions in other packages, specify the function's fully-qualified name, for example:

```
from a in all A() select com.myCorporation.custom.myCustomAggregate(a)
```

Alternatively, you can specify a `using` statement. For example, suppose you define the `myCustomAggregate()` function in the `com.myCorporation.custom` package. To use that function

inside another package, insert a statement such as the following in the file that contains the monitor in which you want to use the function:

```
using com.myCorporation.custom.myCustomAggregate;
```

Insert the using statement after the optional package declaration, but before any other declarations. You can then simply specify the function name. For example:

```
from a in all A() select myCustomAggregate(a)
```

Be sure to inject the file that contains the function definition before you inject the files that contain monitors that use the function.

See also [“Names” on page 693](#).

Example of defining a custom aggregate function

The following example shows the definition of a custom aggregate function that returns the weighted standard deviation of the input values.

```
aggregate bounded wstddev( float x, float w ) returns float {
    // 1st argument is the value, 2nd is the weight.
    float s0;
    float s1;
    float s2;
    action add( float x, float w ) {
        if (w != 0.0) {
            s0 := s0 + w;
            s1 := s1 + w*x;
            s2 := s2 + w*x*x;
        }
    }
    action remove( float x, float w ) {
        if (w != 0.0) {
            s0 := s0 - w;
            s1 := s1 - w*x;
            s2 := s2 - w*x*x;
        }
    }
    action value() returns float {
        if (s0 != 0.0) { return ((s2 - s1*s1/s0)/s0).sqrt(); }
        else { return float.NAN; }
    }
}
```

Defining actions in custom aggregate functions

Certain actions in a custom aggregate function have special meanings, and you must define them as follows:

- `init()` — The `init()` action is optional. If a custom aggregate function defines an `init()` action, it must take no arguments and must not return a value. The correlator executes the `init()` action once for each new aggregate function instance it creates in a stream query.

- `add()` — A custom aggregate function must define an `add()` action. The `add()` action must take the same ordered set of arguments that are specified in the custom aggregate function signature. That is, the names, types, and order of the arguments must all be the same. The correlator executes the `add()` action once for each item added to the set of items that the aggregate function is operating on.
- `remove()` — A bounded aggregate function must define a `remove()` action. An unbounded aggregate function must not define a `remove()` action. If you do not specify either bounded or unbounded, the `remove()` action is optional. The `remove()` action must take the same ordered set of arguments as the `add()` action and must not return a value. The correlator executes the `remove()` action once for each item that leaves the set of items that the aggregate function is operating on. The value that `remove()` is called with is the same value that `add()` was called with.
- `value()` — All custom aggregate functions must define a `value()` action. The `value()` action must take no arguments and its return type must match the return type in the aggregate function signature. The correlator executes the `value()` action once per lot per aggregate function instance and returns the current aggregate value to the query.

Custom aggregate functions can declare other actions, including actions that are executed by the above named actions. A custom aggregate function cannot contain a field whose name is `onBeginRecovery`, `onConcludeRecovery`, `init`, `add`, `value`, or `remove`, even if, for example, the custom aggregate function does not define a `remove()` action.

Overloading in custom aggregate functions

As with event types, the names of custom aggregate functions must be unique. Unlike the built-in aggregate functions, there is no overloading, so it is not possible to declare two aggregate functions with the same name and different parameters or two aggregate functions with different bounded and unbounded specifiers and the same name. For example:

```
aggregate unbounded max( float value) returns float {...}
aggregate bounded max( float value) returns float {...}
// Error! You cannot use the same function name.

aggregate unbounded maxu( float value) returns float {...}
aggregate bounded maxb( float value) returns float {...}
// Both of these queries are correct. They have different names.
```

In contrast, the built-in bounded and unbounded aggregate functions are overloaded.

Distinguishing duplicate values in custom aggregate functions

Each item in a stream is considered to be unique. However, when duplicate values appear in the set of items that a custom aggregate function operates on, it is not possible for the function to identify the particular instance of the value. If your implementation requires being able to distinguish between instances of duplicate values, you can accomplish this by extending the signatures of the function's `add()` and `remove()` actions.

For example, you might see the following set of `float` values in a stream:


```
1.0  2.0  3.0  4.0  3.0  2.0  1.0
```

Each occurrence of a particular value in the stream represents an individual value, separate from any other occurrences of that value. But when a query presents these values to a custom aggregate function (by means of the `add()` and `remove()` actions), the value alone is not enough to identify the particular occurrence that this value represents.

To distinguish one occurrence from another, extend the action signatures as follows:

- The `add()` action can return a value, which can be of any type.
- If the `add()` action does return a value, then the `remove()` action must accept, as its last argument in addition to its standard arguments, an argument of the same type as that returned by the `add()` action.

When an item is added to the aggregate, the value returned by the `add()` action is stored with the item. When that item is removed from the aggregate, the same value will be passed to the `remove()` action. Thus, it is possible to distinguish between items with duplicate values by comparing the additional data that is passed to the `remove()` action.

The following example shows an aggregate function that returns the entire window contents, in order, as a sequence:

```
aggregate windowOf(float f) returns sequence<float> {
    dictionary<integer,float> d;
    integer i;
    action init() { d.clear(); i := 0; }
    action add(float f) returns integer {
        i := i+1;
        d[i] := f;
        return i;
    }
    action remove(float f, integer k) { d.remove(k); }
    action value() returns sequence<float> { return d.values(); }
}
```

Working with lots that contain multiple items

Each time a stream query or stream listener is activated, it might be processing more than one item at a time. Each simultaneously processed group of items is referred to as a *lot*. Like an auction lot, a lot can contain just one item or it can contain a number of items. Stream listeners can be activated once per item or once per lot. Stream queries try to process each item in a lot as if it arrived separately. See [“Behavior of stream queries with lots” on page 226](#) for a discussion of cases where this is not possible.

When a lot contains multiple items, all items in the lot appear in the output stream at the same time. However, the correlator preserves the order in which the stream query generated the items in the lot. When that output stream is the input stream for another stream query, the subsequent query uses the preserved order, if necessary, to determine how to process the items.

Stream queries that generate lots

To generate a lot that contains multiple items, a stream query must specify a simple projection or an aggregate projection that contains a `group by` clause. The stream query must also either receive lots that contain multiple items or must contain one of the following:

- A batched window.
- A timed window with the `rstream` keyword (this must be a simple projection, and not an aggregate projection).
- A join of either type.

A query with a non-grouped aggregate projection never generates multiple items. It generates a single item or nothing.

A timed window with the `rstream` keyword can generate lots because multiple items can have the same timestamp. In a timed window, when items with the same timestamp expire, they all leave the window at the same time. However, the correlator still maintains the order in which the items were generated or received.

Behavior of stream queries with lots

This topic provides advanced information about how queries process lots that they receive on their input streams. The information here requires a thorough understanding of streams, queries, and the information about lots presented so far.

To understand how stream queries behave when receiving lots that contain more than one item, consider the window content of the query before the lot is input and the window content of the query after the lot is input. The difference between these two states determines the output of the query. For example, consider the following queries:

```
// event A { float x; }  
stream<A>      sA := from a in all A() retain 3 every 3 select a;  
stream<float> sB := from a in sA select a.x;  
stream<float> sC := from a in sA select sum(a.x);
```

The following diagram shows the lot output by each stream on each activation of the query.

	t0	t1	t2	t3	t4	t5
e0	A(1.)	A(1.)	A(1.)			
e1		A(2.)	A(2.)			
e2			A(3.)			
e3				A(4.)	A(4.)	A(4.)
e4					A(5.)	A(5.)
e5						A(6.)
sA Output	--	--	A(1.) A(2.) A(3.)	--	--	A(4.) A(5.) A(6.)
sB Output	--	--	1. 2. 3.	--	--	4. 5. 6.
sC Output	--	--	6.	--	--	15.

As can be seen, in the queries that contain aggregate functions, the aggregate expressions (and projections) are evaluated, at most, once per query activation. All queries, with the exception of those containing a group by clause, behave in this way.

Size-based windows and lots

When a size-based window is processing a lot that contains more than one item, all of the items are processed in the window before any of the rest of the stream query is processed. None of the intermediate states are visible to the query. This means that in the following query:

```
from a in sA retain 3 select sum(a.i);
```

if the window contains the events A(1), A(2) and A(3) and a lot containing both A(4) and A(5) arrives, those will displace A(1) and A(2) immediately. The state of the window A(2), A(3), A(4) will never have existed. This is more relevant when the lot contains more items than will fit in the window. In this case, if five more events arrived in a single lot, the three events will fall out of the window, the last three events will go into the window and the two interim events will disappear – never having been in the window at any point.

This behavior means that care must be taken with fixed-size windows when events might be processed in lots.

Join operations and lots

The principle of updating the state of a query in a single operation without the intermediate state being visible is most relevant for join operations. The two diagrams that follow illustrate how a cross-join behaves when several events arrive in a single lot.

In the diagrams, the items on the left side of the join are represented by the numbered items that come in from the left side, and the items on the right side of the join are represented by the lettered items that come in from the top. Each square in the grid can be a joined event. In both diagrams, the results of the join before the lot arrives are mostly highlighted in blue. The items joined after the lot arrives are mostly highlighted in teal. The relevant stream query in both examples is:

```
from a in sA retain 3
  from b in sB retain 3
  select C(a, b);
```

The complete set of values in the table represents all of the combinations of items from *sA* and items from *sB* that could possibly be generated by the join when considering alternative ways of ordering the *sA* and *sB* items arriving in the lot. In general, there is no particular ordering of the *sA* and *sB* items that is superior (more meaningful) than all other orderings. Thus, when considering the transitions, there is no preferred path from the initial window content to the final window content. Hence, it is considered that the correct output for the join is achieved by taking the difference between the initial window content and the final window content, ignoring any intermediate states.

		sB			
		a	b	c	d
sA	1				
	2				
	3				
	4				
	5				

In the first diagram, there are nine joined events before the lot arrives. These are represented by the seven blue squares and the two orange squares. Two items, 4 and 5, arrive on *sA* and displace items 1 and 2. Also, one item, *d*, arrives on *sB* and displaces item *a*. The result is nine joined events after the lot arrives, of which two were there before (represented by the two orange squares, and seven are new, represented by the teal squares. A non-aggregating query that outputs the *istream* (as given above) would return the seven new items (shown in teal). If, instead, the query was selecting the *rstream*, then it would return the seven items that are no longer a result of the join (shown in blue).

	sB						
	a	b	c	d	e	f	g
sA	1						
	2						
	3						
	4						
	5						
	6						
	7						

In the second example, there are again nine joined events before the lot arrives. These are represented by the nine blue squares. Four items, 4, 5, 6, and 7 arrive on sA and displace items 1, 2, and 3. Because this is a `retain 3` window, item 4, as the oldest item in the lot, never makes it into the window. Also, items d, e, f, and g arrive on sB, which displaces items a, b, and c, and again, because it is a `retain 3` window, item d never appears in the window. After the lot arrives, the result is nine new joined events, which are represented by the teal squares.

Since there are no joined events that are present both before the lot arrives and after the lot arrives, all nine events that were previously the result of the join would be returned by a query selecting the remove stream of this join. The nine new events are output by the query that selects the input stream. No events containing either 4 or d are ever visible as a result of the query, even though both values were present on one of the inputs.

Grouped projections and lots

Suppose that a query that contains a `group by` clause processes a lot that contains several items. The query generates new projected items for the groups where the state of the group after the lot is input differs from the state of the group before the lot is input.

Stream network lifetime

After you create a stream or stream listener, it exists until one of the following happens:

- You explicitly terminate it.
- The monitor that contains the stream or stream listener terminates.

- You terminate another stream or stream listener in the same stream network and that causes the stream or stream listener to terminate.

A stream or stream listener is explicitly terminated by calling the `quit()` method on a variable that refers to it. Hence, to explicitly terminate a stream or stream listener, you must retain a reference to it. You can also terminate a stream or stream listener by terminating a related stream or stream listener in the same stream network (as detailed below).

You can create a stream or stream listener that is not referenced by any variable and cannot be terminated by quitting any other streams or stream listeners in the stream network. If this is unintentional, then we refer to it as a stream or stream listener leak. This situation is similar to an event listener leak (see [“Avoiding listeners and monitor instances that never terminate” on page 423](#)). Here is an example:

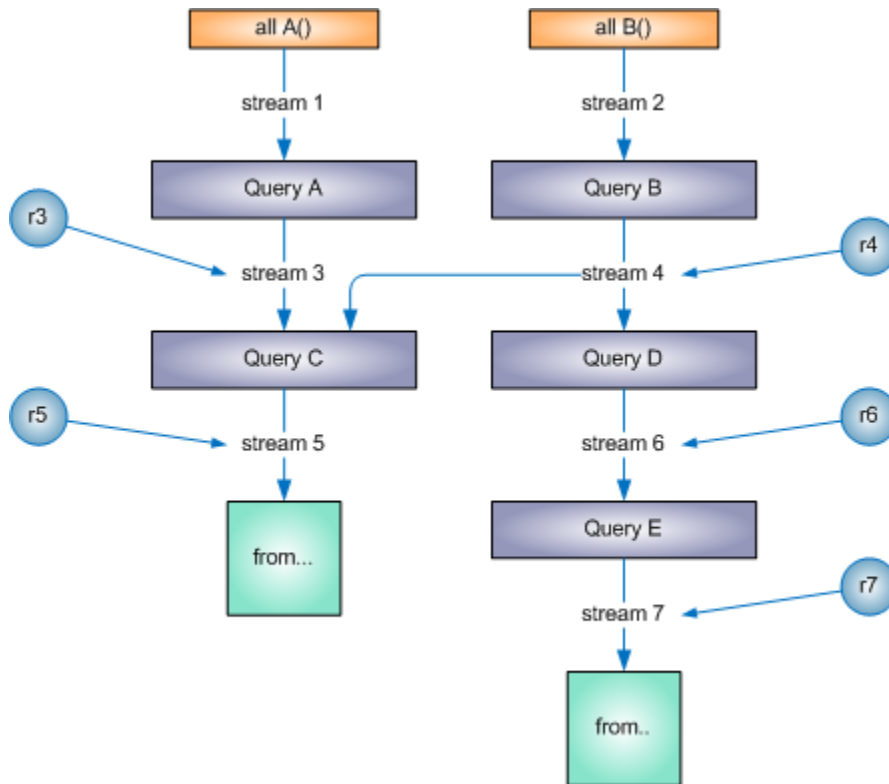
```
action createStreamListener() returns listener {
    stream <A> sA := all A();
    listener l := from a in all A() select a.x as x { print x.toString(); }
    return l;
    // error: meant to use sA in the query above
}
```

Although executing the code returns a listener variable that refers to the created stream listener, it inadvertently creates an unreferenced stream (the local variable `sA` did refer to this stream but is no longer in scope).

Calling `quit()` on a stream or stream listener in a stream network typically has side effects. A side effect can be one of the following:

- Termination of additional streams, stream queries, stream listeners, or stream event expressions.
- Disconnection between the terminated element and another element.

When determining which queries to terminate, the correlator uses the following rule: when, due to another stream or query terminating, a query can no longer generate any output, it is also terminated. For example, the following diagram shows a stream network with two stream source templates generating input events for five queries, eventually connected to two stream listeners. There are four stream variables pointing to the streams in the network.



Suppose you call `quit()` on either `r6` or `r7` (the stream variables on the right). The correlator terminates the whole of the branch from Query D down. This is because, whichever stream you quit, nothing can be generated by anything connected to those streams. `stream 4`, however, is also feeding Query C, which can still generate output. Therefore, the rest of the network, including Query B and both stream event expressions, remains active.

If you subsequently call `quit()` on `r5`, this will terminate the stream listener and Query C, which will then terminate `stream 3` and `stream 4`, since they are not connected to any other queries, and also `stream 1`, `stream 2` and both stream source templates.

The stream variables after their streams are terminated will be dummy references. Subsequent attempts to create a query using those streams are ignored (the result is an inert stream).

Disconnection vs termination

In the example above, quitting `r6` disconnects Query D from `stream 4`. Because `stream 4` has other stream queries using it, this disconnection does not terminate `stream 4` immediately. Streams terminate when all the queries using them have disconnected.

If you were instead to call `quit()` on `r4`, this would terminate everything on the right side of the diagram, no matter how many queries are using `stream 4`. However, the stream would just be disconnected from Query C. Whether this terminates Query C depends on the state of the join in Query C. If it is joining a size-based window from `stream 4`, the items in the window would remain to be joined against new items in `stream 3`. If it was a time-based window, then Query C would remain until everything in the window had been discarded. At that point, since nothing can ever

be added to that side of a join, Query C terminates, causing the rest of the network to also be terminated.

Rules for termination of stream networks

The complete set of rules for when a part of a stream network is terminated are:

- Stream listeners:
 - `quit()` is called on a `listener` variable pointing at that stream listener.
 - The stream the listener is connected to is terminated.
- Streams:
 - `quit()` is called on a `stream` variable pointing at that stream.
 - The stream query generating the stream is terminated.
 - All the stream queries using the stream are terminated.
- Stream queries:
 - The stream the query generates is terminated.
 - All of the streams the query uses are terminated and either the query does not define a window or it defines a `within` or `within...every` window and there are no live items in the window.

A live item is an item whose expiration (the item falls out of the window) can cause query output. For example, if the only items in a timed window fail to satisfy a `where` clause in the window definition, then those items cannot change query output when they expire.

If none of the items in the window are live, the query terminates when all items have fallen out of the window. However, the query might terminate earlier if the correlator can determine that none of the items are live and that all streams that the query uses have terminated. Regardless of when such a query quits, there are no observable effects except in two situations:

 - The query is the only thing keeping the monitor active. That is, when the query terminates, then the monitor's `ondie()` action is called.
 - Calculation of the size of the window has one or more side effects.
- Stream source templates:
 - The stream the stream source template generates is terminated.

Using dynamic expressions in stream queries

The expressions in stream queries can contain variables and action calls from EPL. Unlike parameters to event templates, the correlator evaluates these expressions each time the query is

used and not just when it is created. This allows the behavior of the query to be altered during program execution.

Behavior of static and dynamic expressions in stream queries

A static expression is an expression that refers to only static elements. Static elements are:

- Constants (defined with the `constant` keyword)
- Literal values, for example:

```
from a in all A() within 20.0 select sum(a.i);
```

- Primitive types that are local variables, for example:

```
integer width := 10;
from a in all A() retain width select sum(a.i);
```

The correlator can fully evaluate static expressions when it creates the stream query.

A dynamic expression is an expression that refers to one or more dynamic elements. In a query, the value of a dynamic expression can change throughout the lifetime of that query. Consequently, the correlator must re-evaluate each dynamic expression at appropriate points in the execution of the query.

Dynamic elements are:

- Any reference type
- Any monitor global variable
- Where the stream query is created by an action on an event, the members of that event
- Any action, method or plug-in call

The correlator fully evaluates an event template in a stream source template when the correlator creates the query. For example, consider the following two queries:

```
from a in all A(id=currentMatch) select a;
from a in all A() where id = currentMatch select a;
```

During execution, if `currentMatch` is a global variable, a change to the value of `currentMatch` affects the behavior of the second query, but it does not affect the behavior of the first query.

When to avoid dynamic expressions in stream queries

Where possible, use static expressions in preference to dynamic expressions. This allows the compiler to optimize the query to improve performance. For example, consider the following query:

```
stream<float> vwaps := from t in all ticks
    within vwapPeriod
    select wavg(t.price,t.volume);
```

When `vwapPeriod` is a monitor global variable whose value does not change, then it is preferable to copy the value to a local variable first. For example:

```
float period := vwapPeriod;
stream<float> vwaps := from t in all ticks
  within period
  select wavg(t.price,t.volume);
```

Similarly, if it is known that a given action call always returns the same value, then it is preferable to copy the result to a local variable and use this in place of the action call. For example:

```
float period := getVwapPeriod(symbol);
stream<float> vwaps := from t in all ticks
  within period
  select wavg(t.price,t.volume);
```

Ordering and side effects in stream queries

To determine when it is safe to use dynamic expressions in stream queries, it is important to understand that:

- In a query, the order in which the correlator executes the action calls is not defined. Although the order is not defined, the correlator always executes the action calls in the same order for a particular Apama release.
- When processing each item passed to the query, if an action call with a given set of arguments appears multiple times within a stream query, then the number of times the correlator executes the action is not specified. It might be equal to or less than the number of times that the action call appears within the query. However, this number is always the same for a particular release.
- In a stream network, the order in which the correlator executes the queries is not defined except for when the output of a query forms the input to a second query. In this case, the correlator always executes the first query before the second. Again, in a particular release, the execution order is always the same.

Because of these points, it is best to avoid actions with side effects in expressions executed in stream queries. Such actions can make a program more difficult to understand and debug. Instead, execute any such actions in stream listeners.

A method or expression that produces a value has a side effect if it modifies something or interacts with something outside the program. This includes, but is not limited to:

- Modifying a global variable
- Changing the value of an argument
- Calling plug-in methods
- Routing, enqueueing, emitting or sending an event
- Calling another action that has side effects
- Setting up event listeners or new streams

Understanding when the correlator evaluates particular expressions

All expressions in a stream query can contain dynamic elements. To understand the behavior of a query that specifies dynamic elements, it is necessary to know under what circumstances the correlator re-evaluates an expression and uses the result in the query.

Using dynamic expressions in windows

A window definition can contain some or all of the following:

- A partition key expression
- The window duration, size or both duration and size
- An every batch period or size
- The key for a `with unique` clause

The following table shows when the correlator evaluates each of these:

Window Definition	Description
<code>retain <i>n</i></code>	The correlator evaluates <i>n</i> every time an item arrives on the stream. The correlator uses the new value of <i>n</i> to calculate what should be in the window.
<code>retain <i>n</i> every <i>m</i></code>	The correlator stores incoming items until the current value of <i>m</i> is satisfied. When <i>m</i> is satisfied, the correlator evaluates both <i>n</i> and <i>m</i> . The correlator uses the new value of <i>n</i> to calculate what should be in the window, including the stored items. Because <i>m</i> is evaluated only after it has been satisfied, meeting that condition is always based on the old value of <i>m</i> .
<code>within <i>d</i></code>	The correlator evaluates <i>d</i> every time an item arrives on the stream and every time an item is due to be removed from the window. The correlator uses the new value of <i>d</i> to calculate what should be in the window.
<code>within <i>d</i> every <i>p</i></code>	<p>The correlator stores incoming items until <i>p</i> seconds have elapsed. When <i>p</i> seconds have elapsed, the correlator evaluates <i>p</i> and <i>d</i> only if there are any items in the window or stored. The correlator uses the new value of <i>d</i> to calculate what should be in the window, including stored events. The correlator uses the new value of <i>p</i> to determine the next time the window can change.</p> <p>If there are no items in the window or waiting to enter the window then, for efficiency, the correlator does not evaluate <i>p</i>. When the correlator evaluates <i>p</i>, it is always based on the old value of <i>p</i>.</p>

Window Definition	Description
<code>...retain <i>n</i></code>	<p>If a <code>within</code> or <code>within every</code> window definition also specifies <code>retain</code>, the correlator evaluates <i>n</i> whenever the window content can change. The correlator uses the new value of <i>n</i> to calculate what should be in the window.</p> <p>If the window definition specifies <code>every</code>, the window content can change only when <i>p</i> is satisfied.</p> <p>Otherwise, the window content can change when an item arrives on the stream and when an item is due to be removed from the window.</p>
<code>partition by <i>k1</i>[, <i>k2</i>]...</code>	<p>If the window definition specifies a timed <code>every <i>p</i></code> clause, the correlator evaluates each partition expression when <i>p</i> seconds have elapsed. Otherwise, the correlator evaluates each key expression when an item arrives on the stream. The correlator uses the new value of each key expression to calculate what should be in each partition.</p>
<code>with unique <i>w</i></code>	<p>The correlator evaluates <i>w</i> once for each item whenever that item is about to enter the window. If there is an <code>every</code> clause, an item can enter the window only when <i>m</i> or <i>p</i> is satisfied. Otherwise, an item can enter the window when it arrives on the stream.</p>

Using dynamic expressions in equi-joins

The format of a query that contains an equi-join is as follows:

```
from x in s1 join y in s2 on j1 equals j2 ...
```

Suppose that *j1* and *j2* are dynamic expressions that return the left and right join keys for each input item. The correlator evaluates these expressions once for each input item when it enters the window. This is regardless of how many items are joined from the other side.

Using dynamic expressions in where predicates

The correlator evaluates the predicate in a `where` clause once for each item. This happens as soon as a join operation produces an item, or if there is no join operation, as soon as an item enters a window.

Using dynamic expressions in projections

In a simple projection, the correlator evaluates the `select` expression once for each item. The correlator evaluates the `select` expression as soon as a join operation produces an item, or if there is no join operation, as soon as an item enters a window.

In a simple projection, regardless of whether the `select` clause specifies the `rstream` keyword, the correlator evaluates expressions in the projection when the items would be present on the insert stream and the results are stored until needed for the remove stream.

In an aggregate projection, the correlator evaluates expressions in the projection when the items would be present on the insert stream.

If an aggregate projection contains a `group by` clause the correlator evaluates the group key once for each item. This happens as soon as a join operation produces an item, or if there is no join operation, as soon as an item enters a window.

The correlator evaluates aggregate and grouped expressions in two stages. The correlator evaluates arguments to aggregate functions once for each item as soon as it is produced by a join or if there is no join, as soon as it arrives in the window. The correlator evaluates the rest of the aggregate expression once for each lot.

Examples of using dynamic expressions in stream queries

Following are some examples of using dynamic elements in stream queries. These examples are simplified, for brevity.

Example of altering query window size or period

The following code fragment shows part of a monitor that accepts requests from external entities to monitor/generate the volume-weighted average price (VWAP) for a given symbol. After you create a monitor like this, an external entity can, at any time, change the parameters that control the period over which the monitor calculates the VWAP and/or the output frequency of the VWAP events.

```
monitor VwapMonitor {
  VwapRequestParams params;
  action onload() {
    on all VwapRequest() as v spawn monitorVwap(v);
    // Simplified. Assumes no duplicate requests.
  }
  action monitorVwap(VwapRequest v) {
    params := v.params;
    from t in all Ticks(symbol=v.symbol)
      within params.duration
      every params.period
      select Vwap(t.symbol,wavg(t.price,t.volume)) as vwap {
        route vwap;
      }
    on all VwapRequestUpdate(symbol=v.symbol) as u {
      params := u.params;
    }
  }
}
```

When accumulating the raw tick data to generate the VWAP price, no prescience is involved. There is no anticipation that the window size is to be increased. Changing the `within` duration to a larger value causes the window duration to increase but does not recover historic events. Hence the effective sample duration over which the monitor calculates the VWAP will, over time (as new

tick items arrive), extend from the smaller setting to the larger setting. When switching from a larger within duration to a smaller one, the change takes effect immediately. The correlator discards the items that are no longer in the within duration.

Example of altering a threshold

The following code fragment shows part of a monitor that accepts requests from external entities to monitor the value of the trades for a given symbol. After you create a monitor like this, an external entity can, at any time, change the thresholds at which the monitor recognizes the trade as a high value trade.

```
monitor CountHighValueTicks {
    float threshold;
    action onload() {
        on all CountHighValueTicksRequest() as r {spawn
            monitorHighValueTicks (r);
        }
    }
    // Simplified. Assumes no duplicate requests.
    action monitorHighValueTicks(CountHighValueTicksRequest r) {
        threshold := r.threshold;
        stream<Tick> filtered := from t in all Tick(symbol=r.symbol)
                                where t.price*t.volume > threshold
                                select t;
        from t in filtered within 60.0 every 60.0 select count() as c {
            print "Count of high value trades in previous minute: " +
                c.toString();
        }
        on all CountHighValueTicksRequestUpdate(symbol=r.symbol) as u {
            threshold := u.threshold ; }
    }
}
```

This example uses two queries. The first query filters out any ticks with values below the threshold. The second query accumulates the high-value ticks received in the last minute and outputs the count of high-value ticks in that period. This could have been written as a single query with the filtering performed after the window operation. For example:

```
from t in all Ticks(symbol=v.symbol) within 60.0 every 60.0
where t.price*t.volume > threshold select count();
```

However this query's window contains all of the low value ticks received in the last 60 seconds, as well as the high value ticks. This is not an optimal use of memory resources. Hence the two query approach is preferred.

Alternatively, you can specify an embedded query to amalgamate the two queries into a single statement:

```
from t in
    (from t2 in ticks where t2.price*t2.volume > threshold select t2 )
within 60.0 every 60.0
select count() as c { ... }
```

The parentheses around the embedded query are optional.

Example of looking up values in a dictionary

The following statement shows a query that calculates the current value of a basket of stocks based on the most recent prices for those stocks. When using dictionaries in this way, be careful to ensure that all values used as keys are in the dictionary. A missing key value causes a runtime error and the correlator terminates the monitor instance. In the example, it is assumed that the `prices` stream was filtered to contain prices for only the stocks in the basket.

```
stream<Tick> basketPrices :=
  from p in prices
  partition by p.symbol
  retain 1
  select sum( p.price * basketVolume[t.symbol] );
```

Example of actions and methods in dynamic expressions

Actions and methods can be considered to be dynamic elements. There are various reasons why you might want to use actions and methods in queries:

- If you are using a particular common complex expression in several places in queries within a monitor, it might be preferable to implement this as an action.
- If you are using a method that is implemented in a plug-in.
- To add protection to expressions that, if unprotected, might cause runtime errors. For example:

```
stream<Tick> basketPrices :=
  from p in prices
  partition by p.symbol
  retain 1
  select sum( p.price * getBasketVolume(t.symbol) );
...
action getBasketVolume( string symbol) returns float {
  if ( basketVolume.containsKey(t.symbol) ) {
    return basketVolume[t.symbol];
  } else {
    return 0.0;
  }
}
```

Troubleshooting and stream query coding guidelines

This section provides high-level guidelines for writing stream query applications that implement best practices.

Prefer on statements to from statements

Do not use streams unnecessarily. If an event expression in an `on` statement meets your needs, use it. Take advantage of mixing code elements for listeners and event expressions, stream processing, and responsive program actions, all in the same monitor.

Know when to spawn and when to partition

As a rule, you should listen for only those events or streams that you are interested in now. Apama applications typically define monitors that spawn to handle a new situation, for example, to automatically manage the trading of a new large order. Each monitor instance is usually interested in only one particular substream of a larger stream, for example, `Tick` events for a particular stock rather than all `Tick` events.

Consequently, the common pattern is to create a new monitor instance and for that instance to set up stream queries that process the events of interest, for example, to calculate the average price. This is more efficient than defining a monitor that processes all events (for example, all `Tick` events for all stocks), generates added-value items and then forwards these items to client monitors. However, there are situations when the latter approach is required. You should decide which solution approach is best in which circumstances.

Filter early to minimize resource usage

To minimize processing and memory overhead, it is preferable to filter streams as early as possible in the processing chain or network. Filtering early can reduce the number of items processed or retained in memory and can also reduce the size of the items held. If possible, filter items right at the beginning of the query chain, that is, in the event template.

For example, it is preferable to rewrite this query:

```
from l in all LargeEvent()  
  within largeWindowPeriod  
  where l.key = key  
  select mean(l.value);
```

If the key is static, rewrite it this way:

```
from l in all LargeEvent(key=key)  
  within largeWindowPeriod  
  select mean(l.value);
```

If the key is dynamic, rewrite it this way:

```
from v in  
  from l in all LargeEvent()  
    where l.key = key select l.value  
  within largeWindowPeriod select mean(v);
```

In the static case, the correlator filters the large event before the event gets to the window. In the dynamic case, the embedded query filters the event before the event gets to the window in the enclosing query. Because the `select` statement specifies only `l.value`, the correlator discards the rest of the event. There is no need to bring the whole event into the window.

Avoid duplication of stream source template expressions

When you are maintaining code, you might add a stream query whose *streamExpr* is an event template that is already used in a query elsewhere in the same monitor. However, duplicated

stream source template expressions do not always produce the behavior you want. Consider the following two code fragments:

```
stream<float> means := from t in all Temperature()
    within 10.0
    select mean(t.temperature);
from t in all Temperature()
    from m in means select t-m as d {
    print "Difference from mean is " + d.toString();
}
```

The first fragment behaves differently than this fragment:

```
stream<float> temperatures := all Temperature();
stream<float> means := from t in temperatures
    within 10.0
    select mean(t.temperature);
from t in temperatures
    from m in means
    select t-m as d {
    print "Difference from mean is " + d.toString();
}
```

Of the two code fragments above, the second one has the desired behavior. The first example creates two event listeners, one for each `all Temperature()` clause. Each listener matches each incoming `Temperature` event, but the listeners trigger independently, one after the other. This means that there is no time when the second query has an item in each of its source streams. Consequently, the cross-join never produces any output.

In the second example, there is a single `Temperature` event listener that places matching events in the `temperatures` stream. The `temperatures` stream is the source stream for two queries. Now both source streams of the last query contain items at the same time and the query generates output.

Avoid using large windows where possible

In Apama, all data being processed is held in memory, including data within stream windows. If you specify query windows that contain a large number of items or hold items for a long period of time, the memory that the application uses necessarily increases.

A memory requirement that is more than the memory available to the application causes paging to occur, which can decrease application throughput. Where possible, consider reducing the size of any stream query windows by doing one or more of the following:

- Filter items to reduce the number or size of the items in the window.
- Use a complex event expression to achieve the same result.
- Use `retain all` instead of specifying a `within` clause. See [“In some cases prefer retain all to a timed window” on page 242](#) for details.

In some cases prefer retain all to a timed window

When you specify `retain all` in a stream query, the correlator does not retain the items indefinitely. The correlator processes each new item when it arrives (for example, it might execute an aggregate function) and then discards it. Consequently, queries that specify `retain all` use less memory than queries that define time-based or size-based windows.

A situation that typically tempts you to define a time-based window is when you want to calculate some aggregate values for a session. For example, a session could be from the start of a day to the end of a day, or an incoming event could initiate a session that requires aggregated values such as placing an order in an automated trading system.

After the session begins, interest in the aggregated values usually continues until the session ends, for example, at the end of day or when the full volume of the placed order has been traded. In situations such as these, use a `retain all` window instead of a `within session` window.

Prefer equi-joins to cross-joins

In a query using an equi-join, the items from the two input sets are joined based on equality of key values. The identification of matching items is very efficient.

Cross-joins have no expressions, so it is more efficient to calculate them than equi-joins. However, cross-joins are less preferable to equi-joins if they produce unwanted items that must subsequently be filtered out.

Be aware that time-based windows can empty

Consider the query below:

```
from s in Shipment(destination="SPQ")
  within 604800.0
  select sum(s.qty)/count()
```

After creation of the query, suppose that several shipments are sent in the first week and no shipments are sent in the second week. The value of the `count()` aggregate function drops to zero, which results in an attempt to divide by zero. This terminates the monitor instance.

Be aware that fixed-size windows can overflow

Consider the following example:

```
stream<temperature> batchedTemperatures :=
  from t in all Temperature(sensorId="S001")
  within 60.0 every 60.0 select t;
from t in batchedTemperatures
  retain 5
  select count() as c { print c.toString(); }
```

During execution of the first query, suppose that more than 5 matching events are found within one minute. The query outputs all of the matching events as a single lot. A lot that contains more

than 5 items overflows the `retain` window in the second query. All but the most recent five items are lost. Calculations operate on only the most recent 5 items.

Note that you are unlikely to need the query combination shown in the code example above.

Beware of accidental stream leaks

Just as it is possible to leak event listeners, it is also possible to leak streams. Suppose that you create a stream but you do not specify the stream as input to any query. This stream still remains in existence, keeps a monitor instance alive, and consumes resources so it is considered to be a stream leak. A stream leak causes memory to be used and not freed. It can also cause unnecessary computation to occur.

A stream leak can happen if you create a stream that you want to use later on in your code. To be able to use this stream, you must assign it to a stream variable that is in scope in the location where you want to use the stream. If the stream variable goes out of scope or you assign another stream to that variable, the original stream still exists within the monitor instance's internal stream network, but it is no longer accessible. For example:

- The stream variable that references the stream goes out of scope:

```
action streamLeakExample1(string s) {
    stream<float> prices :=
        from t in all Tick(symbol=s) select t.price;
    ... // If the elided code does not use the stream
}
    // a leak occurs when the prices variable goes out of scope.
```

- You overwrite the stream variable that refers to an unused stream:

```
action streamLeakExample2(pattern<string> symbols) {
    string s;
    stream<float> prices;
    for s in symbols {
        prices := from t in all Tick(symbol=s) select t.price;
        ... // If the elided code does not use the prices stream
            // a leak occurs when you overwrite prices.
    }
}
```

Any code that creates a stream leak is erroneous. Code that repeatedly creates unused, inaccessible streams quickly uses up machine resources. To avoid leaking streams:

- Avoid creating streams you do not intend to use immediately.
- Quit a stream before the variable referring to it goes out of scope.

6 Defining What Happens When Matching Events Are Found

■ Using variables	246
■ Defining actions	250
■ Defining static actions	262
■ Getting the current time	263
■ Generating events	264
■ Handling the any type	269
■ Handling any values of different types with the switch statement	272
■ Assigning values	273
■ Defining conditional logic with the if statement	273
■ Defining conditional logic with the ifpresent statement	274
■ Defining loops	275
■ Exception handling	276
■ Logging and printing	279
■ Sample financial application	283

In a monitor, when the correlator detects a matching event, it triggers the action defined by the listener for that event. This section discusses what you can specify in the triggered actions.

In a query, when a match set is found, it triggers execution of the procedural code block in the `find` statement. A subset of the EPL constructs that are available in a monitor are available in a query. See [“Restrictions in queries” on page 134](#) to understand what is not allowed in a query.

Using variables

EPL supports the use of variables in monitors. Depending on where in the monitor you declare a variable, that variable is global or local:

- **Global.** Variables declared in monitors and not inside actions or events are global variables. Global variables are in monitor scope.
- **Local.** Variables declared inside actions are local variables. Local variables are in action scope.

A variable can be of any of the primitive or reference types that are listed under [“Types” on page 583](#) in the EPL Reference.

Information about variables is presented in the topics below.

See also [“Using action type variables” on page 255](#).

Using global variables

Variables in monitor scope are global variables; you can access a global variable throughout the monitor. You can define global variables anywhere inside a monitor except in actions and event definitions. For example:

```
monitor SimpleShareSearch {  
    // A monitor scope variable to store the stock received:  
    //  
    StockTick newTick;  
}
```

This declares a global variable, `newTick`, that can be used anywhere within the `SimpleShareSearch` monitor including within any of its actions.

The order does not matter. In the following example, `f` is a global variable:

```
monitor Test {  
    action onload() {  
        print getZ().toString();  
    }  
    action getZ() returns integer {  
        return f.z;  
    }  
    Foo f;  
    event Foo{  
        integer z;  
    }  
}
```

If you do not explicitly initialize the value of a global variable, the correlator automatically assigns a value to that global variable. See also [“Default values for types” on page 586](#).

Using local variables

A variable that you declare inside an action is a local variable. You must declare a local variable (specifying its type) and initialize that variable before you can use it.

Although the correlator automatically initializes global variables that were not explicitly assigned a value, the correlator does not do this for local variables. For local variables, you must explicitly assign a value before you can use the variable.

If you try to inject an EPL file that declares a local variable and you have not initialized the value of that local variable before you try to use it, the correlator terminates injection of that file and generates a message such as the following: Local variable 'var2' might not have been initialized. EPL requires explicit assignment of values to local variables as a way of achieving the best performance.

When you declare a variable in an action, you can use that variable only in that action. You can declare a variable anywhere in an action, but you can use it only after you declare it and initialize it.

For example,

```
action anAction(integer a) returns integer {
    integer i;
    integer j;
    i := 10;
    j := a;
    return j + i;
}
```

You can use the local action variables, `i` and `j` in the action, `anAction()`, after you initialize them. The following generates an error:

```
action anAction2(integer a) returns integer {
    i := 10; // error, reference to undeclared variable i
    j := a; // error, reference to undeclared variable j
    integer i;
    integer j;
    i := 2;
    j := 5;
    return j + i;
}
```

Suppose that an action scope variable has the same name as a monitor scope variable. Within that action, after declaration of the action scope variable, any references to the variable resolve to the action scope variable. In other words, a local action variable always hides a global variable of the same name.

Consider again the definition for `anAction2()` in the previous code fragment, but with `i` and `j` variables declared in the monitor scope. The first use of `i` and `j` resolves successfully to the values of the `i` and `j` monitor scope variables. The second use occurs after the local declaration and

initialization of `i` and `j`. That use resolves to the local (within the action) occurrence. This results in the following values:

- Global variable `i` is set to 10.
- Local variable `i` is set to 2.
- Global variable `j` is set to the value of `a`.
- Local variable `j` is set to 5.

Since you must explicitly initialize local variables before you can use them, the following example is invalid because `j` and `i` are not initialized to any value before they are used.

```
action anAction3(integer a) returns integer {
    integer i;
    integer j;
    return j + i; // error, i and j were not initialised
}
```

It is possible to initialize a variable on the same line as its declaration, as follows:

```
action anAction4(integer a) returns integer {
    integer i := 10;
    integer j := a;
    return j + i;
}
```

It is also possible to initialize a local variable by coassigning to it in an event listener. For example, the following is correct:

```
action onload() {
    on all Event() as e {
        log e.toString();
    }
}
```

You can also initialize a local variable by coassigning to it from a stream. For example:

```
action onload() {
    from x in all X() select x.f as f {
        log f.toString();
    }
}
```

Using variables in listener actions

Suppose you use a local variable in a listener action, as in the following example:

```
monitor MyMonitor {

    integer x;

    action onload() {
        integer y := 10;
        on all StockTick(*,*) {
            log x.toString();
        }
    }
}
```



```

        log y.toString();
    }
    y := 5;
}

```

In this example, `x` is a global variable, and `y` is a local variable. There are references to both variables in the listener action.

A reference to a global variable in a listener action is the same as a reference to a global variable anywhere else in the monitor. However, a reference to a local variable in a listener action causes the correlator to retain a copy of the local variable for use when the event listener triggers. The value held by this copy is the value that the local variable has when the correlator instantiates the event listener.

When the event listener triggers the correlator executes the listener action. This will be at some point in the future, and after the rest of the body of the enclosing action has been executed. Since the action has already been executed, any of the original local variables no longer exist. This is why the correlator retains a copy of the local variable to make available to the listener action when it is executed.

In the example above, when the event listener triggers and the correlator executes the listener action

- `x` has a value of 0, which is the value that the correlator automatically assigns
- `y` has a value of 10, which is the value it was set to when the event listener was instantiated

The value of `y` that the correlator retained when it instantiated the event listener is not affected by the subsequent statement (after the `on` statement) that sets the value of `y` to 5.

Note:

For reference types (see also [“Reference types” on page 584](#)), retaining as a copy of the variable really means only retaining as a copy of its reference. Hence, if any code changes the contents of the referenced object(s) between event listener creation and event listener triggering, then this does affect the values used by the triggered event listener.

Specifying named constant values

In a monitor or in an event type definition, you can specify a named boolean, decimal, float, integer, or string value as constant. The format for doing this is as follows:

```
constant type name := literal;
```

Element	Description
<i>type</i>	Specify boolean, decimal, float, integer, or string. This is the type of the constant value.
<i>name</i>	Specify an identifier for the constant. This name must be unique within its scope — monitor, event, or action.

Element	Description
<i>literal</i>	Specify the value of the constant. The type of the value must be the type that you specify for the constant.

Benefits of using constants include:

- Using a named constant can often be better than using a literal because it lets you define that constant in a single place. There is no chance of one instance becoming incorrect when the value is changed elsewhere. An alternative to using a constant would be to define a variable to contain the value. The disadvantage with this approach is that someone could accidentally assign a new value to the "constant", which would cause errors.
- A named constant can make code easier to read because the name can be meaningful in a way that a magic number, such as 42, is not.
- Constants appear in memory once. For example, spawning multiple copies of a monitor that contains a constant does not consume memory to store extra copies of the constant. A non-constant variable takes up space in memory for every copy of the event or monitor in the correlator.

You can refer to a declared constant in any code in the event or monitor being defined. When you define a constant in an event you can refer to it from outside the event by qualifying the name of the constant with the event name, for example, `MyEvent.myConstant`.

Following is an example of specifying and using a constant:

```
event Paper {  
    constant float GOLDEN := 1.61803398874;  
    float width;  
    action getLength() {  
        return GOLDEN * width;  
    }  
    action getWidth() {  
        return width;  
    }  
}
```

You cannot declare a constant in an action.

Defining actions

Actions are similar to procedures.

A monitor can define any number of actions. Finding an event, or pattern of events, of interest can trigger an action.

A query can define any number of actions. If defined, actions must be after the `find` statement. Expressions in the `find` pattern or `find` block can invoke the actions defined in that query.

You can also trigger an action by invoking it from inside another action. You can also declare an action as part of an event type definition, and then call that action on an instance of that event.

The topics below provide information about defining actions.

Format for defining actions

The format for defining an action that takes no parameters and returns no value is as follows:

```
action actionName() {
    // do something
}
```

Optionally, an action can do either one or both of the following:

- Accept parameters
- Return a value

The format for defining an action that accepts parameters and returns a value is as follows:

```
action actionName(type1 param1, type2 param2, ...) returns type3 {
    // do something
    return type3_instance;
}
```

For example:

```
action complexAction(integer i, float f) returns string {
    // do something
    return "Hello";
}
```

An action that accepts input parameters specifies a list of parameter types and corresponding names in parentheses after the action name. Parentheses always follow the action name, in declarations and calls, whether or not there are any parameters. Parameters can be of any valid EPL type. The correlator passes primitive types by value and passes complex types by reference. EPL types and their properties are described in the *API Reference for EPL (ApamaDoc)*.

When an action returns a value, it must specify the `returns` keyword followed by the type of value to be returned. In the body of the action, there must be a `return` statement that specifies a value of the type to be returned. This can be a literal or any variable of the same type as declared in the action definition.

An action can have any name that is not a reserved keyword. Actions with the names `onload()`, `onunload()` and `ondie()` can only appear once and are treated specially as already described in [“About monitor contents” on page 34](#). It is an EPL convention to specify action names with an initial lowercase letter, and a capital for each subsequent word in the action name.

Actions and global variables must not have the same names. See [“Using action type variables” on page 255](#). If you have any code that uses the same identifier for an action and a global variable, you must change it.

Invoking an action from another action

To invoke an action from another action, specify the action name followed by parentheses. If the action takes one or more input parameters, specify values for the parameters inside the parentheses. For example:

```
// First action:
action myAction1() {
    myAction2();
}

// Second action that is called by the first action:
action myAction2() {
    // . . .
}
```

In the example above, `myAction1()` calls `myAction2()` from inside the `myAction1()` declaration block. `myAction2()` takes no parameters and does not return a value.

When an action returns a value, you can invoke that action only from within an expression. You cannot specify a standalone statement that invokes an action that returns a value. Discarding the return value is illegal in EPL. For example:

```
action myAction3() returns string {
    return "Hello";
}

action myAction4() {
    string response;
    response := myAction3(); // Valid
    myAction3();             // Invalid
}
```

Consider this extended example:

```
// First action:
//
action myAction1() {
    myAction2();
}

// Second action that is called by the first action:
//
action myAction2() {
    string answer1, answer2;
    myAction5(5, 10.5);
    on anEvent() myAction5(5, 10.5);
    answer1 := myAction6(256, 1423.2);
    answer2 := myAction7();
}

// Action that is called by myAction2:
//
action myAction5 (integer i, float f) {
    ...
}
```

```
// Another action that is called by myAction2:
//
action myAction6 (integer i, float f) returns string {
    return "Hello";
}

// Yet another action that is called by myAction2:
//
action myAction7() returns string {
    return "Hello again";
}
```

`myAction2()` takes no parameters and does not return a value.

`myAction5()` accepts input parameters. You can invoke it from a standalone statement:

```
myAction5(5, 10.5);
```

You can also invoke it as a listener action:

```
on anEvent() myAction5(5, 10.5);
```

`myAction6()` accepts input parameters and returns a value. You can invoke `myAction6()` only from within an expression:

```
answer1 := myAction6(256, 1423.2);
```

`myAction7()` returns a value but does not take any parameters. You can invoke it only from within an expression:

```
answer2 := myAction7();
```

Specifying actions in event definitions

You can specify an action in an event type definition. This lets you call that action on an instance of the event, just as you would call a built-in method on some other type, such as calling the `toString()` method on the `integer` type.

When you define an action in an event, it behaves almost the same way as an action in a monitor or query. For example, an action in an event can

- Set up event or stream listeners (only in a monitor)
- Call other actions within that event
- Access members of that event

In a monitor, but not in a query, an action in an event has an implicit `self` argument that refers to the event instance that the action was called on. The `self` argument behaves in the same way as the `this` argument in C++ or Java.

Example

For example, consider the following event type definition:

```
event Circle {  
    action area() returns float {  
        return 3.14159 * radius * radius;  
    }  
    action circumference() returns float {  
        return 2.0 * 3.14159 * self.radius;  
    }  
    float radius;  
}
```

The specifications here of `radius` and `self.radius` are equivalent.

You can then write code that looks like this:

```
Circle c := Circle(4.0);  
print "Circle area = " + c.area().toString();  
print "Circle circumference = " + c.circumference().toString();
```

Of course, the output is as follows:

```
Circle area = 50.26544  
Circle circumference = 25.13272
```

Behavior

The correlator never executes actions in events automatically. In an event, if you define an `onload()` action, the correlator does not treat it specially as it does when you define the `onload()` action in a monitor.

When you call an action in an event, the correlator executes the action in the monitor or query instance in which the call was made. In a monitor, if the action sets up any listeners, these listeners are in the context of this monitor instance. If this monitor instance dies, the listeners also die.

You can use plug-ins from within event actions. In the event definition, specify the `import` statement to give the plug-in an alias within the event. Specify the `import` statement in the same way that you specify it for a monitor or query. You use the plug-in alias to call functions on the plug-in in the same way as you use it for a monitor or query.

When you define an event, there are no ordering restrictions for the definition of fields, imports, or actions. You can define them in any order.

Spawning

From an action within an event, you can spawn to an action in the same event. The correlator spawns a monitor instance and executes the specified action on the event instance in the new monitor instance.

Note:

In a query, `spawn` and `spawn...to` statements are not needed and so they are not allowed.

It is not possible to spawn from outside a particular event to an action that is a member of that particular event. Instead, spawn to an action that calls the action that is the event member. For example:

```

event E {
  action spawntotarget() {
    spawn target();                // legal
  }
  action target() {
    log "Spawned "+self.toString();
  }
}

monitor m {
  action onload() {
    E e;
    spawn e.target();              // not legal
    spawn calltarget(e);          // legal
    e.spawntotarget();
  }
  action calltarget(E e) {
    e.target();
  }
}

```

Be sure to follow the `spawn` keyword with an action name identifier. Actions spawned to must have no return value, as before. See also [“Utilities for operating on monitors” on page 57](#).

Restrictions

To summarize, when you define an action in an event, the following restrictions apply:

- If the action contains an `on` statement, you can coassign a matching event only to local variables. You cannot coassign a matching event to the event's fields nor to items outside the event or in the monitor.
- In a monitor, if you declare an instance of an event that has an action member, you cannot specify a call from that action to an action that is defined in the monitor.
- You cannot assign values to the implicit `self` parameter, any more than you can assign to `this` in Java.

Using action type variables

In addition to defining an action, you can define a variable whose type is `action`. This lets you assign an action to an action variable of the same action type. An action is of the same type as an action variable if they have the same argument list (the same types in the same order) and return type (if any).

Defining action variables

The format for defining an action type variable is as follows:

```
action<[type1[, type2]...]>[returns type3]name;
```

Specify the keyword, `action`.

Follow the `action` keyword with zero, one or more parameter types enclosed in angle brackets and separated by commas. The angle brackets are required even when the action takes no arguments.

Optionally, follow the parameter list with a `returns` clause. Specify the `returns` keyword followed by the type of the returned value.

Finally, specify the name of the variable. For example:

```
action<string> a;  
action<integer, integer> returns string b;
```

You can use an action variable anywhere that you can use a sequence or dictionary variable. For example, you can

- Pass an action as a parameter to another action.
- Return an action from execution of an action.
- Store an action in a local variable, global variable, event field, sequence, or dictionary.

You cannot route, emit, enqueue or send an event that contains an action variable field.

You must initialize an action variable before you try to invoke it.

When an action variable is a member of an event the behavior of the action depends on the instance of the event that the action is called on. Consequently, it can be handy to bind an action variable member with a particular event instance. See [“Creating closures” on page 259](#).

Built-in methods are treated exactly the same as user-defined actions. This means you can assign a built-in method to an action variable. For example:

```
action<float> returns string f := float.toString;
```

Invoking action variables

The only operation that you can perform on an action variable is to call it. You do this in the normal way by passing a set of parameters in parentheses after an expression that evaluates to the action variable. For example:

```
monitor Test{  
  integer i;  
  action<string> x; // Uninitialized global action variable.  
  action onload() {  
  
    // Invoke the runMe action. The first argument to runMe is an  
    // action variable for an action having a single argument of  
    // type integer and no return value.  
    // Since the printInteger action conforms to the argument  
    // expected by runMe, you can pass printInteger to runMe.  
    runMe(printInteger, 10);  
  
    // Declare a local action variable, g. This action takes one  
    // integer argument and does not return a result.  
    // The printInteger action conforms to this so  
    // assign printInteger to g.
```



```

    action<integer> g := printInteger;

    // Invoke the runMe action again.
    // Pass g instead of explicitly passing printInteger.
    runMe(g, 20);

    // Declare a local dictionary that contains action variables.
    // Each action variable takes a single integer argument and
    // and does not return a result.
    // Add printInteger to the dictionary.
    // Invoke printInteger and pass 30 as the argument.
    dictionary<string, action<integer>> do := {};

    do["printIt"] := printInteger;
    do["printIt"] (30);

    // Invoke x. Since this global variable was never
    // initialized, the monitor instance terminates.
    x("hello!");
}

action runMe(action<integer> f, integer i) {
    f(i);
}

action printInteger(integer i) {
    print i.toString();
}
}

```

After injection, this monitor prints

```

10
20
30

```

and then terminates upon invocation of `x` because `x` was never initialized.

Calling an uninitialized, local action variable causes an error that prevents the correlator from injecting the monitor. While the correlator injects code that contains an uninitialized, global action variable, trying to call the uninitialized variable causes a runtime error and the monitor instance terminates.

Declaring action variables in event definitions

When you define an action as a member field in an event, that action has an implicit `self` argument as the first argument (see [“Specifying actions in event definitions” on page 253](#)). You must include this implicit argument when determining whether an action definition conforms to an action variable declaration. For example, the following is illegal:

```

event A {
    action foo(float f) returns string {
        return "Hello";
    }
    action bar() {
        action<float> returns string f := A.foo;
    }
}

```

```
}

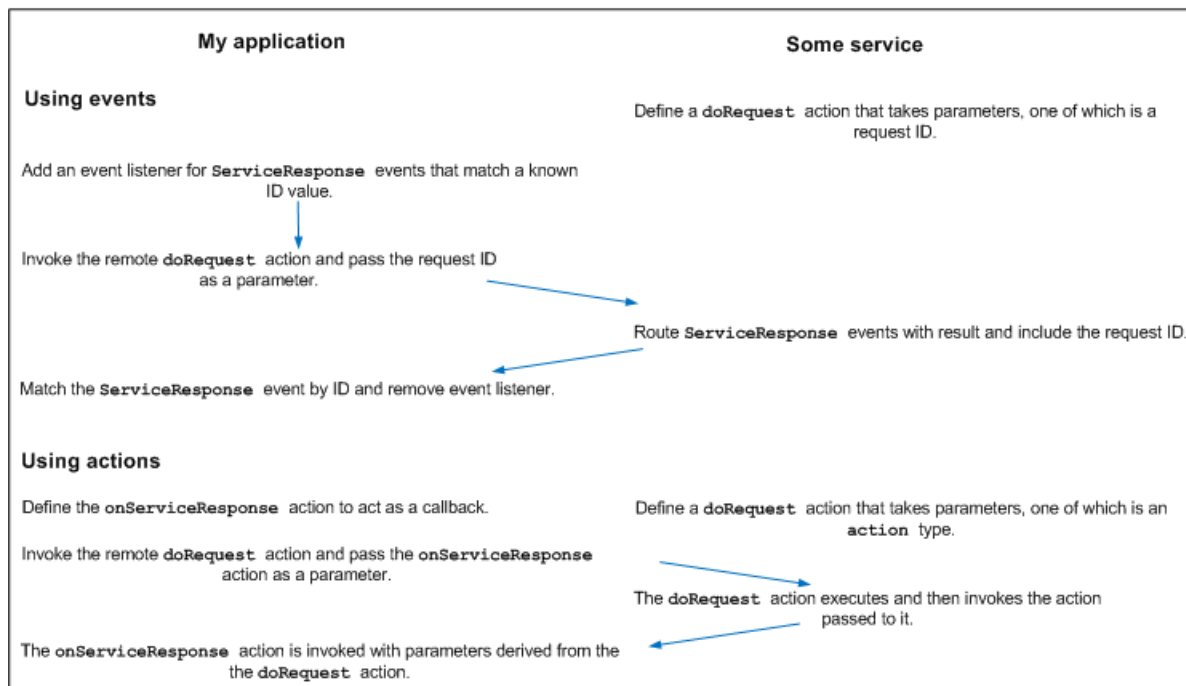
```

In the previous code, you cannot assign the `A.foo` action to `f` because `f` takes a single `float` argument whereas `A.foo` has two arguments — the implicit `A` argument and then the `float` argument. To correct this example, specify `A` as the first action argument in the body of the `bar` action.

```
event A {
  action foo(float f) returns string {
    return "Hello";
  }
  action bar() {
    action<A, float> returns string f := A.foo;
  }
}
```

Actions in place of routed events

In some situations, you might find it more efficient to use action type variables instead of routing events. For example, suppose you implement a service that takes an action variable as one of its parameters. Now suppose that the service needs a response from an adapter or some other service before it can send a response. When ready, the service can respond with a routed event, but that means you have to set up an event listener for that event. Routing events and setting up event listeners is more expensive than invoking actions. So instead of routing and listening, the service can respond by invoking the action on the event that initiated the service request. For example:



The following sample code uses a routed event. Following this code there is a sample that uses an action on an event.

```
event ServiceResponse {
  string requestId;
  ...
}
```

```

event Service {
    action doRequest( string requestId, ... ) {
        ...
        // when asynchronous 'service actions' are complete
        route ServiceResponse( requestId, ... );
    }
    ...
}

monitor Client {
    Service service;
    action onload() {
        ...
        string id := ...;
        on ServiceResponse( requestId=id )as r {
            ...
        }
        service.doRequest( id, ... );
    }
}

```

The following sample code uses an action on a `Client` monitor:

```

event Service {
    action doRequest( action< ... > callback, ... ) {
        ...
        // when asynchronous 'service actions' are complete
        callback( ... );
    }
    ...
}

monitor Client {
    Service service;
    action onload() {
        ...
        string id := ...;
        service.doRequest( onServiceResponse, ... );
    }
    action onServiceResponse(...) {
        ...
    }
}

```

Creating closures

When an action is a member of an event the behavior of the action depends on the instance of the event that the action is called on. Consequently, you might want to bind an action member with a particular event instance. When you bind an action member to an event instance you are creating a closure. The advantages of creating a closure are:

- Simpler syntax for executing the action
- Greater flexibility in making assignments to action variables

Consider the following event definition:

```
event E {
    integer i;
    action foo() { print "Foo "+i.toString(); }
    action times(integer j) returns integer { return i*j; }
}
```

With this definition, `E(1).foo()` would print "Foo 1", while `E(42).foo()` prints "Foo 42". The action `E.foo` always has a specific instance of `E` to work with. You can achieve this by specifying the action's implicit self argument when you call the action, as described earlier in this topic. When you use this technique you identify the event instance when you call the action variable.

Alternatively, you can create a closure that binds an action member with an event instance. You store the closure in an action variable. The action variable and the action member must be of the same action type. That is, they must take the same argument(s), if any, and return the same type, if any.

When you use this technique you identify the event instance when you assign the event's action member to the action variable.

The following code shows an example of binding an event instance to an action member by storing the closure in an action variable.

```
monitor m {
    action <> a;
    action onload() {
        E e := E(42);
        a := e.foo;
        a(); // Prints "Foo 42"
    }
}
```

In this example, `e.foo` denotes `E.foo` called on `e`. That is, when you assign the action `e.foo` to the a action variable you are identifying which instance of `E` to use when you call the `a` action. This closure binds a reference to `E` to the `E.foo` action and stores it in the `a` action variable. After you create a closure, you can call an action on an event as though it is a simple action. This gives you considerable flexibility in what you can assign to an action variable.

More about closures

EPL performs its own garbage collection. Consequently, you do not need to consider how long a bound object must last. This is handled automatically.

A closure binds by reference. Consider the following example, which uses the same event `E` as above:

```
monitor m {
    action <integer> returns integer a;
    action onload() {
        E e := E(3);
        a := e.times;
        print a(2).toString(); // Prints "6"
        e.i := 5;
        print a(2).toString(); // Prints "10"
    }
}
```

In a portion of code, you can define multiple action variables that contain closures for the same object. For example:

```
event Counter {
  integer i;
  action increment() { i := i+1; }
  action output() { print i.toString(); }
}
event Increment {}

event Finish {}

monitor m {
  action <> incrementAction;
  action <> outputAction;
  action onload() {
    Counter counter := new Counter;
    incrementAction := counter.increment;
    outputAction := counter.output;
    on all Increment() and not Finish() { incrementAction(); }
    on all Finish() { outputAction(); }
  }
}
```

In an event type, when an action member refers to another action member in the same event type a closure happens implicitly. For example:

```
event E {
  action <integer> returns integer a;
}

event Plus {
  integer i;
  action f(integer j) returns integer { return i+j; }
  action setA(E e) { e.a := f; }
}
```

Here, the `f` in `e.a := f` is equivalent to `self.f`, just as it would be if `setA` had called `f` instead of assigning it to an action variable. This creates a closure. After `setA` is called on some instance of `Plus`, `e.a` will call `f` on that same instance.

Other ways to specify closures

You can create a closure using any value and any action on that value. Thus, it is possible to:

- Bind a built-in method to a value.
- Bind actions to primitive types and other reference types instead of to events.
- Bind actions to a literal or a function's return value instead of a variable's value.

For example:

```
// Print "E(42)"
E e := E(42);
action <> printE42 := e.toString;
```

```
// Print "Foo 12345"
action <> printFoo12345 := E(12345).foo;

// Take a floating-point number and return e to that power:
action <float> returns float eToTheX := 2.718282.pow;

// Return a random integer from 0 to 9 inclusive.
// (The brackets around 10 are needed so that "10." is not treated as a
// floating-point number.)
action <> returns integer randomDigit := (10).rand;

// Return the strings in a sequence, separated by colons.
action <sequence<string>> returns string j := ":".join;
```

Restrictions

You cannot route, enqueue, emit or send an event that contains an action variable field. It is okay to route, enqueue, emit or send an event that contains an action definition.

An action variable cannot be a key in a dictionary. An event that contains an action field cannot be a key in a dictionary.

JMon

In a JMon application, you cannot declare event types that have action type members. Consequently, events that contain action type fields are invisible to JMon applications.

Defining static actions

In contrast to the regular actions that are described in [“Defining actions” on page 250](#), static actions do not apply to specific instances of an event. They are not as powerful as regular actions, but they are helpful in situations where it makes more sense to use an action that is related to the event type, and where the action is not called on an instance of that event.

Static actions can only be declared inside an event type. They are defined in just the same way as regular actions (see [“Format for defining actions” on page 251](#)). The only differences are that they start with `static`, cannot reference `self`, and cannot reference members of the event. For example:

```
static action staticActionName() {
    // do something
}
```

The following example contrasts a regular action with a static action.

```
event MyEventType {
    integer i;
    action someAction() {
        print i.toString(); // Valid
    }
    static action someStaticAction() {
        print i.toString(); // Not valid
        someAction();       // Not valid
    }
}
```

```
MyEventType e := new MyEventType;
e.someAction();
MyEventType.someStaticAction();
```

A static action can be used, for example, if you have a factory action which constructs a new instance of a particular event type and initializes its members to values that make sense for that type. Although it is possible to have such code in any place, in terms of program readability, it is more helpful to "associate" the static action with the event type that it is creating. For example:

```
event MyEventType {
    string s;
    integer i;
    static action initialise() returns MyEventType {
        MyEventType ret := new MyEventType;
        ret.s := "Default";
        ret.i := 100;
        return ret;
    }
    ...
}
MyEventType e := MyEventType.initialise();
```

With the above definition, the static action can then be called using a single line of code anywhere in your program.

Getting the current time

In the correlator, the current time is the time indicated by the most recent clock tick. However, there are some exceptions to this:

- If you specify the `-xclock` option when you start the correlator, the correlator does not generate clock ticks. Instead, you must send time events (`&TIME`) to the correlator. The current time is the time indicated by the most recent received, externally generated, time event. See [“Externally generating events that keep time \(&TIME events\)” on page 183](#).
- If you have multiple contexts, it is possible for the current time to be different in different contexts. A particular context might be doing so much processing that it cannot keep up with the time ticks on its queue. In other words, if contexts are mostly idle, then they would all have the same current time.
- When the correlator fires a timer, the current time in the context that contains the timer is the timer's trigger time. See [“About timers and their trigger times” on page 180](#).

The information in the remainder of this topic assumes that the current time is the time indicated by the most recent clock tick.

Use the `currentTime` variable to obtain the current time, which is represented as seconds since the epoch, January 1st, 1970 in UTC. The `currentTime` variable is similar to a global read-only constant of type `float`. However, the value of the `currentTime` variable is always changing to reflect the correlator's current time.

In the correlator, the current time is never the same as the current system time. In most circumstances it is a few milliseconds behind the system time. This difference increases when the input queues of public contexts grow.

When a listener executes an action, it executes the entire action before the correlator starts to process another event. Consequently, while the listener is executing an action, time and the value of the `currentTime` variable do not change. Consider the following code snippet,

```
float a;
action checkTime() {
    a := currentTime;
}

// ... Lots of additional code
// A listener calls the following action some time later
action logTime() {
    log a.toString(); // The time when checkTime was called
    log currentTime.toString(); // The time now
}
```

In this code, an event listener sets float variable `a` to the value of `currentTime`, which is the time indicated by the most recent clock tick. Some time later, a different event listener logs the value of `a` and the value of `currentTime`. The values logged might not be the same. This is because the first use of `currentTime` might return a value that is different from the second use of `currentTime`. If the two event listeners have processed the same event, the logged values are the same. If the two event listeners have processed different events, the logged values are different.

Generating events

As discussed previously, actions can perform calculations and log messages. In addition, actions can dynamically generate events. The topics below discuss this.

Generating events with the route statement

The `route` statement generates a new event that goes to the front of the input queue of the current context.

Any active listeners seeking that event then receive it. There is only one difference between an externally sourced event (passed in through a live message feed) and an event that was generated internally through a `route` statement. The difference is that internally routed events are placed at the front of the context's input queue in the same order as they are routed within an action, and after any previously internally routed events where multiple listener actions have been triggered by an event. The correlator processes the routed events on the input queue before it processes the next non-routed event on the input queue. See [“Event processing order for monitors” on page 46](#).

For example:

```
action simulateCrash() {
    route StockTick(currentStock.name, 50.0);
    route StockTick(currentStock.name, 30.0);
    route StockTick(currentStock.name, 20.0);
    route StockTick(currentStock.name, 10.0);
}
```



```

route StockTick(currentStock.name, 5.0);
route StockTick(currentStock.name, 1.0);
}

```

The `simulateCrash()` action shown above routes six `StockTick` events for the monitor's specific stock name, with drastically reducing prices. Other monitors (or the same monitor) may receive these events and process them accordingly.

The `route` statement can operate on any values as well as events, provided that the any value is of a routable event type.

You cannot route an event if the event (or one of its members) contains a field of an unroutable type (`action`, `chunk`, `listener`, `stream`). There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type `action`, `chunk`, `listener`, or `stream`.

Note that you can route an event whose type is defined in a monitor.

Generating events with the `send` statement

The `send` statement sends an event to a channel, a context, a sequence of contexts, or a `com.apama.Channel` object.

When you send an event to a channel, the correlator delivers it to all contexts and external receivers that are subscribed to that channel. To send an event, use the following format:

```
send event_expression to expression;
```

The result type of `event_expression` must be an event. It cannot be a string representation of an event. The `send` statement can operate on any values as well as events, provided that the any value is of a routable event type.

To send an event to a channel, the `expression` must resolve to a string or a `com.apama.Channel` object that contains a string. If there are no contexts and no external receivers that are subscribed to the specified channel, then the event is discarded. See [“Subscribing to channels” on page 54](#).

The only exception to this is the default channel, which is the empty string. Events sent to the default channel go to all public contexts. All running Apama queries receive events sent on the default channel as well as events sent on the `com.apama.queries` channel. See [“Defining Queries” on page 59](#).

To send an event to a context, the `expression` must resolve to a context, a sequence of contexts, or a `com.apama.Channel` object that contains a context. You must create a context before you send an event to the context. You cannot send an event to a context that you have declared but not created. For example, the following code causes the correlator to terminate the monitor instance:

```

monitor m {
    context c;
    action onload()
    {
        send A() to c;
    }
}

```

If you send an event to a sequence of contexts and one of the contexts has not been created first, then the correlator terminates the monitor instance. Sending an event to a sequence of contexts is non-deterministic. You cannot send an event to a sequence of `com.apama.Channel` objects. For details, see [“Sending an event to a sequence of contexts” on page 295](#).

All routable event types can be sent to contexts, including event types defined in monitors. There is a runtime check if the event (or one of its members) can contain an `any` field; an exception is thrown if the `any` field contains an object of type `action`, `chunk`, `listener`, or `stream`.

If a correlator is configured to connect to Universal Messaging, then a channel might have a corresponding Universal Messaging channel. If there is a corresponding Universal Messaging channel, then Universal Messaging is used to send the event to that Universal Messaging channel.

See "Choosing when to use Universal Messaging channels and when to use Apama channels" in *Connecting Apama Applications to External Components*.

Sending events to `com.apama.Channel` objects

A `com.apama.Channel` object is particularly useful when writing services that can be used in both distributed and local systems. For example, by using a `Channel` object to represent the source of a request, you could write a service monitor so that the same code sends a response to a service request. You would not need to have code for sending responses to channels and separate code for sending responses to contexts.

Consider the following `Request` event and `Service` monitor definitions:

```
event Request {
    ...
    Channel source;
}

monitor Service {
    action onload() {
        monitor.subscribe("Requests");
        on all Request() as req {
            Response rep := Response(...);
            send rep to req.source;
        }
    }
}
```

EPL code in a context in the same correlator as the `Service` monitor could send a `Request` event with the `source` field set to `context.current()` and would receive the `Response` event that the `Service` monitor sends. For example:

```
monitor LocalRequester {
    action onload() {
        Request req := Request(...);
        req.source := Channel(context.current());
        send req to "Requests";

        on all Response() as rep {
            ...
        }
    }
}
```

```
}
```

Now consider a monitor that is in a correlator that is connected to the Service monitor host correlator. For example, the correlators can be connected by means of `engine_connect`. The remote monitor could send a `Request` event with the `source` field set to a `Channel` object that contains the name of a channel that the remote monitor is subscribed to. For example:

```
monitor RemoteRequester {
  action onload() {
    monitor.subscribe("Responses");
    Request req := new Request;
    req.source := Channel("Responses");
    send req to "Requests";
    on all Response() as rep {
      //...
    }
  }
}
```

In this example, if the correlators are connected by means of `engine_connect` then the connections would need to be subscribed to the `Requests` channel and the `Responses` channel. As you can see, the service monitor does not require different code according to whether the request is coming from a local or remote context. The service monitor simply sends the response back to the source and it does not matter whether the source is a context or a channel.

You can send a `Channel` object from one Apama component to another Apama component only when the `Channel` object contains a string. You cannot send a `Channel` object outside a correlator when it contains a context.

Enqueuing to contexts

To enqueue an event to a particular context, use the `enqueue...to` statement:

```
enqueue event_expression to context_expression;
```

Note:

The `enqueue...to` statement is superseded by the `send...to` statement. The `enqueue...to` statement will be deprecated in a future release. Use the `send...to` statement instead. See [“Generating events with the send statement” on page 265](#).

The result type of `event_expression` must be an event. It cannot be a string representation of an event. The result type of `context_expression` must be a context or a variable of type `context`. It cannot be a `com.apama.Channel` object that contains a context. The `enqueue...to` statement can operate on any values as well as events, provided that the any value is of a routable event type.

The `enqueue...to` statement sends the event to the context's input queue. Even if you have a single context, a call to `enqueue x to context.current()` is meaningful and useful.

You must create the context before you enqueue an event to the context. You cannot enqueue an event to a context that you have declared but not created. For example, the following code causes the correlator to terminate the monitor instance:

```
monitor m {
  context c;
```

```
action onload()
{
    enqueue A() to c;
}
}
```

If you enqueue an event to a sequence of contexts and one of the contexts has not been created first then the correlator terminates the monitor instance. For details, see [“Sending an event to a particular context” on page 294](#).

[“Sending an event to a sequence of contexts” on page 295](#) is non-deterministic.

All routable event types can be enqueued to contexts, including event types defined in monitors. There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type action, chunk, listener, or stream.

Generating events to emit to outside receivers

The `emit` statement dispatches events to external registered event receivers, which means that the events leave the correlator. Active listeners do not receive emitted events.

Note:

The `emit` statement is superseded by the `send` statement. See [“Generating events with the send statement” on page 265](#). The `emit` statement will be deprecated in a future release. Use `send` rather than `emit`.

There are two formats available for using `emit`. You can directly emit an event, as the example below does first, or else place the event in a string and emit that. If you use this latter format, you must ensure that you define the string to represent a valid event. The correlator does not check whether the string you specify represents an event that is compliant with any event type that has been injected. In fact, you can use this mechanism to emit an event of a type that has not been defined in EPL anywhere else.

For example, consider a revised version of an earlier example. The result, instead of being printed as a message on the screen, is now being sent out as an event message:

```
event StockTickPriceChange {
    string owner;
    string name;
    float price;
}

// A new processTicks action that dispatches an output event
// to external applications instead of logging
action processTicks() {

    // The following emit format sends the event itself.
    emit StockTickPriceChange(currentStock.owner,
        newTick.name, newTick.price) to
        "com.apamax.pricechanges";

    // Or, use the following emit format, which sends a string that
    // contains the event.
    emit "StockTickPriceChange(\""+currentStock.owner+
```

```
"\", \"\"+newTick.name+\"\", \"\"+newTick.price.toString()+\"\" to  
"com.apamax.pricechanges";
```

Events are emitted onto named channels. In the above code the `StockTickPriceChange` event is being published on the `com.apamax.pricechanges` channel. For an application to receive events from Apama it must register itself as an event receiver and subscribe to one or more channels. Then if events are emitted to those channels they will be forwarded to it.

Channels effectively allow both point-to-point message delivery as well as through publish-subscribe. As in the above example, channels can be set up to represent topics. External applications can then subscribe to event messages of the relevant topics. Otherwise a channel can be set up purely to indicate a destination and have only one application connected to it.

The `emit` statement can operate on any values as well as events, provided that the any value is of a routable event type.

You cannot emit the following events:

- An event whose type is defined inside a monitor.
- An unroutable event type. There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type `action`, `chunk`, `listener`, or `stream`.

If a correlator is configured to connect to Universal Messaging, then a channel might have a corresponding Universal Messaging channel. If there is a corresponding Universal Messaging channel, then Universal Messaging is used to emit the event to that Universal Messaging channel.

See "Choosing when to use Universal Messaging channels and when to use Apama channels" in *Connecting Apama Applications to External Components*.

Handling the any type

EPL supports an any type that can hold a value of a concrete EPL type (that is, a type other than the any type). See the *API Reference for EPL (ApamaDoc)* for full details of the any type.

The `switch` statement is the preferred way of handling any values unless the type is known or not important. See "[Handling any values of different types with the switch statement](#)" on page 272 for details on the `switch` statement.

An any value may be empty and not contain a value, or it can contain a value which has a type associated with it. The type of the value can be obtained using the `getTypeName()` method on the any type.

A variable of a concrete type can be used where an any value is expected in:

- assignments and initialization, for example:

```
any anyVariable := "string value";
```

- return values, for example:

```
action a() returns any { return new sequence<integer>; }
```

- passing a parameter to an action, for example:

```
actionWithAnyParameter("string value");
```

- the index of a dictionary with an any key type.

In these cases, the concrete type is automatically converted to the any type. This is always safe and valid, and will not throw an exception.

Reflection on types

Reflection allows EPL to act on values of any type in a generic way, including altering the behavior to adapt to what fields or actions a type has. This can be values passed as an any parameter value to some common code, matching an any() listener (see [“Listening for events of all types” on page 152](#)), or created via the any.newInstance method.

Fields or entries from an any value can be accessed using the following methods:

- setEntry(any key, any value)
- getEntry(any key) returns any
- getEntries() returns sequence<any>

For event types, the *key* should be a string containing the field name. For sequences, *key* is the index and should have an integer value.

The actions (including methods) and constants can be obtained with the following methods of the any type:

- getActionNames() returns sequence<string>
- getAction(string name) returns any
- getConstant(string) returns any
- getConstantNames() returns sequence<string>

Actions may be cast to the correct action type and then invoked directly.

For actions, a list of the action's parameter names, a dictionary mapping from the parameter name to the parameter type, and the name of the return type can be obtained with the following methods of the any type:

- getActionParameterNames() returns sequence<string>
- getActionParameters() returns dictionary<string,string>
- getActionReturnTypeName() returns string

For actions, it is also possible to use a “generic” form to call the action via the following method of the any type, even if the signature type is not known at compile time:

- getGenericAction() returns action<sequence<any>> returns any

A sequence<any> of the parameter values of the correct count and types must be supplied.

For detailed information on the above methods, see the any type in the *API Reference for EPL (ApamaDoc)*.

Cast operations

EPL supports casting of the any type to a concrete target type and vice versa.

■ Casting to the any type

Casting a concrete type is allowed to any only. The cast is redundant in this case. Example:

```
integer i := 10;

any a := <any> i;    //redundant cast
any a2 := i;        //valid
```

Example of a cast that is not redundant:

```
sequence<any> entries := (<any> evt).getEntries();
```

■ Casting to a concrete type

```
targetType tgtValue := <targetType> anyValue;
```

Casting an any type with an empty value throws Exception with type set to CastException. See also the Exception type in the *API Reference for EPL (ApamaDoc)*.

If the anyValue does not contain an object of targetType, it throws Exception with type set to CastException, and with the message mentioning the actual type contained by anyValue and targetType.

Examples:

```
any a := 10;

integer i := <integer> a; // Valid

// Will inject but throws a CastException during runtime.
string s := <string> a ;
```

■ Casting to the optional type

```
optional<targetType> opt := <optional<targetType>> anyValue ;
```

Casting to optional<targetType> will never throw. If the any value cannot be converted, then an empty optional<targetType> is returned instead.

If the anyValue is empty, the cast returns an empty optional<targetType>.

If anyValue contains object of optional<targetType> type, the cast returns that object of type optional<targetType>.

If the anyValue contains an object of targetType, the cast returns object of type optional<targetType> containing targetType.

If the `anyValue` does not contain an object of `targetType` or `optional<targetType>`, the cast returns an empty `optional<targetType>`.

Examples:

```
any a := 10;

// Returns optional <integer> containing the value 10.
optional<integer> opti := <optional<integer>> a;

// Returns an empty optional <string>.
optional<string> opts := <optional<string>> a;
```

Handling any values of different types with the switch statement

The `switch` statement is used to conditionally execute a block of code. Unlike the `if` and `if ... else` statements, the `switch` statement can have a number of possible execution paths.

The `switch` statement operates on an expression of the `any` type (see also [“Handling the any type” on page 269](#)). At runtime, the type of the value is examined, and the case clause for that type is executed if there is one, otherwise the default clause is executed. If the `any` value is empty, the default clause is always executed.

If the default clause is not present and none of the case clauses match the type of the value passed to the `switch` statement, then the `switch` statement throws `Exception` with type set to `UnmatchedTypeException`. See also the description of the `Exception` type in the *API Reference for EPL (ApamaDoc)*.

The `switch` statement names the expression as an identifier with the `as` keyword followed by an identifier to name the value. In each case clause block, the identifier has the same type as the case clause.

If the expression is a simple identifier (that is, it is referring to a variable or parameter), then the `as Identifier` part can be omitted. The new local retains the same name.

The following code example shows the usage of the `switch` statement in an action which returns a string, where the case clauses use `return` statements to return from the action. The identifier value in each clause has the same type as the case clause.

Example:

```
action getStringOrBlank(any value) returns string {
    switch(value) {

        // value will be of type float in this block
        case float : { return "float "+ value.toString(); }

        // value will be of type string in this block
        case string: { return value; }

        // value will be of type decimal in this block
        case decimal: { return "decimal "+value.toString(); }
```



```

// value will be of type integer in this block
case integer: { return "integer: " + value.toString(); }

// value will be of type sequence<string> in this block
case sequence<string> : { return " ".join(value); }

// value will be of any type in this block
default: { return ""; }

}
}

```

See also [“The switch statement” on page 655](#).

Assigning values

Valid examples of an assignment statement are:

```

integerVariable := 5;
floatVariable := 6.0;
stringVariable := "ACME";
stringVariable2 := stringVariable;

```

Assignments are only valid if the type of the literal or variable on the right hand side corresponds to the type of the variable on the left hand side, or can be implicitly converted. Implicit conversions are allowed when assigning to an any type, or to an optional (provided the contained type of the optional matches the value being assigned).

When doing an assignment from a variable to another variable, the behavior of EPL depends on the type of the variable.

- In the case of primitive types, the variable on the left hand side is set to the same value as the variable on the right hand side. The value is therefore copied and the two variables remain distinct.
- In the case of complex reference types, the variable on the left hand side is set to reference the same object as the variable on the right hand side. Only the reference is copied, while the underlying object remains the same. If the object is subsequently changed, both variables would reflect the change.
- In the case of the any type, setting to a primitive type creates a new object automatically to hold the primitive value. This is transparent to EPL, but is significantly more expensive than a simple primitive to primitive assignment.

Defining conditional logic with the if statement

EPL supports conditional `if` and `if ... else` statements.

An `if` statement is followed by a boolean expression followed by an optional `then` keyword followed by a block. A block consists of one or more statements enclosed in curly braces, `{ }`. If the boolean expression is `true`, the contents of the block are executed.

The boolean expression must evaluate to the boolean values `true` or `false`.

The `if` statement can be optionally followed by an `else` keyword and a second block. This second block is executed if the boolean expression is `false`. Instead of the `else` block, a single `if` statement, not enclosed in braces, may be used.

EPL example:

```
if floatVariable > 5.0 {
    integerVariable := 1;
} else if floatVariable < -5.0 {
    integerVariable := -1;
} else {
    integerVariable := 0;
}
```

Defining conditional logic with the `ifpresent` statement

The `ifpresent` statement is used to check if one or more values are empty (that is, whether they have a value or not). It unpacks the values into new local variables and conditionally executes a block of code.

The `ifpresent` statement is followed by one or more expressions with an optional name of a target variable followed by a block of code. If each expression is not empty, then the value of the expression is placed in a new local variable whose name is supplied after the keyword `as`. If the expression is a simple identifier (that is, it is referring to a variable or parameter), then the `as identifier` part can be omitted; a new local variable (which shadows the original variable) is created with the same name. Multiple expressions can be used in a single `ifpresent` statement, separated by commas.

If all the expressions have non-empty values, then the first block is executed. A block consists of one or more statements enclosed in curly braces, `{ }`. The new local variables are only available in the first block of the `ifpresent` statement, where they are guaranteed to have non-empty values.

`ifpresent` can be optionally followed by an `else` keyword and block, which is executed if any of the supplied expressions do not have a value.

For optional types, the new local variable is of the unpacked type (the contained type of the optional), for example:

```
optional<integer> possibleNumber := 42;
ifpresent possibleNumber {
    // in this block, possibleNumber is of type integer,
    // so we can perform arithmetic on it:
    nextNumber := possibleNumber + 1;
} else {
    // in practice, won't be executed as possibleNumber
    // has been initialized with a value.
}
```

Thus, `ifpresent` is the recommended way of handling optional variable types. Usually, there is no need to call the `getOrThrow` method of the optional type. `ifpresent` combines the check of whether the value is empty with extracting the value and control flow, and thus reduces the amount of code that could throw an exception.

As an alternative to the `ifpresent` statement, you can use the `getOr` method of the `optional` type, which is less verbose than using `ifpresent`. This is helpful if you want to treat a missing (empty) value as having a value. For example, `possibleNumber.getOr(0)` will give you the number in `possibleNumber`, or `0` if it is empty.

`ifpresent` operates on expressions of the following types:

- `optional`
- `chunk`
- `stream`
- `listener`
- `context`
- `action`
- `any`

See the *API Reference for EPL (ApamaDoc)* for more information on these types.

See also [“The `ifpresent` statement” on page 653](#).

Defining loops

EPL supports two loop structures, `while` and `for`.

An EPL example for `while` is:

```
integerVariable := 20;
while integerVariable > 10 {
    integerVariable := integerVariable - 1;
    on StockTick("ACME", integerVariable) doAction();
}
```

The `for` looping structure allows looping over the contents of a sequence. The counter must be an assignable variable of the same type as the type of elements of the sequence. For example:

```
sequence<integer> s;
integer i;
s.append(0);
s.append(1);
s.append(2);
s.append(3);
for i in s {
    print i.toString();
}
```

The loop will iterate through all the indices in the sequence, checking whether there are any more indices to cover each time. In the example above, `i` will be set to `s[0]`, then `s[1]`, and so on up to `s[3]`. The counter continues incrementing by one each time, and is checked to verify whether it is less than `s.size()` before a further iteration is carried out. Looping only terminates when the next

index would be beyond the last element of the sequence, or equal to `size()` (since indices are counted from 0).

When the correlator executes a `for` loop, it operates on a reference to the sequence. Consequently, if the code in the `for` loop assigns some other sequence to the sequence expression specified in the `for` statement this has no effect on the iteration. However, if the code in the `for` loop changes the contents of the sequence specified in the `for` statement, this can affect the iteration. For example:

```
sequence <string> tmp := ["X", "Y", "Z"];
sequence <string> seq := ["A", "B", "C", "D", "E"];
string s;
for s in seq {
    seq := tmp;
    print s;
}
```

The `for` loop steps through whatever `seq` referred to when the loop began. Therefore, assigning `tmp` to `seq` inside the loop does not affect the behavior of the loop. This code prints A, B, C, D, and E on separate lines.

In the following example, the code in the `for` loop changes the contents of the sequence specified in the `for` statement and this affects the behavior of the loop.

```
sequence<string> seq := ["A", "B", "C", "D", "E"];
string s;
for s in seq {
    seq[2] := "c";
    print s;
}
```

This code prints A, B, c, D, and E on separate lines.

In the following code, the changes to the contents of the specified sequence would prevent the `for` loop from terminating.

```
sequence<string> seq := ["x"];
string s;
for s in seq {
    seq.append(s);
}
```

EPL provides the following statements for manipulating `while` and `for` loops. Usage is intuitive and as per other programming language conventions:

- `break` exits the innermost loop. You can use a `break` statement only inside a loop.
- `continue` moves to the next iteration of the innermost loop. You can use a `continue` statement only inside a loop.
- `return` terminates both the loop and the action that contains it.

Exception handling

EPL supports the `try ... catch` exception handling structure. The statements in each block must be enclosed in curly braces. For example:

```

using com.apama.exceptions.Exception;
...
action getExchangeRate(
    dictionary<string, string> prices, string fxPair) returns float {
    try {
        return float.parse(prices[fxPair]);
    } catch(Exception e) {
        return 1.0;
    }
}

```

Exceptions are a mechanism for handling runtime errors. Exceptions can be caused by any of the following, though this is not an exhaustive list:

- Invalid operations such as trying to divide an integer by zero, or trying to access a non-existent entry in a dictionary or sequence
- Methods that fail, for example trying to parse an object that cannot be parsed
- Plug-ins
- Operations that are illegal in certain states, such as `spawn-to` in an `ondie()` or `onunload()` action, or sending an event to a context and specifying a variable that has not been assigned a valid context object
- The `throw` statement. See [“The throw statement” on page 651](#) for more information.

An exception that occurs in `try block1` causes execution of `catch block2`. An exception in `try block1` can be caused by:

- Code explicitly in `try block1`
- A method or action called by code in `try block1`
- A method or action called by a method or action called by code in `try block1`, and so on.

Note that the `die` statement always terminates the monitor, regardless of `try ... catch` statements.

The variable specified in the `catch` clause must be of the type `com.apama.exceptions.Exception`. Typically, you specify `using com.apama.exceptions.Exception` to simplify specification of exception variables in your code. The `Exception` variable describes the exception that occurred.

The `com.apama.exceptions` namespace also contains the `StackTraceElement` built-in type. The `Exception` and `StackTraceElement` types are always available; you do not need to inject them and you cannot delete them with the `engine_delete` utility.

An `Exception` type has methods for accessing:

- A message — Human-readable description of the error, which is typically useful for logging.
- A type — Name of the category of the exception, which is useful for comparing to known types to distinguish the type of exception thrown. Internally generated exceptions have types such as `ArithmeticException` and `ParseException`. For a list of exception types, see the description of the `Exception` type in the *API Reference for EPL (ApamaDoc)*.

- A stack trace — A sequence of `StackTraceElement` objects that describe where the exception was thrown. The first `StackTraceElement` points to the place in the code that immediately caused the exception, for example, an attempt to divide by zero or access a dictionary key that does not exist. The second `StackTraceElement` points to the place in the code that called the action that contains the immediate cause. The third `StackTraceElement` element points to the code that called that action, and so on. Each `StackTraceElement` object has methods for accessing:
 - The name of the file that contains the relevant code
 - The line number of the relevant code
 - The name of the enclosing action
 - The name of the enclosing event, monitor or aggregate function

Information in an `Exception` object is available by calling these built-in methods:

- `Exception.getMessage()`
- `Exception.getType()`
- `Exception.getStackTrace()`
- `StackTraceElement.getFilename()`
- `StackTraceElement.getLineNumber()`
- `StackTraceElement.getActionName()`
- `StackTraceElement.getTypeName()`

In the catch block, you can specify corrective steps, such as returning a default value or logging an error. By default, execution continues after the catch block. However, you can specify the catch block so that it returns, dies or causes an exception.

You can nest `try ... catch` statements in a single action. For example:

```
action NestedTryCatch() {
    try {
        print "outer";
        try {
            print "inner";
            integer i:=0/0;
        } catch(Exception e) {
            // inner catch
        }
    } catch(Exception e) {
        // outer catch
    }
}
```

The block in a `try` clause can specify multiple actions and each one can contain a `try ... catch` statement or nested `try ... catch` statements. An exception is caught by the innermost enclosing `try ... catch` statement, either in the action where the exception occurs, or walking up the call stack. If an exception occurs and there is no enclosing `try ... catch` statement then the correlator logs the stack trace of the exception and terminates the monitor instance.

See [“About executing `ondie\(\)` actions” on page 44](#) for information about how `ondie()` can optionally receive exception information if an instance dies due to an uncaught exception.

Logging and printing

The following operations are provided for debugging and textual output:

- `print string`
- `log string [at identifier]`

The `print` statement outputs its text to standard output, which is normally the active display or some file where such output has been piped. See also [“Strings in print and log statements” on page 282](#).

The `log` statement sends the specified string to a particular log file depending on the applicable log level. For details, see “Setting EPL log files and log levels dynamically” in *Deploying and Managing Apama Applications*.

The topics below provide information for using the `log` statement.

Specifying log statements

The format of a `log` statement is as follows:

```
log string [at identifier]
```

Syntax description

Syntax Element	Description
<i>string</i>	Specify an expression that evaluates to a string.
<i>identifier</i>	Optionally, specify the desired log level. Specify one of the following values: CRIT, FATAL, ERROR, WARN, INFO, DEBUG or TRACE. If you do not specify an identifier, the default is INFO.

It is recommended that you do not use the FATAL or CRIT log levels, which are present only for historical reasons. It is better to use ERROR for all error conditions regardless of how fatal they are, and INFO for informational messages.

For each encountered `log` statement, the correlator compares the specified identifier with the applicable log level to determine whether to send the specified string to a log file. If the string is to be sent to a log file, the correlator determines the appropriate log file to send it to.

The correlator uses the tree structure of EPL code to identify the applicable log level and the appropriate log file. See “Setting EPL log files and log levels dynamically” in *Deploying and Managing Apama Applications*.

Log levels determine results of log statements

The correlator supports the following log levels:

0	OFF	No entries go to log files.
1	CRIT	Least amount of entries go to log files.
2	FATAL	
3	ERROR	
4	WARN	
5	INFO	
6	DEBUG	
7	TRACE	Greatest amount of entries go to log files.

You use log levels to filter out log strings. If the log level in effect is lower than the log level in the log statement the correlator does not send the string to the log file. For example, if the log level in effect is `ERROR` (3) and the log level in the log statement is `DEBUG` (6) the correlator does not send the string to the log file since the log level in effect is lower than the log level in the log statement.

Suppose that a string expression in a log statement executes an action or has side effects. In this situation, the correlator executes the log statement so that side effects always take place. However, if the log level in effect is lower than the log level in the log statement the correlator still does not send the string to the log file.

Here are some examples where the log level in effect is `WARN`:

```
log "foo bar" at CRIT; // Sends "foo bar" to the log file.
log "foo bar" at INFO; // Does not send anything to the log file.

log "foo" + "bar" + 12345.toString() at INFO;
// Does not send anything to the log file.
// The expression in the log statement is not evaluated as
// the log level is too low to send output to the log file,
// and the expression does not have side effects.

log "foo" + bar() + 12345.toString() at INFO;
// Does not send anything to the log file.
// Calls bar() since that action might have side effects,
// for example, the action could send an event.
```

Actions on events or monitors are assumed to have side effects. The `com.apama.epl.SideEffectFree` annotation (see [“Adding predefined annotations” on page 52](#)) can be added to an action definition to mark it as side effect free. Note that with this annotation, actions will only be called from log statements if the log statement would write to the log file. This is more compact than checking the log level before executing the log statement. If the action does in fact have side effects, then changing the log level can change the behavior of your program. It is recommended to only add the `SideEffectFree` annotation on an action if a profile shows that a lot of time is spent in calling that

action (premature optimizations add to program complexity for no benefit). Actions called via an action variable are always assumed to have side effects, as the EPL runtime does not know which action is invoked.

For more information on the profile, see "Profiling EPL Applications" in *Using Apama with Software AG Designer*.

To determine the log level in effect, the correlator checks whether you set a log level for the following in the order specified below:

1. The monitor or event that contains the `log` statement.
2. A parent of the monitor or event that contains the `log` statement. The correlator starts with the immediate parent and works its way up the tree as needed.
3. The correlator.

The log level in effect is the first log level that the correlator finds in the tree structure. See "Setting EPL log files and log levels dynamically" in *Deploying and Managing Apama Applications*. If the correlator does not find a log level, the correlator uses the correlator's log level. If you did not explicitly set the correlator's log level, the default is `INFO`.

After the correlator identifies the applicable log level, the log level itself determines whether the correlator sends the `log` statement output to the appropriate log file as follows:

Log level in effect	For log statements with these identifiers, the correlator sends the log statement output to the appropriate log file	For log statements with these identifiers, the correlator ignores log statement output
OFF	None	CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE
CRIT	CRIT	FATAL, ERROR, WARN, INFO, DEBUG, TRACE
FATAL	CRIT, FATAL	ERROR, WARN, INFO, DEBUG, TRACE
ERROR	CRIT, FATAL, ERROR	WARN, INFO, DEBUG, TRACE
WARN	CRIT, FATAL, ERROR, WARN	INFO, DEBUG, TRACE
INFO	CRIT, FATAL, ERROR, WARN, INFO	DEBUG, TRACE
DEBUG	CRIT, FATAL, ERROR, WARN, INFO, DEBUG	TRACE
TRACE	CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE	None

An advantage of this framework is that there is no performance penalty for having `log` statements that do not specify actions in your application. You control the overhead of executing such `log` statements by specifying the appropriate log level.

Where do log entries go?

When the correlator needs to send the `log` statement output to a log file, the correlator checks whether you set a log file for the following in the order specified below:

1. The monitor or event that contains the `log` statement.
2. A parent of the monitor or event that contains the `log` statement. The correlator starts with the immediate parent and works its way up the tree as needed.
3. The correlator.

The log file that receives the `log` statement output is the first log file that the correlator finds. If the correlator does not find a log file, the default is that the correlator sends the string and identifier to `stdout`.

Examples of using log statements

Suppose you insert `DEBUG` `log` statements without actions in a monitor. You specify `ERROR` as the log level for that monitor. The correlator ignores `log` statement output of `log` statements with identifiers of `INFO` or `DEBUG`. But then there are some problems. You use the `engine_management` correlator utility to change the log level to `DEBUG`. Now the correlator sends output from all `log` statements to the appropriate log file.

Following is another example:

```
log "Log statement number " + logNo() at DEBUG;
action logNo() {
    logNumber := logNumber + 1;
    return logNumber.toString();
}
```

In this example, the correlator always executes the `log` statement because it calls an action. However, the log level in effect must be `DEBUG` for the correlator to send the string to the log file. If the log level is anything else, the correlator discards the string because the log level in effect is lower than the log level in the `log` statement.

Strings in print and log statements

In both `print` and `log` statements, the string can be any one of the following:

- Literal, for example: `print "Hello";`
- Variable, for example:

```
string welcomeMessage;
...
log welcomeMessage;
```

- Combination of both, for example:

```
string welcomeMessage;
```

```
...
print "Hello " + welcomeMessage + " Bye";
```

Internally, the correlator encodes all textual information as UTF-8. When the correlator outputs a string to a console or `stdout` because of a `print` statement, or sends a string to the log, the correlator translates the string from UTF-8 to the current machine's (where the correlator is running) local character set. However, if you redirect `stdout` to a file, the correlator does not translate to the local character set. This ensures that the correlator preserves as much information as possible.

Sample financial application

This section describes a complete financial example, using the monitor techniques discussed earlier in this chapter. See also: [“Example of a query” on page 60](#).

This example enables users to register interest, for notification, when a given stock changes in price (positive and negative) by a specified percentage.

Users register their interest by generating an event, here termed `Limit`, of the following format:

```
Limit(userID, stockName, percentageChange)
```

For example:

```
Limit(1, "ACME", 5.0)
```

This specifies that a user (with the user ID 1) wants to be notified if ACME's stock price changes by 5%. Any number of users can register their interests, many users can monitor the same stock (with different price change range), and a single user can monitor many stocks.

In EPL, the complete application is defined as:

```
event StockTick {
    string name;
    float price;
}

event Limit {
    integer userID;
    string name;
    float limit;
}

monitor SharePriceTracking {

    // store the user's specified attributes
    Limit limit;

    // store the initial price (this may be the opening price)
    StockTick initialPrice;

    // store the latest price - to give to the user
    StockTick latestPrice;

    // when a limit event is received spawn; creating a new
    // monitor instance for each user's request
    action onload() {
```

```
    on all Limit(*,*,>0.0):limit spawn setupNewLimitMonitor();
}

// If an identical request from a user is discovered
// stop this monitor and die
// If a StockTick event is received for the stock the
// user specified, store the price and call setPrice
action setupNewLimitMonitor() {
    on Limit(limit.userID, limit.name, *) die;
    on StockTick(limit.name, *):initialPrice setPrice();
}

// Search for StockTick events of the specified stock name
// whose price is both greater and less than the value
// specified - also converting the value to percentile format
action setPrice() {
    on StockTick(limit.name, > initialPrice.price * (1.0 +
        (limit.limit/100.0)):latestPrice notifyUser();

    on StockTick(limit.name, < initialPrice.price * (1.0 -
        (limit.limit/100.0)):latestPrice notifyUser();
}

// display results to user
action notifyUser() {
    log "Limit alert. User=" +
        limit.userID.toString() +
        " Stock=" + limit.name +
        " Last Price=" + latestPrice.price.toString() +
        " Limit=" + limit.limit.toString();
    die;
}
}
```

The important elements of this example lie in the life-cycle of different monitor states. Firstly a monitor instance is spawned on every incoming `Limit` event where the limit is greater than zero. Within `setupNewLimitMonitor`, the first `on` statement listens for other `Limit` events from the same user, upon detection of which the monitor instance is killed. This effectively ensures that there is a unique monitor instance per user per stock. This scheme also allows a user to send in a `Limit` event with a zero limit to indicate that they actually no longer want to monitor a particular stock. While this will not be caught by the original monitor instance's event listener and will not cause spawning, it will trigger the event listener in the monitor instance of that user for that stock and cause it to die.

Then a single `on` statement (without an `all`) sets up an event listener to look for all `StockTick` events for that stock type for that user. Once a relevant `StockTick` is detected, new event listeners start seeking a specific price difference for that user. If such a price change is detected it is logged. Note that the `log` statement exploits data from variables used before and after the spawn statement (that is, `limit` and `latestPrice`, respectively).

This example also demonstrates how mathematical operations may be used within event expressions. Here, two `on` statements create event listeners that look for `StockTicks` with prices above and below the calculated price. The calculated price in this case is based on the initial price multiplied by the percentage specified by the user. The first event listener is looking for an increase in the share price to 105% of its original value, while the second is looking for a decrease to 95% of its original value.

7 Implementing Parallel Processing

■ Introduction to contexts	286
■ Creating contexts	288
■ How many contexts can you create?	289
■ Using channels to communicate between contexts	289
■ Obtaining context references	290
■ Spawning to contexts	291
■ Channels and contexts	292
■ Sending an event to a channel	293
■ Sending an event to a particular context	294
■ Sending an event to a sequence of contexts	295
■ Common use cases for contexts	297
■ Samples for implementing contexts	297
■ Contexts and correlator determinism	304
■ How contexts affect other parts of your Apama application	304

By default, the correlator operates in a serial manner. In a monitor, you have the option of implementing contexts for parallel processing.

Note:

Queries automatically take advantage of parallel processing. You do not need to implement parallel processing in queries. The information in this section of the documentation is for application developers who are writing monitors.

During serial correlator operation, the correlator processes events in the order in which they arrive. Each external event matches zero or more listeners. The correlator executes a matching event's associated listeners in a rigid order. The correlator completes the processing related to a particular event before it examines the next event.

For some applications, this serial behavior might not be necessary. In this case, you might be able to improve performance by implementing parallel processing. Parallel processing lets the correlator concurrently process the EPL in multiple monitor instances. To implement parallel processing, you create one or more contexts.

Note:

If a license file cannot be found, the number of contexts that the correlator allows to be created is limited. See "Running Apama without a license file" in *Introduction to Apama*.

Parallel processing in the correlator is quite different from the parallel processing provided by Java, C++, and other languages. These languages allow shared state, and rely on mutexes, conditions, semaphores, monitors, and so on, to enforce correct behavior. The correlator does not automatically provide shared state. Data sharing happens by sending events between contexts and by using the `MemoryStore`. See ["Using the MemoryStore" on page 350](#). Parallel processing in the correlator is a message-passing system.

Introduction to contexts

Contexts allow EPL applications to organize work into threads that the correlator can execute concurrently.

In EPL, `context` is a reference type. When you create a variable of type `context`, or an event field of type `context`, you are actually creating an object that refers to a context. The context might or might not already exist. You can then use the context reference to spawn to the context or send an event to the context. When you spawn to a context, the correlator creates the context if it does not already exist.

What is inside/outside a context?

When you start a correlator it has a single main context. You can then create additional contexts. A context consists of the following:

- One or more monitor instances. Except, the main context exists even if it does not contain any monitor instances.
- An event input queue.

- Listeners that belong to the contained monitor instances.

The correlator maintains event definitions and monitor definitions outside contexts. This lets all contexts share the same event and monitor definitions.

Instances of the same monitor can exist in multiple contexts. Each monitor instance belongs to a single context. For example, suppose you inject monitor A. Monitor A spawns within its own context (the main context) twice and spawns once to the `alpha` context. This creates three additional monitor instances. Two instances are in the main context and one instance is in the `alpha` context. These instances do not share any data, other than by means of passing events.

About context properties

A context has the following properties:

- **Name** — A string that you specify when you create the context. This name does not need to be unique. The name is a convenient identifier that you can use in your code.
- **ID** — The correlator assigns a unique integer.
- **receiveInput flag** — A Boolean value that indicates whether the context can receive external input events on the default channel, which is the empty string (`""`).

A value of `true` lets the context receive external events on the default channel; this is a public context. A value of `true` is equivalent to a subscription to the default channel; there is no requirement for a monitor instance in this context to subscribe to the default channel.

A value of `false` indicates a private context that does not receive external events on the default channel. This is the default.

Note that the main context is public.

- **Channel subscriptions** — A context is subscribed to the union of the channels each of the monitor instances in that context is subscribed to. This is a property of the monitor instances running in a context and is not accessible by means of the context reference object.

You can spawn to other contexts. When the last monitor instance in a context terminates, that context stops doing work and stops consuming resources until you spawn another monitor instance to it.

In a context, when you route an event, the event goes to the front of that context's input queue. You can route events only within a context.

You can send an event to a particular context. When you do this, the event goes to the end of the specified context's input queue. The correlator processes it after it processes any other events that are already on the context's input queue. See [“Sending an event to a particular context” on page 294](#).

You can use a context as part of the key for a dictionary. You can route an event that contains a context field. You cannot parse a context. Context objects are immutable reference objects.

Context lifecycle

A context has a lifecycle that starts when a `spawn...to` operation occurs and ends when the last monitor instance in the context terminates. This is completely independent of any context objects that refer to the context. It is possible for a context to be running when no references to it exist, and it is possible for a context object to refer to a context that is no longer running. In the latter case, spawning to a context that is not running is permissible. The correlator restarts the context as required.

Note:

If a license file cannot be found, the number of contexts that the correlator allows to be created is limited. See "Running Apama without a license file" in *Introduction to Apama*.

Comparison of a correlator and a context

Upon injection, each monitor's initial instance runs in the main context. You must explicitly create additional contexts. Conceptually, a context is like a correlator but with the following differences:

- All contexts share the same namespace, and thus share all monitor and event definitions that have been injected.
- A monitor instance must have a context reference to pass an event to that context.
- Execution of Java is allowed in only the main context.
- The `engine_receive` tool receives events from all contexts or it can be configured to receive events from only specified channels.
- The `engine_send` tool sends events to all public contexts or to the contexts that are subscribed to the channels it is configured to send events on.

Creating contexts

In EPL, you refer to a context by means of an object of type `context`. The `context` type is a reference type.

The recommendation is to use private contexts and have monitor instances subscribe to the channels they require events from. This gives greater flexibility over using public contexts. For information on the constructors needed to create a context, see the description of the `context` type in the *API Reference for EPL (ApamaDoc)*.

The name of a context does not have to be unique, and is only used for diagnostic purposes (it is recommended that context names be meaningful and distinct). Creating a new context object with the same name as another context creates a reference to a different context, not the same context. Context references are independent to the actual context where monitors run. A context continues running if there are no references to it. A reference to a context may exist even though no active monitors are running in that context. You use the context reference to `spawn to` the context or send an event to the context. When you `spawn to` a context, the correlator creates the context if it does not already exist.

When you start a correlator, it has a single main context. You can then create additional contexts. Context reference objects are lightweight and creating one only creates a stub object and allocates an ID. In other words, when you create an EPL context object, you are actually creating a context reference.

The following example creates a reference, *c*, to a private context whose name is *test*:

```
context c:=context("test");
```

For information on the methods you can call on a context, see the description of the `context` type in the *API Reference for EPL (ApamaDoc)*.

See also [“How many contexts can you create?” on page 289](#).

How many contexts can you create?

You can create any number of contexts. A context is a very lightweight object. Creating a context just allocates an identifier and creates a small object. Consequently, it is possible to create a thousand contexts with little performance penalty.

You can have any number of running contexts. A running context means that the context contains at least one monitor instance that has work to do. The more CPU cores you have, the more contexts it is practical to be running at a given time. The performance of multiple contexts running concurrently should scale approximately according to the number of CPU cores available on the host.

Because the cost of each context is low, it is possible to divide applications into the finest level of parallelism possible and let the correlator balance running those contexts across all CPU cores. This is true even if that means creating very many contexts.

Using channels to communicate between contexts

Contexts can subscribe to channels, using the `monitor.subscribe(channelName)` operation. When a monitor executes `monitor.subscribe(channelName)`, it causes the context it is running in to be subscribed to that channel. The subscription's lifetime is tied to the lifetime of the monitor instance that executes `subscribe()`. The subscription is active until that monitor instance terminates or executes `monitor.unsubscribe(channelName)`.

Subscriptions are reference counted. That is, if one monitor instance subscribes twice to the same channel then it needs to unsubscribe twice from that channel. If two monitor instances each subscribe once to the same channel then the subscription is active while either monitor instance exists or until both monitor instances unsubscribe from that channel.

When a context is subscribed to a channel it receives all events sent on that channel. This includes:

- Events sent to the correlator from
 - An IAF adapter
 - `engine_send`

- Another correlator connected with `engine_connect` and using `parallel` mode
- Clients
- Universal Messaging
- Events sent from EPL using the `send...to` statement
- Events sent from EPL plug-ins to a specific channel

It does not include events emitted with the `emit...to` statement. Even if the target of an `emit...to` statement is a channel that the context is subscribed to, an event sent by the `emit` statement goes only to external receivers and not to any contexts.

By using a channel for each stream of data an application may be interested in, an application can control which streams of data it receives through execution of the appropriate `monitor.subscribe(channelName)` and `monitor.unsubscribe(channelName)` operations. The correlator can efficiently distribute events within the correlator to multiple contexts, plug-ins or receivers subscribed to channels. If further scale-out is required, using channels allows some application components to be deployed to correlator processes running on other hosts, which are connected using the `engine_connect` correlator tool or Universal Messaging. See "Tuning Correlator Performance" in *Deploying and Managing Apama Applications*.

Obtaining context references

To obtain a reference to the context that a piece of code is running in, call the `context.current()` method. This is a static method that returns a context object that is a reference to the current context. The current context is the context that contains the EPL that calls this method.

For a monitor instance to interact with the EPL by means of a context object in another context, the monitor instance must have a reference to that context. A monitor instance can obtain a reference to another context in only the following ways:

- By creating the context.
- By receiving a context reference, which must be of type `context`. A monitor instance can receive this reference by means of a routed or sent event, or a spawn operation.

For example:

```
on all Calculate() as calc {
    integer calcId:=integer.getUnique();
    spawn doCalculation(calc, calcId, context.current())
        to context("Calculation");
        do something
}
action doCalculation(Calculate req, integer id, context caller) {
    do something
    send CalculationResponse(id, value) to caller;
}
```

If a monitor instance that creates a context does not send a context reference outside itself, and does not subscribe to any channels, no other context can send events to that context, except by means of EPL plug-ins. This affords some degree of privacy for the context.

A context object (a context reference) does not do anything. It is simply the target of the following:

- `spawn ActionIdentifier([ArgumentList]) to ContextExpression;`

See [“Spawning to contexts” on page 291](#).

- `send EventExpression to ContextExpression;`

See [“Sending an event to a particular context” on page 294](#).

Spawning to contexts

In a monitor, you can spawn to a context. The format for doing this is as follows:

```
spawn ActionIdentifier([ArgumentList]) to ContextExpression;
```

Replace *ContextExpression* with any valid EPL expression that is of the context type. Typically, this is the name of a context variable. It is possible to spawn to only a context; it is not possible to spawn to a channel.

This statement asynchronously creates a new monitor instance in the target context. The correlator can immediately create the new monitor instance and begin processing it. The correlator does not need to finish processing the monitor instance that spawned to the context before it starts processing the spawned instance. The correlator might create the spawned monitor instance before it finishes processing the action that spawned the new instance. Or, the correlator might create the spawned monitor instance some time after it completes processing the action that spawned the new instance. The order is unpredictable. For example:

```
action analyse(string symbol) {
    context c:=context(symbol);
    spawn submon(symbol) to c;
    ...
}
action submon(string symbol) {
    ...
}
```

If the target context does not yet exist, the correlator creates it.

It is possible for an operation that spawns to a context to block if the input queue of the target context is full. See [“Deadlock avoidance when parallel processing” on page 305](#).

Like the regular `spawn...to` operation, the `spawn...to` operation does the following:

- Creates a new monitor instance by taking a deep copy of all of the spawning monitor instance's global variables
- Does not copy any listeners into the new monitor instance
- Runs the specified action in the new monitor instance

For general information about spawning, see [“Spawning monitor instances” on page 39](#).

Unlike the regular `spawn` operation, the correlator runs the new monitor instance in the specified context. The correlator concurrently processes the new monitor instance and the instance that spawned it.

A context processes `spawn` operations and events in the order in which they arrive. For example, suppose a monitor contains the following statements:

```
spawn action1() to ctx;  
send e1 to ctx;  
spawn action2() to ctx;  
send e2 to ctx;
```

The `ctx` context processes this in the following order: `action1()`, `e1`, `action2()`, `e2`.

Channels and contexts

Contexts can subscribe to particular channels to receive events delivered to those channels from adapters and from other contexts. See [“Channels and input events” on page 31](#) and [“Subscribing to channels” on page 54](#). Contexts that are public, that is, they were created with a `true` flag in the context constructor, have a permanent subscription to the default channel. The name of the default channel is the empty string.

Contexts can send events to channels without knowledge of whether the event is required by contexts, clients, adapters, or some combination. When an event is sent from a context to a channel the event is received by all contexts subscribed to that channel and by all external receivers that are listening on that channel. See [“Generating events with the `send` statement” on page 265](#).

An Apama query automatically runs in a context that has a permanent subscription to the default channel and to the `com.apama.queries` channel.

Channels are useful for:

- Identifying service monitors. If many monitors need to send events to a service monitor you can use a well known name (which can appear in EPL as a string literal or string constant) as a channel name. The service monitor (and only the service monitor) should subscribe to the channel and other monitors send events to that channel. When a request-response event protocol is required the sender can specify a channel to which it is subscribed, or a context to send the response to.
- Applications that have different contexts that consume different streams of data can use channels to send the data to the intended contexts, even if many contexts require the same data stream or one context requires multiple data streams. For example, statistical arbitrage trading strategies could run in many contexts, each subscribed to a channel for the pair of stock symbols it is trading against each other. If the adapter where the events are coming from is able to use a separate connection per channel, then the application will scale very well as more trading strategies on different symbols are added.
- Different components of an application can be de-coupled by using an event protocol that sends events to channels for each interaction point between components. This allows adapters to be replaced with monitors that simulate those adapters for testing, and makes it easy to scale an application across several hosts by running different parts on different correlators and then connecting them.

Sending an event to a channel

In a monitor, you can send an event to a channel by using either

- A string value that identifies the channel name.
- A `com.apama.Channel` type that either names a channel or holds a context reference.

The format for sending an event to a particular context is as follows:

```
send EventExpression to ChannelExpression;
```

Replace *EventExpression* with any valid EPL expression that is of an event type.

Replace *ChannelExpression* with any valid EPL expression that is of the `string` or `com.apama.Channel` type. Typically, this is a `string` value.

This statement asynchronously sends an event to everything subscribed to the specified channel. Subscribers can include:

- Contexts.
- Receivers connected to external components by means of Apama's messaging, JMS or Universal Messaging.
- EPL plug-ins that have subscribed an `EventHandler` object.

For each target subscribed to a channel, the event goes to the back of the context's input queue.

In a target context, the correlator can immediately process the sent event. The correlator does not need to finish executing the action that sends the event before it processes the sent event in a target context. The correlator might process the sent event before it finishes executing the action that sent the event. Or, the correlator might process the sent event some time after it completes executing the action that sent the event. The order is unpredictable. The order in which the target contexts receive the sent event is also unpredictable. For example:

```
action analyse(string symbol) {
    spawn submon(symbol) to context(symbol);
    log "Listening for "+symbol;
    on all com.apama.marketdata.Tick(symbol=symbol) as tick {
        send tick to symbol;
    }
    on com.apama.marketdata.Finished() {
        send com.apama.marketdata.Finished() to symbol;
    }
}

action submon(string symbol) {
    monitor.subscribe(symbol);...
}
```

It is possible for a `send...` operation to block the sending context from further processing if the input queue of any target (context, receiver or plug-in) is full. Either an event that you send to a particular target arrives on the target's input queue or the sending context waits for room on the target's input queue.

If you send an event to a channel that has no subscribers, the correlator discards the event because there are no listeners for it. This is not an error.

See also:

- [“Generating events with the send statement” on page 265](#)
- [“Using EPL plug-ins written in Java” on page 530](#)

Sending an event to a particular context

In a monitor, you can send an event to a particular context, as described here, or you can send an event to a sequence of contexts, described in the next topic. The format for sending an event to a particular context is as follows:

```
send EventExpression to Expression;
```

or:

```
enqueue EventExpression to ContextExpression;
```

Note:

The `enqueue . . . to` statement will be deprecated in a future release. Use the `send . . . to` statement. Both statements perform the same operation.

- Replace *EventExpression* with any valid EPL expression that is of an event type. You cannot specify a string representation of an event. For example, you cannot send `&TIME` pseudo-ticks.
- Replace *Expression*, in the first format, with any valid EPL expression that is of the context type or with a `com.apama.Channel` object that contains a context. See [“Sending events to `com.apama.Channel` objects” on page 266](#).
- Replace *ContextExpression* with any valid EPL expression that is of the context type. This can be the name of a context variable or a method that returns a context. This cannot be a `com.apama.Channel` object that contains a context.

This statement asynchronously sends an event to the specified context. The event goes to the back of the context's input queue.

In the target context, the correlator can immediately process the sent event. The correlator does not need to finish executing the action that sent the event before it processes the sent event in the target context. The correlator might process the sent event before it finishes executing the action that sent the event. Or, the correlator might process the sent event some time after it completes executing the action that sent the event. The order is unpredictable. The order in which the target contexts receive the sent event is also unpredictable. For example:

```
action analyse(string symbol) {  
    context c:=context(symbol);  
    spawn submon(symbol) to c;  
    log "Listening for "+symbol;  
    on all com.apama.marketdata.Tick(symbol=symbol) as tick {  
        send tick to c;  
    }  
}
```

```

    on com.apama.marketdata.Finished() {
        send com.apama.marketdata.Finished() to c;
    }
}
action submon(string symbol) {
    ...
}

```

The `send...to` and `enqueue...to` statements do not place the event on the special enqueued events queue. Instead, they put the event on the end of the target context's input queue. Consequently, it is possible for a `send...to` or `enqueue...to` operation to block the sending context from further processing if the input queue of the target context is full. Either an event that you send to a particular context arrives on the target context's input queue or the sending context waits for room on the target context's input queue.

If you send an event to a context that does not contain any monitor instances, the correlator discards the event because there are no listeners for it.

If you do not have a reference to a particular context, then send an event to a channel. See [“Generating events with the send statement” on page 265](#).

In some situations, for example when you change a single-context application to use parallel processing, you might want to explicitly send an event to only the context that contains the monitor instance that contains the send statement. To send an event to only this context specify:

```
send eventExpression to context.current();
```

You must set a valid value to a context variable before you send an event to the context. You cannot send an event to a context that you have declared but has not been set to a valid value. For example, the following code causes the correlator to terminate the monitor instance:

```

monitor m {
    context c;
    action onload()
    {
        send A() to c;
    }
}

```

See also [“Generating events with the send statement” on page 265](#).

Sending an event to a sequence of contexts

In a monitor, you can send an event to a sequence of contexts. The format for doing this is as follows:

```
send EventExpression to ContextSequenceExpression;
```

or:

```
enqueue EventExpression to ContextSequenceExpression;
```

Note:

The `enqueue...to` statement will be deprecated in a future release. Use the `send...to` statement. Both statements perform the same operation.

- Replace *EventExpression* with any valid EPL expression that is an event. You cannot specify a string representation of an event.
- Replace *ContextSequenceExpression* with any valid EPL expression that resolves to `sequence<context>`. You cannot specify a sequence that contains `com.apama.Channel` objects.

Each statement asynchronously sends a copy of an event to each context in the specified sequence. The event goes to the back of the input queue of each context.

In each target context, the correlator can immediately process the sent event. The correlator does not need to finish executing the action that sent the event (in the source context) before it processes the sent events in the target contexts. The correlator might process a sent event before it finishes executing the action that sent the event. Or, the correlator might process a sent event some time after it completes executing the action that sent the event. The order is unpredictable, depending on the relative execution speeds of the contexts.

The following example uses the sequence type:

```
action analyse(string symbol) {
    context c1:=context(symbol + "-1");
    context c2:=context(symbol + "-2");
    context c3:=context(symbol + "-3");

    spawn submon(symbol) to c1;
    spawn submon(symbol) to c2;
    spawn submon(symbol) to c3;
    sequence <context> ctxs := [ c1, c2, c3 ];

    log "Listening for "+symbol;
    on all com.apama.marketdata.Tick(symbol=symbol) as tick {
        send tick to ctxs;
    }
    on com.apama.marketdata.Finished() {
        send com.apama.marketdata.Finished() to ctxs;
    }
}
action submon(string symbol) {
    ...
}
```

The following example uses the `values()` method on a dictionary of contexts to obtain a sequence of contexts:

```
action analyse(string symbol) {
    context c1:=context(symbol + "-1");
    context c2:=context(symbol + "-2");
    context c3:=context(symbol + "-3");

    spawn submon(symbol) to c1;
    spawn submon(symbol) to c2;
    spawn submon(symbol) to c3;

    dictionary <string, context>
        ctxs := [ "c1": c1, "c2": c2, "c3": c3 ];
```



```

log "Listening for "+symbol;
on all com.apama.marketdata.Tick(symbol=symbol) as tick {
    send tick to ctxs.values();
}
on com.apama.marketdata.Finished() {
    send com.apama.marketdata.Finished() to ctxs.values();
}
}
action submon(string symbol) {
    ...
}

```

The `send...to` and `enqueue...to` statements do not place the event on the special enqueued events queue. Instead, they put the event on the end of the input queue of each target context. Consequently, it is possible for a `send...to` or `enqueue...to` operation to block the sending context from further processing if the input queue of a target context is full. The sending context does not continue beyond a `send...to` or `enqueue...to` statement until the event has been placed on the input queues of all target contexts.

If one of the contexts in the sequence does not contain any monitor instances the correlator ignores the sent event in that context because there are no listeners for it.

If one of the contexts in the sequence does not have a valid value before you send an event to it then the correlator terminates the monitor instance.

Consider the following two code fragments:

```

for c in mySequence {
    send myEvent to c;
}

send myEvent to mySequence;

```

Execution of each of these fragments is typically equivalent. However, you cannot rely on equivalence. When the correlator executes the first fragment, it always delivers the event to the contexts according to their order in the sequence. When the correlator executes the second fragment it can deliver the event to contexts in any order. For example, if a context's input queue is full this can affect the order in which the correlator delivers the event to the contexts.

Common use cases for contexts

See [“Tuning contexts” on page 425](#).

Samples for implementing contexts

Apama provides a number of applications that illustrate the use of contexts. These examples are in the `samples\ep1\contexts` directory and in the `samples\ep1\concurrency-theory` directory.

Information for using these examples is given in the topics below.

Simple sample implementation of contexts

In your Apama installation directory, in the `samples\ep1\contexts` directory, there are two versions of a simple application. One version implements serial processing and the other implements parallel processing. Open the `analyse-parallel.mon` and `analyse-serial.mon` files from the `Input` directory in Software AG Designer to compare the implementations.

The sample uses the PySys testing framework. To execute the sample, use `pysys run`. The script runs the serial application and then the parallel version.

On a 2.4GHz Quad core Intel Q6600 machine, the serial implementation completes in about 63 seconds, while the parallel implementation completes in about 17 seconds. For an equivalent dual-core processor, you can expect the parallel implementation to complete in about 30 seconds.

Look at `serial-results.evt` and `parallel-results.evt` to compare the results. While the per-symbol output for each implementation is identical, the ordering of sent events for different symbols is different. Also, in the parallel implementation, there is more variation in the time taken to process all events for one symbol. The sample uses eight worker contexts. Each context is doing much the same work, but on different segments of the data. While it is not required, an application that has eight contexts typically working most of the time benefits from running on an 8-core host. You can expect an 8-core processor to run the sample parallel implementation more than seven times faster than it runs the serial implementation.

Running samples of common concurrency problems

Sample applications in the `samples\ep1\concurrency-theory` directory illustrate a few common concurrency problems. There are three implementations of a simple deposit bank:

- **Race** — implements `Get` and `Set` events, and corresponding `Response` events, so that a teller can find the value of an account, perform some modification and then set the new account value.
- **Deadlock** — lets tellers lock an account.
- **Compareswap** — is similar to the race implementation but it does not rely on locking and it does not compute values based on out-of-date information.

> To run these samples

1. Start an Apama Command Prompt as described in *Deploying and Managing Apama Applications* in the topic "Setting up the environment using the Apama Command Prompt".
2. Change to the `samples/ep1/concurrency-theory` directory of your Apama installation.
3. Invoke the following:

```
pysys run -X MODE=mode
```

where *mode* can be *race*, *deadlock* or *compareswap*, according to which sample you want to run. The subsequent topics describe each sample.

The script starts a correlator on the default port (15903). Consequently, you should not have a correlator already running on the default port. If you do, the script causes the application to be injected into the running correlator and it also shuts the correlator down when the sample execution is complete. The script creates an event file in the `Output` directory (which it creates). The event file has the name of the sample with an `.evt` file suffix (for example, `race.evt`, `deadlock.evt` or `compareswap.evt`).

About the samples of concurrency problems

The sample of concurrency problems try to implement a simple deposit bank. The customer-visible part of the bank consists of a number of tellers, who have the ability to transfer money from one account to another. In an effort to scale well, the bank is implemented with each teller running in a separate context, which lets all tellers work concurrently. Of course, the simple work of the tellers does not require or even justify this, but the purpose of these samples is to show potential bugs, not to be a practical system. Similarly, no security checks are enforced.

Because data cannot be shared between contexts, the application requires a separate monitor that acts as the bank's database. The tellers send requests to the bank's database and receive responses from the database. There is also a simple mechanism to initialize the state of the bank database (`SetupAccount` event) and for tellers to discover the context in which the database is running. The communication between the bank and the tellers typically needs to get or set an account's value. The tellers perform the actual arithmetic on a bank account's value. Each implementation (*race*, *deadlock*, and *compareswap*) differs mainly in the way the tellers and database interact with each other.

Customer interactions with tellers are the same across all implementations. The customer sends a `TransferMoney` event, specifying which teller to use. It is assumed that customers know the names of tellers, the from and to account, and the amount to transfer. The customer receives a `TransferMoneyComplete` event when the transfer is complete.

The state of the bank's accounts can be inspected by sending a `SendBalances` event to the correlator, which causes the correlator to log and send the balances.

To expose the problems, there are calls to the `spinSleep` action at key places in the implementations. If the correlator receives an `ExposeRaces` event, the `spinSleep` action suspends work by the specified teller for the specified time. This simulates tellers working at different rates, and means that difficult to reproduce conflicts are easier to identify. While this is useful for exposing bugs, it is not suitable for general-purpose sleeps because it consumes CPU time while sleeping and does not let other work in that context get done. This strategy is useful for exposing problems only when you know exactly where to place the sleeps.

Each implementation has its own `transfer-sample_name.evt` file, which the script sends as each bug is exposed with a different set of input data.

About the race sample

The race sample is in `Bank-race.mon`. It implements `Get` and `Set` events, and corresponding `Response` events. A teller can find the value of an account, perform some modification and then set the new account value. To take money from one account, the protocol is as follows:

1. Send a `Get` event to obtain the current value of the account.
2. Wait for a `GetResponse` event that contains the current value.
3. Compute the new account value.
4. Send a `Set` event to set the new account value.
5. Wait for a `SetResponse` event.

This works well when a single transfer occurs at a time. However, there is a bug because between the time that teller 1 obtains an account value and the time that teller 1 sets the new account value, teller 2 can obtain the account value, compute a new value, and set a new account value. The following time line demonstrates this:

Time	Teller 1	Teller 2	Bank Database
0 (setup)	Transfer 50 from A to B		A: 100 B: 100 C: 100
	Get A, Get B		
	A=100, B=100		
	Sleep 1 second		
0.5		Transfer 25 from B to C	
		Get B, Get C	
		B=100, C=100	
		newB=75, newC=125	
		Set B, Set C	
			A: 100, B: 75, C: 125
1.0	newA=50, newB = 150		
	Set A, Set B		
			A: 50, B: 150, C: 125

B's account should have $100 + 50 - 25 = 125$. But it ends up with 150 because teller 1 overwrites teller 2's value for B's account (75). Teller 1 based its calculation on values that were out of date at the point they were sent to the database.

About the deadlock sample

While EPL does not provide any mutual exclusion locking primitives, you can implement something similar in a monitor. The deadlock sample's bank implements a locking mechanism. Tellers can send a `Lock` event for an account, and the database returns a `LockResponse` event when the account is locked. If another teller tries to lock the same account, the correlator queues the request until it processes an `Unlock` event to unlock the account. Note that the locking is fair; the correlator allocates locks in the order in which they are requested.

The deadlock implementation does no checking. For example, it does not check that the unlock event comes from the teller that locked an account, nor that a teller holds a lock for an account before performing an operation on that account. (A robust application would of course perform such checking.)

The deadlock sample fixes the problem shown in the race sample where a value was overwritten by a value that resulted from computation on out-of-date values. If you replicate the race pattern of events, teller 2 would wait to lock B's account until teller 1 had finished with it. (This assumes all tellers follow the correct protocols. A robust implementation would perform checks to ensure that was the case).

However, even when all tellers follow the locking protocol correctly, there is a different problem. If teller 1 locks account A and teller 2 locks account B, and teller 1 tries to lock account B and teller 2 tries to lock account A, then each teller waits for the other teller to release a lock. The following timeline shows this:

Time	Teller 1	Teller 2	Bank Database
0	Transfer 50 from A to B		A: 100 B: 100 C: 100
	Lock A		
			A: Locked by t1
	Sleep 1 second		
0.5		Transfer 25 from B to A	
		Lock B	
			A: locked by t1 B: locked by t2
		Lock A	A: locked by t1, t2 waitingB: locked by t2
		(waiting for LockResponse(A))	
	Lock B		A: locked by t1, t2 waitingB: locked by t2, t1 waiting

Time	Teller 1	Teller 2	Bank Database
1.0	(waiting for LockResponse(B))		

At this point, neither teller can make any further progress.

One solution to this (not implemented here) is to implement a timeout. If a lock request is outstanding for more than some threshold, the correlator abandons the lock. When this happens, the tellers would wait a random amount of time and try again. The random wait should prevent the retries from overlapping, if not on the first retry, then on a subsequent retry. However, such a mechanism invariably performs poorly in the (hopefully rare) case that a lock times out.

Alternatively, you can prevent deadlock by defining priority orders for locks. For example, you can specify that A must always be locked before B. Applying this priority order to all transactions would prevent deadlock.

About the compareswap sample

This compareswap sample is more like the race sample. The protocol between tellers and the database consists of Get and Set events, except the Set event is a CompareSet event, which contains an expected old value. If the old value does not match the database account value, then the teller retries the operation — getting a new value and re-computing the account value.

This has the advantage that it does not rely on locking (so does not suffer from deadlock) and does not result in values computed from out of date data being set in the database.

The only disadvantage is that under some circumstances (the same as for the race sample), the tellers need to re-try a calculation. However, unlike the timeout on locking, tellers know about this as soon as they receive an event back from the database, and no timeouts are involved.

This strategy is the recommended way to share state between different contexts. Note that while it guarantees progress is made by at least one context, an interaction between the database and a single context can take an unbounded amount of time, as other contexts can require the context to re-try its transaction. A further refinement would be to use a generation counter that the correlator increments on every successful Set event. This detects the difference between the database's value being unchanged and the database's value being changed back to a previous value. While such a difference might not matter in many situations, it might when you are computing interest.

Note:

Due to the requirement to retry, the compareswap implementation is slightly different from the race implementation. One account is modified at a time; the teller transfers money from the fromAccount, and then adds it to the toAccount.

Time	Teller 1	Teller 2	Bank Database
0 (setup)	Transfer 50 from A to B		A: 100 B: 100 C: 100
	Get A		

Time	Teller 1	Teller 2	Bank Database
	A=100		
	newA=50		
			A: 50, B: 100, C:100
			Set A success
	Get B		
	B = 100		
	Sleep 1		
0.5		Transfer 25 from B to C	
		Get B	
		B=100	
		newB=75	
		Set B (old=100)	
			A: 100, B: 75, C: 100
			Set B success
		Get C	
		C=100	
		newC=125	
		Set C (old=100)	
			A: 50, B: 75, C: 125
			Set C success
1.0	newB = 150		
	Set B (old=100)		
			A: 50, B: 75, C: 125
			Set B FAILED
	Get B		
	B = 75		
	newB = 125		

Time	Teller 1	Teller 2	Bank Database
	Set B (old=75)		
			A: 50, B: 125, C: 125
			Set B success

Contexts and correlator determinism

Creating one or more contexts makes the correlator non-deterministic. In other words, injecting the same monitor can produce different results if the monitor contains statements that spawn to contexts.

For example, suppose an application creates two contexts, spawns to each of them, and each context runs code that calls `integer.getUnique()`. The assignment of unique integers to contexts is not deterministic; if you re-run the code, each context might receive an integer that is different from the integer it received during the previous run. Other behavior that can be non-deterministic in a parallel processing application includes the following:

- The assignment of particular IDs to particular contexts
- The order in which contexts send events
- The order in which contexts spawn to other contexts

See also [“About input logs and parallel processing”](#) on page 304.

How contexts affect other parts of your Apama application

When you implement contexts in an EPL application, an understanding of how contexts affect other parts of your Apama application is required.

The topics below provide information to help you understand the behavior.

About input logs and parallel processing

Applications that implement parallel processing might have non-deterministic behavior. While you can inject a parallel application into a correlator that you started with the `--inputLog` option, you cannot expect to use that input log to exactly duplicate correlator execution.

For applications that use multiple contexts or that send events, just re-sending the events and EPL sent to the correlator is insufficient to reproduce the same output and state. The timing of which context ran which `send`, `emit`, `enqueue...` or other operation is important. Operations that can affect the state of other contexts or the sent events are non-deterministic when run in parallel.

Deadlock avoidance when parallel processing

Parallel processing in the correlator uses a message passing system. Each context has a fixed-size input queue for events (messages). A deadlock is possible when all of the following conditions are true:

- Context 1 is enqueueing an event to context 2.
- Context 2 is enqueueing an event to context 1.
- The input queues for context 1 and context 2 are both full.

In this situation, each context is blocked from further processing until the queue of the other context is no longer full. Neither context can process the next event on its input queue. Such a deadlock is not limited to two contexts but can occur with any number of contexts enqueueing events to each other.

The correlator avoids such a deadlock by detecting the potential for it to occur and then expanding input queues as needed. Also, the correlator logs a warning that a potential deadlock was detected. The correlator expands input queues only when not doing so causes a deadlock. The correlator does not expand input queues when one or more contexts are blocked from further processing while one or more contexts are processing as usual. However, it is still possible to create applications that result in out of memory errors or other kinds of deadlocks. Out of memory errors can result from requiring excessive expansion of input queues through the deadlock avoidance mechanism, or other means, such as creating a very large sequence.

Clock ticks when parallel processing

Since all contexts receive clock ticks, timers work in all contexts. However, it is possible for some contexts to run behind others. That is, a timer in a particular monitor for which there are monitor instances in multiple contexts might fire at different points in real time. In each context, the timer can process the series of clock ticks at a speed that is different from the other contexts.

A context that is running a monitor instance in a very long running loop might not remove entries from its input queue for a long time. If a context has a full input queue the clock tick distributor thread does not block. Instead, the correlator quashes clock ticks onto the end of the context's input queue. This means that the correlator unpacks the clock tick event when the context input queue either drains or accepts a new event. There is no perceptible difference between normally received clock ticks and quashed clock ticks.

Using EPL plug-ins in parallel processing applications

The standard MemoryStore and Time Format plug-ins are thread safe, which means that you can use them in parallel applications. The MemoryStore can be quite helpful in a parallel application and is very efficient when used simultaneously by multiple contexts.

8 Using Correlator Persistence

■ Description of state that can be persistent	308
■ When persistence is useful	309
■ When non-persistent monitors are useful	309
■ How the correlator persists state	310
■ Enabling correlator persistence	311
■ How the correlator recovers state	314
■ Designing applications for persistence-enabled correlators	316
■ Upgrading monitors in a persistence-enabled correlator	317
■ Sample code for persistence applications	318
■ Requesting snapshots from EPL	320
■ Developing persistence applications	320
■ Backing up the persistence database while the correlator is running	321
■ Using EPL plug-ins when persistence is enabled	323
■ Using the MemoryStore when persistence is enabled	323
■ Comparison of correlator persistence with other persistence mechanisms	325
■ Restrictions on correlator persistence	326

When the correlator shuts down, the default behavior is that all state is lost. When you restart the correlator, no state from the previous time the correlator was running is available. You can change this default behavior by using correlator persistence.

Correlator persistence means that the correlator automatically periodically takes a snapshot of its current state and saves it on disk. When you shut down and restart that correlator, the correlator restores the most recent saved state.

To enable persistence, you indicate in your EPL code which monitors you want to be persistent. Optionally, you can write actions that the correlator executes as part of the recovery process. When code is injected for a persistence application, the correlator that the code is injected into must have been started with a persistence option.

Persistent monitors must be written in EPL. State in JMon monitors cannot be persistent. State in chunks, with a few exceptions, also cannot be persistent.

To protect the security of personal data, see "Handling personal data "at rest" in the correlator persistence and JMS datastores" in *Developing Apama Applications*.

If you plan to install a new version of Apama, see "Persistence database backup" in *Installing Apama*.

Note:

If a license file cannot be found, the number of persistent monitors that the correlator allows is limited. See "Running Apama without a license file" in *Introduction to Apama*.

Description of state that can be persistent

A correlator that is running with persistence enabled automatically stores state on disk and automatically recovers state when it restarts. Saved state includes the following:

- For a persistent EPL monitor, all of that monitor's state is saved. This includes all events, strings, primitives, sequences, dictionaries, action variables, closures, and global variables. It also includes all the state of listeners, streams and queries — local variables captured by them and all active listeners, sublisteners and queries, including the events currently flowing through them.
- All source code that was injected into the correlator, including any non-persistent EPL monitors and JMon monitors. EPL files that were injected from a correlator deployment package (CDP) are not stored in plain text.

Code that is not injected includes the following:

- EPL plug-ins, which are imported at runtime. The actual plug-in file must be on a specified path that the correlator can load it from.
- Any Java class files on the correlator's classpath but not injected.
- The correlator runtime itself.
- Contents of all context queues.

- Some correlator-global state including `integer.getUnique()` and `integer.incrementCounter()` IDs and context IDs. See the *API Reference for EPL (ApamaDoc)* for more information on the `integer` type and its built-in methods `incrementCounter` and `getUnique`.

Note:

In general, chunks cannot be persistent. However, chunks used by the Apama Time Format plug-in and the Apama MemoryStore plug-in can be persistent.

When persistence is useful

Enabling correlator persistence is a good fit for applications in which it is unacceptable to lose any information. For example, an application for processing mortgage requests does not need to be available continuously. A small amount of downtime, especially outside business hours, might be acceptable. However, losing any state associated with a mortgage application would be unacceptable.

In such a mortgage processing application, there is unlikely to ever be a point at which there are no open applications and thus no state to preserve. But state might change over the course of weeks, rather than seconds. Enabling correlator persistence lets you implement complex event expressions such as the following:

```
on all LoanRequest() -> (PropertyValuation() and ProofOfIncome())
  within (4 * week) ...
```

With persistence enabled, the event expression can still be running even if weeks elapse between when it is created and when it finally completes. Without persistence, the event expression's state is susceptible to being lost if there are system restarts, software upgrades, and the like.

When non-persistent monitors are useful

A correlator that is running with persistence enabled can have persistent and non-persistent monitors injected. Non-persistence is a good choice for a monitor that does one or more of the following:

- Uses legacy code that does not use the persistence feature. See [“Designing applications for persistence-enabled correlators” on page 316](#).
- Interacts with user-defined EPL plug-ins or Apama EPL plug-ins other than the Time Format or MemoryStore plug-ins.
- Contains large amounts of fast-changing state that is undesirable to persist for performance reasons.
- Operates as a stateless utility that just responds to incoming events.
- Contains minimal state that can be reconstructed by the `onBeginRecovery()` action on a persistent monitor.

Also, all JMon monitors are non-persistent monitors.

How the correlator persists state

When persistence is enabled, the correlator periodically writes data to disk to reflect the correlator's runtime state. To do this, the correlator

1. Suspends all execution in the correlator across all contexts.
2. Takes an in-memory snapshot of what needs to be stored.
3. Resumes processing while the state is written to disk.

The correlator waits to suspend execution until all contexts have completed any in-progress event processing and any in-progress deletions. It can take time for the correlator to pause all contexts. Consequently, it is best practice that a single event listener does not take a long time to process. When there is a need to perform a large amount of work, try to split the work across multiple events.

How fine-grained to split work depends on the performance requirements of the application. Avoid very fine-grained work units as the overhead of scheduling will start to dominate and lead to the application running slowly.

Committing the snapshot to disk is an atomic operation. That is, a failure while storing state reverts the stored data to the previously successfully stored snapshot.

By default, the correlator does the following when you enable persistence:

- Takes a snapshot of state changes every 200 milliseconds. This is the snapshot interval. The correlator tracks the in-memory objects that have changed since the last snapshot and writes only that state to disk. If only a small fraction of the correlator's state changes, then only a fraction of the correlator's state must be stored for each snapshot.
- Automatically adjusts the snapshot interval. For example, if a significant percentage of the correlator's state changes, then the correlator increases the snapshot interval so that the overall throughput is not adversely affected.
- Stores persistent state in the current directory, which is the directory in which the correlator was started.
- Uses `persistence.db` as the name of the file that contains persistent state. This is the recovery datastore.
- Copies the recovery datastore to the input log if one was specified when the correlator was started. This happens only upon restarting the correlator.
- For applications that do not use the correlator's internal clock (correlators started with the `-xclock` option), the correlator uses the time of day in the last committed snapshot as the current time in the restarted correlator.

Enabling correlator persistence

Before you enable persistence, you should design and develop your application to handle persistence and recovery. See [“Designing applications for persistence-enabled correlators” on page 316](#).

Note:

If a license file cannot be found, the number of persistent monitors that the correlator allows is limited. See “Running Apama without a license file” in *Introduction to Apama*.

To enable correlator persistence, you must proceed as follows:

- Insert the word `persistent` before the monitor declaration for each monitor written in EPL that you want to be persistent. For example:

```
persistent monitor Order {
    action onload() {
        ...
    }
}
```

For a monitor declared as persistent, the correlator persists the state of all monitor instances of that name, and all instances of events that the monitor instances create.

You do not mark event types as persistent. Whether or not an event is persisted depends on whether it is used from a persistent or non-persistent monitor. If an event is on a context queue when the correlator takes a snapshot, the event is persisted.

- Optionally, define `onBeginRecovery()` and `onConcludeRecovery()` actions in your persistent monitors. The correlator executes any such actions as part of the recovery process. To determine whether you need to define these actions, see [“Designing applications for persistence-enabled correlators” on page 316](#), [“Defining recovery actions” on page 315](#) and [“Sample code for persistence applications” on page 318](#).
- Specify one or more persistence options when you start the correlator. To enable correlator persistence, you specify one of the following:

- the `-P` or `-Penabled=true` option, or
- the `--config` option together with the name of a YAML configuration file that contains the following definition:

```
correlator:
  persistence:
    enabled: true
```

Specify just one of the above options (without any additional persistence options) to implement the default behavior for correlator persistence.

To change the default behavior, also specify one or more of the options described in the table below. The correlator uses the default when you do not specify an option that indicates otherwise. For example, if you specify `-P`, `-PsnapshotIntervalMillis` and `-PstoreLocation` (or `--config` with a YAML configuration file that contains the corresponding options), the correlator

uses the values you specify for the snapshot interval and the recovery datastore location and uses the default settings for all other persistence behavior.

For more information on the different `-P` options and the `--config` option, see "Starting the correlator" in *Deploying and Managing Apama Applications*.

For information on all of the persistence options that you can specify in a YAML configuration file, see "Configuring persistence in a YAML configuration file" in *Deploying and Managing Apama Applications*.

Note:

During development of a persistence application, it varies whether you want to specify a persistence option when you start the correlator. In the earlier stages of development, you might choose not to specify a persistence option since you might make many and frequent changes to early versions of your program, thereby making recovery of a previous version impossible. For example, you might have changed the structure and perhaps added new variables. Once your program structure becomes relatively stable, you must take into account what happens during recovery and you will want to define `onBeginRecovery()` and `onConcludeRecovery()` actions. These actions never get called in a correlator that was not started with a persistence option. To deploy a persistence application, the correlator must be started with a persistence option.

- If you are using both correlator persistence and the compiled runtime (`--runtime compiled` option), we recommend the use of the `--runtime-cache` option to improve recovery times. For more information on these options, see "Starting the correlator" in *Deploying and Managing Apama Applications*.

The following table describes correlator persistence behavior, the default behavior, and the options you can specify to change default behavior.

Correlator Persistence Behavior	Default	Option for Changing
The correlator waits a specified length of time between snapshots.	200 milliseconds	<code>-PsnapshotIntervalMillis=interval</code> Or the corresponding option in a YAML configuration file: <code>snapshotIntervalMillis: interval</code> Specify an integer that indicates the number of milliseconds to wait.
The correlator can automatically adjust the snapshot interval according to application behavior.	<code>true</code> . The correlator automatically adjusts the snapshot interval.	<code>-PadjustSnapshot=boolean</code> Or the corresponding option in a YAML configuration file: <code>adjustSnapshot: boolean</code>
It can be useful to set this to <code>false</code> to diagnose a problem or test a new feature.		

Correlator Persistence Behavior	Default	Option for Changing
The correlator puts the recovery datastore in a specified directory.	The directory in which the correlator was started. That is, the current directory.	<p><code>-PstoreLocation=path</code></p> <p>Or the corresponding option in a YAML configuration file:</p> <pre>storeLocation: path</pre> <p>You can specify an absolute or relative path. The directory must exist.</p>
The correlator copies the snapshot into a specified file. This is the recovery datastore.	<code>persistence.db</code>	<p><code>-PstoreName=filename</code></p> <p>Or the corresponding option in a YAML configuration file:</p> <pre>storeName: filename</pre> <p>Specify a filename without a path.</p>
For correlators that use an external clock, the correlator uses a specified time of day as its starting time when it restarts.	The time of day captured in the last committed snapshot.	<p><code>-XrecoveryTime num</code></p> <p>To change the default, specify an integer that indicates seconds since the epoch.</p>
This behavior is useful only for replaying input logs that contain recovery information.		
The correlator can automatically copy the recovery datastore to the input log when a persistence-enabled correlator restarts.	The correlator copies the recovery datastore to the input log.	<p><code>-noDatabaseInReplayLog</code></p> <p>Or the corresponding option in a YAML configuration file:</p> <pre>includeDatabaseInInputLog</pre> <p>You might set this option if you are using an input log as a record of what the correlator received. The recovery datastore is a large overhead that you probably do not need. Or, if you maintain an independent copy of the recovery datastore, you probably do not want a copy of it in the input log.</p>

Important:

If an option is specified both with one of the `-P` options on the command line and in a YAML configuration file, the value on the command line takes precedence and a warning is logged.

How the correlator recovers state

When you restart a correlator for which persistence has been enabled the correlator

- Detects, recompiles, and re-injects all code that was injected and not deleted as of the last committed snapshot.
- Restarts and restores the state of all persistent monitors as of the last committed snapshot.
- Restarts non-persistent EPL monitors and JMon monitors at their `onload()` action.
- Executes any `onBeginRecovery()` and `onConcludeRecovery()` actions. See [“Defining recovery actions” on page 315](#).
- Recovers persistent connections (connections created with `engine_connect -p`) and resumes them at the first opportunity.

Code is re-injected in the order in which it was originally injected. The correlator tracks which objects (monitors, events, Java objects) were deleted and does not re-inject them. Such objects might have been deleted explicitly with the `engine_delete` utility or implicitly as when all instances of a monitor have terminated. If a snapshot shows that an object was deleted and then re-injected, recovery ignores the first injection and re-injects the monitor or event at the point of its second injection.

For a persistent monitor, recovery appears to be a pause in processing. This pause has the potential to be long enough to cause some events to be stale. All non-persistent monitors appear to have spontaneously reverted to their `onload` state. Communication channels to external components have been interrupted and can be assumed to not yet be connected. Except, the correlator treats connections created with `engine_connect -p`, which are persistent connections, the same as it treats persistent state. Persistent connections continue until you explicitly remove them. Upon recovery, the correlator tries to reconnect to the external components that were connected with persistent connections. However, events sent or received after the last committed snapshot might have been dropped because there is no reliable delivery on persistent connections.

For a non-persistent monitor, recovery appears the same as starting the correlator. The correlator's current time is up-to-date. The monitor is in the state it would be if it were just injected. External components have not yet connected to the correlator. If a monitor initiates a request of a non-persistent monitor, then the non-persistent monitor might have to queue the request until a connection is made to an external component, for example, the correlator subscribes to a data stream from an external adapter.

Recovery order

When the correlator recovers state from a recovery datastore, it does the following in the following order:

1. Recompile and re-inject all sources except for deleted events and monitors, which are ignored.
2. Restore objects and listeners in persistent monitors. The correlator does not execute any user code in the first two steps. While it sets up listeners, the listeners cannot yet change state.

3. Set `currentTime` to the `currentTime` of the last committed snapshot, which might be considerably earlier than the current time of day if the correlator was down for some time before recovering.
4. Initiate execution of any `onBeginRecovery()` actions on instances of restored events, monitors, and custom aggregate functions in all persistent monitor instances in all contexts. The order of execution of these actions is undefined. See [“Defining recovery actions” on page 315](#).
5. Quiesce — The correlator waits for all events that have been sent to a context to be processed, and also waits for any events that are sent to a context as a result of those events to be processed, and so on, until no more events are generated and sent to a context. The correlator also does this for `spawn...to` statements. This is similar to processing all events in all queues. Be careful not to generate an infinite loop of `send...to` statements.
6. Restore events, clock ticks, pending `spawn...to` statements, and so on, that were waiting on context queues when the snapshot was taken.
7. Send a single clock tick of the time at which the correlator is recovered, that is, the current time of day. If `-XrecoveryTime` was set when the correlator was started, the correlator uses that time for the current time of day.
8. Initiate execution of `onload()` actions in all non-persistent monitors in injection order.
9. Quiesce.
10. Initiate execution of any `onConcludeRecovery()` actions on instances of restored events, monitors, and custom aggregate functions in all persistent monitor instances in all contexts. The order of execution of these actions is undefined. See [“Defining recovery actions” on page 315](#).
11. Quiesce.
12. Start generating clock ticks.
13. Start taking persistence snapshots.
14. Open the server port. External components can now connect with the correlator, for example, IAF, `engine_send`, and `engine_receive`.

Defining recovery actions

In a persistent monitor, you can define one or two actions that the correlator executes as part of the recovery process:

- `onBeginRecovery()` — The correlator executes this action after it re-injects all source code and restores state in persistent monitors. The order of execution of `onBeginRecovery()` actions is undefined.
- `onConcludeRecovery()` — The correlator executes this action just before it begins sending clock ticks, taking persistent snapshots, and becoming available for connections to external components. The order of execution of `onConcludeRecovery()` actions is undefined.

Whether you define zero, one or both actions in each persistent monitor is application-dependent. See [“Designing applications for persistence-enabled correlators” on page 316](#) and [“Sample code for persistence applications” on page 318](#).

You can define an event and specify one or both of these actions as fields in the event. If an event defines a recovery action and an instance of the event is live in a persistent monitor, then the correlator calls the action(s) on those objects as well. A live event is reachable from a global variable or listener-captured local variable and consequently is not a candidate for garbage collection.

You can define `onBeginRecovery()` and `onConcludeRecovery()` actions in custom aggregate functions in the same way as you define them in events. When an aggregate function contains an `onBeginRecovery()` or `onConcludeRecovery()` action, this action is called on each custom aggregate function instance in a live query in a persistent monitor along with the `onBeginRecovery()` and `onConcludeRecovery()` actions in persistent monitors and events.

The order in which the correlator executes instances of `onBeginRecovery()` actions and instances of `onConcludeRecovery()` actions for objects in a monitor is not defined. If a monitor terminates after execution of `onBeginRecovery()` and before recovered queues have been flushed, the correlator does not call that monitor's `onConcludeRecovery()` action (if it has one). If the correlator terminates all of a monitor's listeners in one execution of `onBeginRecovery()`, later calls to `onBeginRecovery()` for that monitor instance still occur because they might instantiate new listeners. If no listeners exist in a monitor after `onBeginRecovery()` and `onConcludeRecovery()` have been executed for every object in that monitor, the monitor instance terminates as usual.

See [“Recovery order” on page 314](#) for more details about when `onBeginRecovery()` and `onConcludeRecovery()` are executed.

Simplest recovery use case

When you observe the following restrictions, the correlator's recovery behavior is straightforward:

- All monitors are persistent. The correlator contains no chunks.
- There are no implementations of `onBeginRecovery()` or `onConcludeRecovery()` actions.

EPL code that adheres to these restrictions appears to behave as if it is running in a completely reliable and fault tolerant system. The downside is that while the correlator is down, incoming or outgoing events are dropped. If you implement a “retransmit until acknowledge” protocol, then the correlator can have a large number of events (and retransmits) to process when it restarts, depending on how long it is down.

Designing applications for persistence-enabled correlators

When you are designing an application that you will deploy on a persistence-enabled correlator, you should consider the following issues:

- You do not need to re-inject code after you restart a persistence-enabled correlator. During recovery, the correlator obtains injected code from the recovery datastore.
- To recover from a hardware failure, you must maintain a copy of the recovery datastore on some form of reliable, shared storage. You want to ensure that the storage medium for the recovery datastore is not a single point of failure. This typically means putting it on a fileserver

with suitable levels of redundancy (disk, power supply, network and controller) that is accessible by two correlator host servers.

- The length of time between when a correlator shuts down and when it restarts is unpredictable. Consequently, you might want to implement `onBeginRecovery()` actions that do the following:
 - Specify behavior according to how long the down time was. For example, you could write a listener that ignores a subset of old events but matches on a new event.
 - Terminate on `all wait(...)` listeners. Such listeners have the potential to fire many times because the time jumps from the time of the last committed snapshot to the time at which the correlator was restarted.
- It is possible for persistent monitors to communicate with non-persistent monitors and to set up state, such as subscriptions to a stream of data, in a non-persistent monitor. If you need to recover this state, you must write code to do it in the `onConcludeRecovery()` action of a persistent monitor or an event within a persistent monitor. In a persistent monitor, having an event that manages an activity in a non-persistent monitor is a recommended practice.

Upgrading monitors in a persistence-enabled correlator

While injection order is fixed and you cannot change it, you might want to upgrade a monitor and this would appear to require a change in the injection order. That is, upon recovery, you want the correlator to restore the upgraded monitor and not the older version of the monitor.

Remember that it is an error if you try to inject a monitor while instances of that monitor are already running in the correlator. The correlator never injects a duplicate monitor definition.

In a correlator without persistence enabled, you can terminate all monitor instances and then inject the updated monitor definition. Since all old versions of the monitor had terminated, the correlator would correctly inject the updated monitor even though it had the same name. Also, since persistence is not enabled, there is no recovery process and so recovery of the older version of the monitor is not an issue.

In a persistence-enabled correlator, terminating all instances of a monitor you want to upgrade is unlikely to be an option. For more information, see [“Versioning and upgrading monitors” on page 336](#).

When your upgrade procedure terminates all instances of the old monitor the recovery process does not restore that monitor since all instances were deleted.

You might find that it makes more sense for your upgrade procedure to leave the instances of the old monitor running while changing the interface for whatever creates new instances of the monitor to create instances of the upgraded monitor instead of instances of the old monitor. The correlator would then be running some old versions of the monitor and some new versions of the monitor. Upon recovery, the correlator would recover both versions until all instances of the old monitor had terminated. This approach might be appropriate when the logic has changed so much that it is not practical to upgrade monitor instances, or when maintaining behavior for existing instances is desired.

Sample code for persistence applications

The topics below provide sample code for persistence applications.

See also [“Versioning and upgrading monitors” on page 336](#) which describes a sample for transferring monitor state using the MemoryStore.

Sample code for discarding stale state during recovery

The following code provides an example of discarding stale data during recovery. This application discards all recovered Data events because their data has become stale. However, the application always processes and does not discard ControlEvent events.

```
persistent monitor eg1 {
  listener l;
  listener lt;
  action onload() {
    initializeState();
    initiateListeners();
    on all ControlEvent() as c { handleControl(c); }
  }
  action initiateListeners() {
    l:=on all Data() as d { process(d); } // Process is moderately expensive
    lt:=on all wait(0.1) { send Average(state) to "output"; }
  }
  action onBeginRecovery() {
    l.quit(); // Discard all recovered Data events.
    lt.quit(); // Stop sending intermittent updates.
    // Do not flood receivers.
    // Note that the ControlEvent listener is still present.
    // The code throttles only Data events. If the
    // ControlEvent listener is not present, this monitor
    // would have no listeners and would terminate
    // after this action.
  }
  action onConcludeRecovery() {
    initiateListeners(); // Go back to normal.
  }
}
```

Sample code for recovery behavior based on downtime duration

The following sample is the same as the discard-stale-data sample with some changes that provide a downtime policy. Downtime is the duration between the last committed snapshot and the time of day upon recovery.

This code sample ignores downtimes that are less than two hours. However, if recovery starts just under the two-hour limit the processing of old data might appear to be beyond the two hour threshold. The downtime policy must take this into account.

```
persistent monitor eg1 {
  import "TimeFormatPlugin" as timeFormatPlugin;
  // ... onload() and so on
```

```

listener l;
listener lt;
action onload() {
    initiateListeners();
    // on all ControlEvent() as c { handleControl(c); }
}
action initiateListeners() {
    // l:=on all Data() as d { process(d); } // Process is moderately expensive
    //lt:=on all wait(0.1) { send Average(state) to "output"; }
}
boolean longDowntime;
action onBeginRecovery() {
    // currentTime is the time of the last snapshot, which is
    // approximately when the correlator went down.
    // timeFormatPlugin.getTime() is the actual time of recovery.
    if (timeFormatPlugin.getTime() - currentTime > (60.0 * 60.0 * 2.0) )
    {
        // If we were down for less than 2 hours, pretend nothing
        // happened. For longer gaps, skip stale data as it will be
        // too expensive to process it.
        longDowntime:=true;
        log "Correlator was down for a long time - will discard stale
            data.";
        l.quit(); // Discard all recovered Data events.
        lt.quit(); // Stop sending intermittent updates.
        // Do not flood receivers.
    }
}
action onConcludeRecovery() {
    if longDowntime {
        longDowntime:=false;
        initiateListeners(); // Go back to normal.
    }
}
}

```

Sample code that recovers subscription to non-persistent monitor

This sample code defines a persistent monitor that subscribes to a non-persistent service monitor. Note that the service monitor can handle the case where the subscription is received before the adapter is connected.

```

monitor service_monitor {
    boolean connected;
    sequence <Subscribe> pendingSubscribes;
    action onload() {
        on all Subscribe() as s {
            if not connected {
                pendingSubscribes.append(s);
            } else {
                if(incrRefCount(s.subkey)) {
                    send Adapter_Subscribe(s.subkey) to "output";
                }
            }
        }
    }
    on all wait(1.0) {

```

```
        send IsAdapterUp() to "output";
    }
    on all AdapterUp() {
        connected:=true;
        Subscribe s;
        for s in pendingSubscribes {
            route s;
        }
        pendingSubscribes.clear();
    }
}
action incrRefCount(string subkey) returns boolean {
    return false; }
}

persistent monitor eg2 {
    listener l;
    Instance i;
    context svcCtx;
    action spawnedInstance(context c) {
        svcCtx:=c; // Contains anything required to recover subscription.
        send Subscribe(i.subkey) to svcCtx;
        l:=on all Data() as d { process(d); }
    }
    action onConcludeRecovery() {
        // Non-persistent service monitor is now reset to its onload state.
        // Re-subscribe.
        send Subscribe(i.subkey) to svcCtx;
    }
}
```

Requesting snapshots from EPL

A persistent or non-persistent monitor can request a snapshot to occur as soon as possible using the Management interface. For details, see [“Using the Management interface” on page 391](#).

Developing persistence applications

While you are writing the EPL code for your persistence application, use Software AG Designer as you usually do, and do not enable persistence. When your application is near completion and has been successfully tested, start testing execution of the `onBeginRecovery()` and `onConcludeRecovery()` actions you defined in your application. Do this as follows:

1. Select **Run, Run configurations, Correlator** component.
2. Add `-P` to the command line of the correlator.
3. Start the correlator.
4. In the **Run configuration, Correlator** component, **Initialization** tab, disable all check boxes so that nothing is re-injected.
5. Stop and restart the correlator. It will have persisted the injected monitors.

6. Test the behavior of `onBeginRecovery()` and `onConcludeRecovery()` actions.
7. If everything is working correctly, you can stop here. Otherwise, modify your code and continue with the following steps.
8. Delete the `persistence.db` file.
9. In the **Run configuration, Correlator** component, **Initialization** tab, re-enable all check boxes so that your code is injected.
10. Start again at step 3 and continue until your code is working as desired.

Ensure that you delete the `persistence.db` file and re-inject fresh monitors only when loss of all state is acceptable, for example, during testing.

Backing up the persistence database while the correlator is running

Backing up the correlator persistence database while the correlator is running is not as simple as copying the file. This is because copying files happens by reading chunks of the file at a time and copying them elsewhere. In between reading chunks of the file, it is possible that the database is modified. Because of this, it is required to make an atomic snapshot of the database file, reading the entire state in one go. This can be done using file-system snapshots, a capability provided by many storage systems (for example, VMWare or NetApp) or operating systems (for example, Shadow Copy on Windows, or LVM with XFS and Ext4 file systems on Linux).

Proceed as follows to create a backup of the persistence database:

1. Create a snapshot of the volume containing the persistence database file.
2. Copy the persistence database from the snapshot.
3. Also copy all other files with similar names in the same folder. Using a wildcard filter such as `"persistence.db*"` will copy the following:
 - `persistence.db`
 - `persistence.db-journal`
 - `persistence.db-wal`
 - `persistence.db-shm`
4. Delete the snapshot when you have copied all required files. The snapshot is no longer needed.

Examples of how to create a snapshot and back up the persistence database on Windows and Linux are given below.

Backing up the persistence database using Shadow Copy on Windows

On Windows server platforms, you can create a persistent snapshot of the volume containing the persistence database, copy the database and related files from the snapshot, and then delete the snapshot.

It is also possible to use a temporary snapshot to copy the files in a single command. This works on all supported Windows operating systems. For example, you can take a snapshot using the VShadow tool. With this tool, snapshots are temporary by default. To invoke the tool, use an elevated command prompt (that is, run the Windows command prompt as an administrator) and enter the following:

```
vshadow -nw -script=SETVAR1.cmd -exec=copyPersistence.bat D:
```

where:

- -nw makes the backup faster by skipping applications that react to a snapshot being taken, which the database does not.
- -script generates a script called SETVAR1.cmd which provides variables for accessing the snapshot.
- -exec runs the script copyPersistence.bat (see below) before the tool exits (that is, while the snapshot still exists).
- The final argument is the drive that contains the persistence database.

The copyPersistence.bat script handles the actual copying. It has the following content:

```
call SETVAR1.cmd
for %%i in (%SHADOW_DEVICE_1%\path\to\persistenceDatabase\persistence.db*)
do copy %%i D:\backup\location
```

Backing up the persistence database using LVM on Linux

Proceed as follows:

1. Log in as root.
2. Create a snapshot of the volume containing the persistence database:

```
lvcreate -L1G -s -n snapshot /dev/vgname/persistence_volume
```

In this case, the name of the snapshot is “snapshot”.

Note:

Some file systems require you to pause writes while creating the snapshot. On XFS, for example, you have to run `xfs_freeze -f /myxfs` before running `lvcreate` and then `xfs_freeze -u /myxfs` after it completes.

3. To copy the database, first mount the snapshot:

```
mount /dev/vgname/snapshot /mnt
```

- Copy the relevant files from the snapshot:

```
cp /mnt/path/to/persistenceDatabase/persistence.db* /backup/
```

- Unmount the snapshot:

```
umount /mnt
```

- Remove the snapshot:

```
lvremove snapshot
```

Using EPL plug-ins when persistence is enabled

A persistent monitor can import an EPL plug-in only when the following conditions are met:

- None of the plug-in's functions/actions, including unused functions/actions, refer to a chunk type.
- The plug-in is capable of persisting its chunks. In this release, only the Time Format plug-in and the MemoryStore plug-in are capable of persisting chunks. User-defined EPL plug-ins and other Apama-provided plug-ins cannot persist chunks.

See also [“Restrictions on correlator persistence” on page 326](#).

Using the MemoryStore when persistence is enabled

When persistence is enabled, a persistent monitor can use the MemoryStore only with a correlator-persistent store. A correlator-persistent store is a store that was created by execution of the `storage.prepareCorrelatorPersistent(store name)` action. A persistent monitor cannot use a store that was created by executing any other `storage.prepare()` action. The only exception to this is if the persistent monitor is in a correlator for which persistence is not enabled. In this situation, the correlator treats persistent monitors in the same way it treats non-persistent monitors.

In a persistence-enabled correlator, both persistent and non-persistent monitors can use correlator-persistent stores. If you try to prepare an in-memory, on-disk or distributed store from a persistent monitor in a persistence enabled correlator, the correlator terminates the monitor that tries to do this. These are runtime errors. The compiler cannot catch these errors. The following table shows when you can use each kind of store.

Store type	Persistent correlator and persistent monitor	Persistent correlator and non-persistent monitor	Non-persistent correlator and persistent monitor	Non-persistent correlator and non-persistent monitor
In-memory		Yes	Yes	Yes
On-disk		Yes	Yes	Yes
Correlator-persistent	Yes	Yes*	Yes*	Yes*

Store type	Persistent correlator and persistent monitor	Persistent correlator and non-persistent monitor	Non-persistent correlator and persistent monitor	Non-persistent correlator and non-persistent monitor
Distributed		Yes	Yes	Yes

* Correlator-persistent store behaves as an in-memory store.

Snapshots include the contents of all correlator-persistent stores that are open. A snapshot can occur at any time, and it is not possible to commit only certain states of correlator-persistent stores or the tables in them. However, when using correlator-persistent stores from persistent monitors, failure and recovery of a correlator should appear as though nothing has happened. That is, all monitor state and table state should be as it was when the most recent snapshot was taken.

Just as you cannot execute `Store.persist()` for in-memory stores, you cannot execute the `Store.persist()` action on correlator-persistent stores. You can, however, use Apama's Management interface to request a snapshot of the entire correlator state and wait for that to complete. See [“Using the Management interface” on page 391](#).

In persistent monitors, `Store`, `Table`, `Row` and `Iterator` events are persistent and their state can be recovered to the latest snapshot. Persistent monitors should not see any inconsistency between the contents of the table and any state in the monitor, including `Store`, `Table`, `Row`, and `Iterator` events. Correlator-persistent stores behave the same as an in-memory stores, except that the state of correlator-persistent stores is preserved across correlator restarts.

When the correlator takes a snapshot, it includes `Row` events held by persistent monitors. Such `Row` events are, of course, versions of rows in a table that is in a correlator-persistent store. A persistence snapshot does not include `Row` events held by non-persistent monitors, even if they represent rows in tables that are in correlator-persistent stores.

Note:

The recovery datastore in which the correlator saves snapshots is different from the stores used with the `MemoryStore`. The recovery datastore contains the state of all persistent monitors, which might include `Row` events, `Iterator` events, and other `MemoryStore`-related events, and also the state of any correlator-persistent stores created with the `MemoryStore`. Thus, the recovery datastore contains any correlator-persistent stores. If non-persistent monitors have opened in-memory and/or on-disk stores, those stores operate independently of the recovery datastore. For example, a non-persistent monitor can request persistence for an on-disk store and this on-disk store would not be persisted in the recovery datastore.

In a `DataView`, you can expose only in-memory and on-disk stores; you cannot expose correlator-persistent stores.

See also [“Using the MemoryStore” on page 350](#).

Comparison of correlator persistence with other persistence mechanisms

Correlator persistence is not the only way to persist Apama application data. The table below compares the various features you can use to persist Apama data. As you can see, correlator persistence provides the most comprehensive, automatic persistence.

Persistence characteristic	Correlator persistence	MemoryStore	Apama Database Connector Adapter (ADBC)
Completeness of what is persisted	All state in persistent EPL monitors	Only state that you explicitly store. Partial listener evaluations are impossible to store.	Only state that you explicitly store. Partial listener evaluations are impossible to store.
Recovery mechanism	Automatic	Manual	Manual
EPL monitors can be notified about recovery	Yes	Yes	Yes
Supported across Apama versions	Yes *	Yes	Yes
Incremental snapshots	Yes	Yes	Yes
Storage type	Embedded	Embedded	Shared servers are supported. You can use any database server or driver.
Atomic snapshots	Yes	Yes	Yes
Performance benefit from pipelining disk writes with processing	Yes	Yes	Yes
Supports multiple contexts	Yes	Yes	Yes

* Please note those upgrading to 5.3 onwards with applications using persistence should read the information about backwards incompatibility at *Release Notes*, "What's New In Apama 5.3", "Backwards Incompatibility with persisted projects recovered to 5.3 from older versions".

Restrictions on correlator persistence

JMON monitors

JMon monitors cannot be persistent.

EPL plug-ins written in C++ and Java

A persistent monitor can use the Apama Time Format and MemoryStore EPL plug-ins and the chunk types contained by the events defined by those plug-ins. A persistent monitor cannot use any other chunk types. This means that a persistent monitor cannot use an event or plug-in that references a chunk type even if the application does not use those chunks.

Backwards compatibility

Please note those upgrading to 5.3 onwards with applications using persistence should read the information about backwards incompatibility at *Release Notes*, "What's New In Apama 5.3", "Backwards incompatibility with persisted projects recovered to 5.3 from older versions".

9 Common EPL Patterns in Monitors

■	Contrasting using a dictionary with spawning	328
■	Factory pattern	329
■	Using quit() to terminate event listeners	330
■	Combining the dictionary and factory patterns	331
■	Testing uniqueness	332
■	Reference counting	332
■	Inline request-response pattern	334
■	Writing echo monitors for debugging	335
■	Versioning and upgrading monitors	336

When developing EPL monitor applications it can be helpful to be familiar with common EPL patterns.

Contrasting using a dictionary with spawning

The sample code in this topic contrasts the use of a dictionary with spawning. Usually, the dictionary approach is preferred. This is because the spawning approach uses an unmatched event expression, which is vulnerable to maintenance issues if someone else loads an event listener for a pattern that you expect to have no other matches.

Translation using a dictionary

The events to be processed:

```
event Input { string value; }
event Output { string value; }
event Translation {
    string raw;
    string converted;
}
```

The monitor:

```
monitor Translator {
    dictionary < string, string > translations;

    action onload() {
        on all Translation() as t addTranslation(t);
        on all Input() as i translate(i);
    }
    action addTranslation(Translation t) {
        translations[t.raw] := t.converted ;
    }
    action translate(Input i) {
        if translations.containsKey(i.value) {
            send Output( translations[i.value] ) to "output";
        }
        else { fail(i); }
    }
    action fail(Input i ) {
        print "Cannot translate: " + i.value;
    }
}
```

Translation using spawning

Same events as translation using dictionary.

The monitor:

```
monitor Translator {
    action onload() {
        on all Translation() as t addTranslation(t);
        on all unmatched Input() as i fail(i);
    }
}
```



```

}
action addTranslation(Translation t) {
    spawn translation(t);
}
action translation(Translation t) {
    on all Input(t.raw) translate(t.converted);
}
action translate(string converted) {
    send Output(converted) to "output";
}
action fail(Input i) {
    print "Cannot translate: " + i.value;
}
}

```

Factory pattern

The factory pattern creates a new monitor instance to handle each new item/request. Its essential features include:

- The `onload()` action sets up an event listener for creation events,
- Each creation event causes a monitor instance to be spawned.

There are two common forms of the factory pattern:

- Canonical form

The monitor instance spawns to an action that initializes the state of the new monitor instance and creates event listeners specific to that monitor instance. The spawned monitor instances use local variables for coassignment and passes them into the action.

It is likely that some of the data from the creation event is copied into global variables.

- Alternate form

The initial monitor instance uses coassignment to global variables to set some state before spawning.

This is a "lazy" form in that it stores the complete creation event inside the monitor. You should not use this form if you are spawning large number of monitor instances and you have a large creation event, where only part of the creation event data needs to be retained.

As an exercise, consider rewriting the example in ["Translation using spawning" on page 328](#), to use the alternate factory form.

Canonical factory pattern

The event:

```
event NewOrder {...}
```

The monitor:

```
monitor OrderProcessor {
    ...
    action onload() {
        on all NewOrder() as order spawn processNewOrder(order);
    }
    action processNewOrder(NewOrder order) {
        ...
    }
}
```

Alternate factory pattern

The event:

```
event NewOrder {...}
```

The monitor:

```
monitor OrderProcessor {
    action onload() {
        on all NewOrder() as order spawn processOrder();
    }
    action processOrder() {
        ...
    }
}
```

Using quit() to terminate event listeners

The example below demonstrates the use of `quit()` to terminate an event listener. This example is somewhat contrived in order to demonstrate a situation where it might be desirable to use `quit()`. Typically, other methods are often more appropriate, for example, you can use `die` to kill a monitor instance and you can specify `and not` to terminate an event listener.

The example shows a monitor that trades received orders by breaking them into smaller orders, which it might place concurrently (perhaps on several exchanges). The monitor listens for fills on these orders, and sums up the fills. (A real monitor might also send status on what the filled volume is for each child order together with the total volume filled for the order. The logic for this is not shown here.) When each order is completely filled the monitor terminates the `Trade` event listener for that order.

The events:

```
event OrderIn {integer id; ... }
event OrderOut {integer id; integer volume; ... }
event Trade {integer orderOutId; integer volume; ... }
```

The monitor:

```
monitor TradeOrderAsSeveralSmallerOrders {
    event PlacedOrderRecord {
        listener listener;
        integer volumeToTrade;
        integer volumeTraded;
    }
}
```

```

}
dictionary < integer, PlacedOrderRecord > records;
action onload() {
    on all OrderIn() as theOrder spawn tradeOrder();
}
action tradeOrder() {
    // some logic determining when and what volume to trade
    ...
    placeOrder( volume ); //called multiple times
    ...
}
action placeOrder(integer volume) {
    PlacedOrderRecord r := new PlacedOrderRecord;
    integer id := integer.incrementCounter("orderId");
    r.listener := on all Trade(orderOutId=id) as t
        processTrade(t);
    records[id] := r;
    r.volumeToTrade := volume;
    route OrderOut(id,volume,...);
}
action processTrade(Trade t) {
    PlacedOrderRecord r := records[t.orderOutId];
    r.volumeTraded := r.volumeTraded + t.volume;
    if (r.volumeToTrade - r.volumeTraded) <= 0 {
        r.listener.quit();
        ...
    }
    ...
}
}
}

```

As stated earlier, for real-world solutions there is generally a better option than using `quit()`. For example, the exchange(s) probably also send `OrderComplete` events. In this case you can change the `on` statement as follows:

```

on all Trade(orderOutId=id) as t and not OrderComplete(orderOutId=id)
    processTrade(t);

```

Of course, you must be certain that the `OrderComplete` event can be received only after all trades for that order have been received.

Combining the dictionary and factory patterns

The dictionary and factory patterns are often combined. This pattern achieves separation of concerns by using two monitors. The first monitor is responsible for managing global concerns, for example, it ensures that each order has a unique key. The second monitor is responsible for local concerns, for example, it manages all data associated with processing that order.

The example does the following:

1. The `OrderFilter` monitor accepts `NewOrder` events and checks for uniqueness of the order key.
2. For all orders with unique keys, the `OrderFilter` monitor routes a `ValidOrder` event.

Testing uniqueness

The events:

```
event OrderKey{...}
event NewOrder {
    OrderKey key; //You can use anything for key as long as it is unique
    ...
}
event ValidNewOrder {
    NewOrder order;
}
```

The monitors:

```
monitor OrderFilter {
    dictionary < OrderKey, NewOrder > orders;
    action onload() {
        on all NewOrder() as order validateOrder(order);
    }
    action validateOrder(NewOrder order){
        if orders.containsKey(order.key) {
            print "Duplicate order!";
            print "Original: " + orders[order.key].toString();
            print "Incoming: " + order.toString();
        }
        else {
            orders.add(order.key,order);
            route ValidNewOrder(order);
        }
    }
}

monitor OrderProcessor {
    ...
    action onload() {
        on all ValidNewOrder() as valid spawn processOrder(valid.order);
    }
    action processOrder( NewOrder order ) {
        ...
    }
}
```

Reference counting

The following pattern is another example that you can use to keep a count of how many clients are using a particular service object, which in turn can be used to determine the lifetime of these service objects. The example subscription management mechanism is fairly sophisticated, possibly too sophisticated, but it provides the big advantage of separating the concerns by using two monitors. If you decide to change the subscription mechanism, you can do so simply by changing the ServiceManager monitor. There is no impact at all on the ServiceItem monitor.

The events:

```
package com.apamax.service;
```

```

event Subscribe {
    string toWhat;
    string originator;
}
event Unsubscribe {
    string fromWhat;
    string originator;
}
event CreateServiceItem {
    string what;
}
event DestroyServiceItem {
    string what;
}

```

The monitors:

```

monitor ServiceManager {
    dictionary <string, dictionary<string, integer>> items;

    action onload() {
        on all Subscribe() as s subscribe(s);
        on all Unsubscribe() as u unsubscribe(u);
    }
    action subscribe(Subscribe s){
        dictionary < string, integer > subscriptions:= {};
        if items.containsKey(s.toWhat) {
            subscriptions :=
                items[s.toWhat];
            if subscriptions.containsKey(s.originator) {
                subscriptions[s.originator] :=
                    subscriptions[s.originator] + 1;
            }
            else {
                subscriptions[s.originator] := 1;
            }
        }
        else {
            items[s.toWhat] := subscriptions;
            route CreateServiceItem(s.toWhat);
        }
    }

    action unsubscribe(Unsubscribe u) {
        if items.containsKey(u.fromWhat) {
            dictionary < string, integer > subscriptions :=
                items[u.fromWhat];
            if subscriptions.containsKey(u.originator) {
                if subscriptions[u.originator] <= 1 {
                    subscriptions.remove(u.originator);
                    if subscriptions.size() = 0 {
                        items.remove(u.fromWhat);
                        route DestroyServiceItem(u.fromWhat);
                    }
                }
            }
            else {
                subscriptions[u.originator] :=
                    subscriptions[u.originator] - 1;
            }
        }
    }
}

```

```
        else {
            print "Unsubscribe failed: no originator: " +
                u.toString();
        }
    }
    else {
        print "Unsubscribe failed: no item: " + u.toString();
    }
}

monitor ServiceItem {
    //...

    action onload() {
        on all CreateServiceItem() as c spawn createServiceItem(c);
    }

    action createServiceItem(CreateServiceItem c) {
        //...
        on all DestroyServiceItem() as d destroyServiceItem(d);
    }

    action destroyServiceItem(DestroyServiceItem d) {
        //...die;
    }
}
```

Inline request-response pattern

You can use the `route` statement to write EPL that exhibits inline (synchronous) request-response behavior. The following example shows that when you want to perform an ordered pattern of operations that contain (as one operation) a request to another monitor, the subsequent operations must wait until the requesting monitor receives the response.

The ordering of the `route` and `on` statements is not relevant. The correlator sets up the event listener before processing the routed event.

A common mistake is to place code after the `on` statement code block and expect that code to execute after the code in the `on` statement code block.

Routing events for request-response behavior

The events:

```
event Request { integer requestId; ... }
event Response { integer requestId; ... }
```

The monitors:

```
monitor Client {
    action doWork() {
        //do some processing
        ...
        integer id := integer.getUnique();
    }
}
```

```

    route Request(id, ... );
    on Response(requestId=id) as r {
        // continue processing
        ...
        // Beware! Any code here will execute immediately
        // (before processing the response)
    }
}

monitor Server {
    action processRequests() {
        on all Request() as r {
            // evaluate response
            route Response(r.requestId);
        }
    }
}

```

Canonical form for synchronous requests

The next example show the canonical form for when you want to code a pattern that specifies two or more synchronous requests.

The events:

```

event RequestA { integer requestId; ... }
event ResponseA { integer requestId; ... }
event RequestB { integer requestId; ... }
event ResponseB { integer requestId; ... }

```

The monitor:

```

monitor Client {
    action doWork() {
        //do some processing
        integer requestId := integer.getUnique();
        route RequestA(requestId,...);
        on ResponseA(id=requestId) as ra doWork2(ra);
    }
    action doWork2(ResponseA ra) {
        //do some more processing
        integer requestId := integer.getUnique();
        route RequestB(requestId,...);
        on ResponseB(id=requestId) as rb doWork3(rb);
    }
    action doWork3(ResponseB rb) {
        //do yet more processing
    }
}

```

Writing echo monitors for debugging

A common practice is to write an echo monitor for debugging purposes. Typically, an echo monitor listens for the same events as your production monitor and tracks various behavior.

Writing an echo monitor is typically straightforward, but keep the following caveat in mind. If your production monitor uses the `unmatched` keyword for a certain event, and your echo monitor listens for the same event, and both monitors are in the same context, your `unmatched` event listener will never trigger. This is because the event listener in the echo monitor matches the event and this prevents the `unmatched` event listener from ever triggering. The scope of an `unmatched` event listener is the context that it is in.

To avoid an `unmatched` event listener that never triggers, specify the `completed` keyword in the event listener in the echo monitor. For example, suppose you have the following code in your production monitor:

```
on all unmatched SubscribeDepth() as subDepth {  
    doSomething();  
}
```

If you want to track `SubscribeDepth` events in your echo monitor, write the event expression in the echo monitor as follows:

```
on all completed SubscribeDepth() as subDepth {  
    doSomethingElse();  
}
```

The `completed` event listener in the echo monitor triggers after the correlator finishes processing the `unmatched` event listener in the production monitor.

Versioning and upgrading monitors

If an application requires functionality to upgrade its monitor instances while still running, there are architectural patterns that must be used during application design. A monitor must contain code that enables its state to be transferred to a new version and for it to terminate upon request. If a deployed monitor does not contain such code, it is not possible to upgrade while transferring its state.

The sample application in the `samples/epl/hot-redeploy/StoreState` directory of the Apama installation shows how to transfer monitor state using the `MemoryStore` (see also [“Using the MemoryStore” on page 350](#)). This sample takes the following steps to upgrade monitor instances:

1. Injects version 1 of the Counter application.
2. Processes version 1 until an upgrade is required.
3. Stops the input events and flushes all queues to ensure that all current events have been processed.
4. Injects version 2 of the Counter application which sends an `Upgrade` event. This informs the version 1 monitor that an upgrade is happening and it should save its state and its listeners should be destroyed.
5. Once version 1 has finished processing the `Upgrade` event, the version 2 monitor iterates the `MemoryStore` table and loads the state of each monitor. The version 2 monitors take over the processing of the input events from version 1.
6. Starts the input events.

7. Processes version 2.

The monitor state that is stored is limited to a small number of required variables contained in a `State` event that can easily be stored in the `MemoryStore`. Complex types (such as chunks, listeners or action references) cannot be stored in the `MemoryStore` and should not be included in a monitor's state. The application monitor supplies a callback to handle the old state, which is typically done by converting the old `State` object to a new `State` object. This can be done automatically for fields of the `State` object that are the same type and name. Added fields or fields whose type is changed, however, need to be handled explicitly by upgrade code. The application monitor can then spawn to handle the upgraded instances. For more information, see the `README.txt` file and the source code files of the sample application.

This sample is deliberately simple to only show the concept of storing and retrieving a monitor instance state. An application may require more functionality. For example, you can enhance the sample as follows:

- Use the distributed `MemoryStore` to transfer monitor instances between correlators (with the same input).
- Spawn to other contexts. This sample only spawns instances to the main context. It could spawn to other contexts, but you need to take care if instances need to be restored to the same context. You would need some form of context store and lookup.
- Add a dedicated channel for the `Upgrade` events that all monitors in any context could subscribe to.
- Modify the sample to work with correlator persistence by marking the monitors as persistent and using `prepareCorrelatorPersistent()`.
- Only kill the monitor factory on `Upgrade` events and allow all partially evaluated event expressions to run with the old version, but all new instances will use the new version.

Note:

This sample only transfers global monitor instance state. It does not transfer event expressions; these will be lost. It is possible to allow partially evaluated event expressions to complete (by not explicitly killing or deleting them) but have all new event expressions created by the new version.

10 Using EPL Plug-ins

■ Overhead of using plug-ins	340
■ When to use plug-ins	340
■ When not to use plug-ins	341
■ Using the TimeFormat Event Library	341
■ Using the MemoryStore	350
■ Using the distributed MemoryStore	365
■ Using the Management interface	391
■ Using the JSON plug-in	395
■ Using MATLAB® products in an application	396
■ Using the R plug-in	404
■ Interfacing with user-defined EPL plug-ins	406
■ About the chunk type	406

In EPL programs (monitors and queries), you can use standard EPL plug-ins provided with Apama and you can also use EPL plug-ins that you define yourself. An EPL plug-in consists of an appropriately formatted library of C++ functions, which can be called from within EPL code. In the case of a plug-in written in Java, the Java classes that are called from an application's EPL code are contained in a jar file. The correlator does not need to be modified to enable or to integrate with a plug-in, as the plug-in loading process is transparent and occurs dynamically when required.

To write custom EPL plug-ins, see .

When using a plug-in, you can call the functions it contains directly from EPL, passing EPL variables and literals as parameters, and getting return values that can be manipulated.

Overhead of using plug-ins

The overhead when using EPL plug-ins is very small.

However, you do need to ensure that you do not block the correlator for a long period of time. For example, you do not want to use a plug-in for doing extensive, synchronous, time-consuming calculations.

If you need to perform a time-consuming operation, use asynchronous processing in the plug-in to perform the calculation on a background thread, and then deliver the result via an event to the monitor instance which requested it. For example, your plug-in method might return an ID for the monitor to listen to an event with that ID. When the calculation is complete, the plug-in sends an event with that ID to the context from which it was originally called:

```
integer id := plugin.compute(inputs);
on ComputeResult(id=id) as cr {
    // process cr.result
}
```

When to use plug-ins

Custom plug-ins can be written in C++, Java or Python. A custom plug-in is a suitable solution in the following situations:

- You have an in-house or third-party library of (possibly complex) functions or classes that you want to re-use.
- The operations you need to perform are more easily/efficiently performed in another language than using EPL. For example, you need to use data structures that are not easily represented in EPL.

Note:

If your concern is purely performance, then the compiled EPL runtime (available on Linux, see also the `--runtime` option in "Starting the correlator" in *Deploying and Managing Apama Applications*) may be sufficient and in some cases can produce results better than other languages, including C++ and Java.

When not to use plug-ins

In general, when you can efficiently write the desired operation in EPL, an all-EPL solution is preferable to one that involves custom-developed plug-ins. Apama customers who experience problems with correlator stability when using custom-developed plug-ins will be asked by Software AG Global Support to remove the plug-in and reproduce the problem prior to being offered further technical help. Software AG Global Support lifts this restriction only if the plug-ins have certification from Apama product management.

Using the TimeFormat Event Library

The TimeFormat event library uses the Time Format plug-in.

The TimeFormat event provides routines to obtain the current time and convert to or from string representations of time.

Internally, the correlator expresses time as seconds since the Unix Epoc (1 Jan 1970, midnight UTC) - this is the form of `currentTime` and is convenient for performing arithmetic, such as differences between times. For more information on this variable, see [“currentTime” on page 680](#).

To convert from string form to float form, use a `parseTime` method. To convert from float form to string form, use a `format` method. Both take a `format String`, which is a string which describes the string form of the time. For more information, see [“Format specification for the TimeFormat functions” on page 343](#).

The `parseTime` method is available on the TimeFormat event directly. Or you can pre-compile a pattern and then perform parsing using the compiled pattern. A `CompiledPattern` object is obtained from the TimeFormat event using one of the `compilePattern` methods (depending on which time zone the pattern should use by default). The `CompiledPattern` object can be stored in a monitor variable, as an instance of an event or in a local variable and used by listeners. Re-using a `CompiledPattern` is more efficient than calling one of the `TimeFormat.parseTime` methods as the `format String` only needs to be read and compiled once. Calling `parse` on the TimeFormat event is equivalent to passing the same `format String` to generate a `CompiledPattern` and calling `parse` on that event. It is also possible to create multiple `CompiledPattern` events if your application needs to use several different formats for time.

For example, the following will behave the same:

```
TimeFormat.parseTime(pattern, time);
TimeFormat.compilePattern(pattern).parseTime(time);
```

There are also functions to obtain the current system time. `getSystemTime()` provides an absolute time while `getMicroTime()` provides a high precision time, which is suitable for high precision relative times (the absolute value of `getMicroTime()` depends on the host operating system).

Patterns with textual elements operate by default in English, but will instead both produce output and expect input in another language if that has been set in the environment. For example, under Linux, if the correlator is running with the `LC_ALL` environment variable set to `"fr_FR"`, the format `"EEEE dd MMMM yyyy G"` produces and expects `"jeudi 01 janvier 1970 ap. J.-C."` for time `0.0`.

When you use the TimeFormat event library you can use the TZ environment variable to select a particular locale to be used by the event library. Specify the value in either of the following formats:

```
Continent/City  
Ocean/Archipelago
```

For example: TZ=Europe/London. The alternative shortened format will not work correctly. For example, TZ=GB will not be recognized. If you specify something like this, Coordinated Universal Time (UTC) is used instead.

Note:

For a list of time zones, see "Timezone ID Values" in the "Using the Dashboard Viewer" part of *Building and Using Apama Dashboards*.

TimeFormat format functions

The format functions convert the `time` parameter to the local time and return that time in the format you specify.

For usage information, see the *API Reference for EPL (ApamaDoc)*.

TimeFormat parse functions

The parse functions parse the value contained by the `timeDate` parameter according to the format passed in the `format` parameter or wrapped by the `CompiledPattern`.

All functions return the result as a float of seconds since the epoch.

For usage information, see the *API Reference for EPL (ApamaDoc)*.

Notes

For all parse functions:

- If the `timeDate` parameter specifies only a time, the date is assumed to be 1 January 1970 in the appropriate timezone. If the `timeDate` parameter specifies only a date, the time is assumed to be the midnight that starts that day in the appropriate timezone. Adding them together as seconds gives the right result.
- If the `timeDate` string specifies a time zone, and there is a matching `z`, `Z`, `v`, or `V` in the `format` string, the time zone specified in the `timeDate` string takes precedence over any other ways of specifying the time zone. For example, when you call the `parseUTC()` or `parseTimeWithTimeZone()` function, and you specify a time zone or offset in the `timeDate` string, the time zone or offset specification in the `timeDate` string overrides the time zone you specify as a parameter to the `parseTimeWithTimeZone()` function and the normal interpretation of times and dates as UTC by the `parseUTC()` function.
- Parsing behavior is undefined if the format string includes duplicate elements such as "MM yyyy MMMM", has missing elements such as "MM", or it includes potentially contradictory elements and is given contradictory input, for example, "Tuesday 3 January 1970" (it was actually a Saturday).

- Dates before 1970 are represented by negative numbers.

Example

The following example returns 837007736:

```
TimeFormat.parseTime("yyyy.MM.dd G 'at' HH:mm:ss", "1996.07.10 AD at 15:08:56")
```

See also [“Midnight and noon” on page 349](#).

The following examples both parse the *timeDate* string as having a time zone of UTC+0900:

```
TimeFormat.parseTimeWithTimeZone("DD.MM.YY Z", "01.01.70 +0900", "UTC");
TimeFormat.parseUTC("DD.MM.YY Z", "01.01.70 +0900");
```

In the first example, the +0900 specification in the *timeDate* string overrides the UTC specification for the time zone *name* parameter. In the second example, the +0900 specification in the *timeDate* string overrides the UTC specified by calling the `parseUTC()` function.

Format specification for the TimeFormat functions

The format and parse functions make use of the `SimpleDateFormat` class provided in the International Components for Unicode libraries. `SimpleDateFormat` is a class for formatting and parsing dates in a language-independent manner.

Pattern letters in format strings

The `TimeFormat` functions use the `SimpleDateFormat` class to transform between a string that contains a time and/or date and a normalized representation of that time and/or date. In this case, the normalized representation is the number of seconds since the epoch.

For the operation to succeed, it is important to define the format string so that it exactly represents the format of the time and/or date you provide as a string in the *timeDate* parameter to a parse function, or expect to be returned from a format function. You specify the format as a time pattern. In this pattern, all ASCII letters are reserved as pattern letters.

The number of pattern letters determines the format as follows:

- For pattern letters that represent text
 - If you specify four or more letters, the `SimpleDateFormat` class transforms the full form. For example, `EEEE` formats/parses Monday.
 - If you specify fewer than four letters, the `SimpleDateFormat` class transforms the short or abbreviated form if it exists. For example, `E`, `EE`, and `EEE` each formats/parses Mon.
- For pattern letters that represent numbers
 - Specify the minimum number of digits.
 - If necessary, `SimpleDateFormat` prepends zeros to shorter numbers to equal the number of digits you specify. For example, `m` formats/parses 6, `mm` formats/parses 06.

- Year is handled specially. If the count of *y* is 2, the year is truncated to 2 digits. For example, *yyyy* formats/parses 1997, while *yy* formats/parses 97.
- Unlike other fields, fractional seconds are padded on the right with zeros.
- For pattern letters that can represent text or numbers
 - If you specify three or more letters, the `SimpleDateFormat` class transforms text. For example, *MMM* formats/parses Jan, while *MMMM* formats/parses January.
 - If you specify one or two letters, the `SimpleDateFormat` class transforms a number. For example, *M* formats/parses 1, and *MM* formats/parses for 01.

The following table provides the meaning of each letter you can specify in a pattern. After the table, there are a number of combined examples.

Descriptions of pattern letters in format strings:

Symbol	Meaning	Presentation	Example	Sample Result
G	Era designator	Text	G	AD
			G	BC
y (lowercase)	Year	Number	yy	96
			yyyy	1996
Y (uppercase)	Year for indicating which week of the year. Use with the <i>w</i> symbol. See “Week in year” later in this table.	Number	See example for “Week in year”.	
u	Extended year	Number	uuuu	5769
M	Month in year	Text or Number	M	9
			MM	09
			MMM	Sep
			MMMM	September
d	Day in month	Number	d	7
			dd	07
			dd	25
h	Hour in AM or PM (1-12)	Number	hh	05
H	Hour in day (0-23)	Number	H	0

Symbol	Meaning	Presentation	Example	Sample Result
	See also “Midnight and noon” on page 349.		HH	05
			HH	14
m	Minute in hour	Number	m	3
	See also “Midnight and noon” on page 349.		mm	03
			mm	55
s	Second in minute	Number	s	5
			ss	05
			ss	59
S	Fractional second	Number	S	2
			SS	20
			SSS	200
E	Day of week	Text	E	Fri
			EE	Fri
			EEE	Fri
			EEEE	Friday
e	Day of week (1-7)	Number	e	4
	This is locale dependent. Typically, Monday is 1.			
D	Day in year	Number	D	7
			DD	07
			DDD	007
			DDD	123
F	Day of particular week in month (1-7). Use with w (uppercase) for week in month. See “Week in month” later in this table.	Number	See example for “Week in month”.	
w (lowercase)	Week in year. Use with uppercase Y.	Number	The first example below uses uppercase transforming	Suppose you are

Symbol	Meaning	Presentation	Example	Sample Result
	<p>The first week of a month/year is defined as the earliest seven day period beginning on the specified "first day of the week" and containing at least the specified "minimal number of days in the first week" of that month/year.</p> <p>Note that the first day of the week and the minimum days in the first week (1 to 7) are dependent upon the calendar in use (which depends upon the locale resource data).</p> <p>For example, January 1, 1998 was a Thursday.</p> <p>If the first day of the week is a Monday and the minimum days in a week is 4 (these are the values reflecting ISO 8601 and many national standards), then week 1 of 1998 starts on December 29, 1997, and ends on January 4, 1998. However, if the first day of the week is a Sunday and the minimum number of days in a week is 4, then week 1 of 1998 starts on January 4, 1998, and ends on January 10, 1998. The first three days of 1998 are then part of week 53 of 1997.</p>		<p>y. The second example shows the difference when you use lowercase y.</p> <p>"'Week' w YYYY"</p> <p>"'Week' w yyyy"</p>	<p>December 31st, 2008, which is a Wednesday (results for the US locale).</p> <p>"Week 1 2009"</p> <p>"Week 1 2008"</p>

Symbol	Meaning	Presentation	Example	Sample Result
w (uppercase)	Week in month. The values are calculated similar to "Week in year" (w). Weeks numbered with a minus sign (such as -2 or -1) and 0 precede the first week. Weeks numbered 2, 3 and so on follow the first week.	Number	"'Day' F 'of Week' W"	"Day 2 of Week 3"
a	AM/PM marker	Text	a a	AM PM
k	Hour in day (1-24)	Number	k kk kk	1 01 24
K	Hour in AM/PM (0-11)	Number	K KK KK	0 07 11
z (lowercase)	Time zone	Text	z, zz, zzz zzzz	GMT+05:30 Pacific Standard Time
Z (uppercase)	Time zone (RFC 822)	Number	Z	-0800
v (lowercase)	Generic time zone	Text	v vvvv	PT (assuming US locale) Pacific Time
V (uppercase)	Short time zone ID	Text	V	gblon
VV	Long time zone ID	Text	VV	Europe/London
VVV	Time zone exemplar city	Text	VVV	London
VVVV	Time zone location text	Text	VVVV	United Kingdom Time
O (uppercase)	Localized GMT time zone	Text	O O000	GMT-8 GMT-8:00

Symbol	Meaning	Presentation	Example	Sample Result
x (uppercase)	ISO8601 time zone with Z	Text	x	-08, +0530, Z
			xx	-0800, Z
			xxx	-08:00, Z
			xxxx	-0800, -075258, Z
			xxxxx	-08:00, -07:52:58, Z
x (lowercase)	ISO8601 time zone without Z	Text	x	-08, +0530
			xx	-0800
			xxx	-08:00
			xxxx	-0800, -075258
			xxxxx	-08:00, -07:52:58
g	Julian day	Number	g	2451334
A	Milliseconds in day	Number	A	69540000
'	Escape for text	Delimiter	"'Week' w YYYY"	"Week 1 2009"
''	Single quote	Literal	"KK 'o'clock'"	"11 o'clock"

Any character in the `format` pattern that is not in the range of `[a..'z']` or `[A..'Z']` is treated as quoted text. For example, the following characters can be in a `timeDate` string without being enclosed in quotation marks:

:
.
,

@

A pattern that contains an invalid pattern letter results in a -1 return value.

The following table gives examples that assume the US locale:

Format pattern	Suitable <code>timeDate</code> string
yyyy.MM.dd G 'at' HH:mm:ss z	1996.07.10 AD at 15:08:56 GMT+08:30
EEE, MMM d, ''yy	Wed, July 10, '96

Format pattern	Suitable timeDate string
h:mm a	12:08 PM
hh 'o'clock' a, zzzz	12 o'clock PM, Pacific Daylight Time
K:mm a, z	0:00 PM, GMT+07:30
yyyyy.MMMM.dd GGG hh:mm aaa	1996.July.10 AD 12:08 PM

When parsing a date string using the abbreviated year pattern (y or yy), `SimpleDateFormat` (and hence all parse functions) must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 79 years before and 19 years after the time the `SimpleDateFormat` instance is created. For example, using a pattern of MM/dd/yy and a `SimpleDateFormat` instance created on Jan 1, 1997, the string 01/11/12 would be interpreted as Jan 11, 2012 while the string 05/04/64 would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, as defined by `Unicode::isDigit()`, will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that is not all digits (for example, -1), is interpreted literally. So 01/02/3 or 01/02/003 are parsed, using the same pattern, as Jan 2, 3 A.D. Likewise, 01/02/-3 is parsed as Jan 2, 4 B.C. Behavior is undefined if you specify a two-digit date that might be either twenty years in the future or eighty years in the past.

If the year pattern has more than two y characters, the year is interpreted literally, regardless of the number of digits. So using the pattern MM/dd/yyyy, 01/11/12 parses to Jan 11, 12 A.D.

When numeric fields abut one another directly, with no intervening delimiter characters, they constitute a run of abutting numeric fields. Such runs are parsed specially. For example, the format HHmmss parses the input text 123456 to 12:34:56, parses the input text 12345 to 1:23:45, and fails to parse 1234. In other words, the leftmost field of the run is flexible, while the others keep a fixed width. If the parse fails anywhere in the run, then the leftmost field is shortened by one character, and the entire run is parsed again. This is repeated until either the parse succeeds or the leftmost field is one character in length. If the parse still fails at that point, the parse of the run fails.

For time zones that have no names, `SimpleDateFormat` uses strings GMT+hours:minutes or GMT-hours:minutes.

The calendar defines what is the first day of the week, the first week of the year, whether hours are zero based or not (0 vs. 12 or 24), and the time zone. There is one common number format to handle all the numbers; the digit count is handled programmatically according to the pattern.

Midnight and noon

The format "HH:mm" parses "24:00" as midnight that ends the day. Given the format "hh:mm a", both "00:00 am" and "12:00 am" parse as the midnight that begins the day. Note that "00:00 pm" and "12:00 pm" are both midday.

Using the MemoryStore

The MemoryStore provides an in-memory, table-based, data storage abstraction within the correlator. All EPL code running in the correlator in any context can access the data stored by the MemoryStore. In other words, all EPL monitors running in the correlator have access to the same data.

The Apama MemoryStore can also be used in a distributed fashion to provide access to data stored in a MemoryStore to applications running in a cluster of multiple correlators. For more information on the distributed MemoryStore, see [“Using the distributed MemoryStore” on page 365](#).

The MemoryStore can also store data on disk to make it persistent, and copy persistent data back into memory. However, the MemoryStore is primarily intended to provide all monitors in the correlator with in-memory access to the same data.

Use the MemoryStore to share data among monitors in the correlator or to persist data on disk. If the situations listed below apply to you, the standard Apama ADBC (Apama Database Connector) adapter is likely to be a better option for you than the MemoryStore.

- You want to interoperate directly with data users other than Apama.
- You need access to more data than can fit in memory.
- You need to key on more than one field.
- You want to join tables.

See also [“Using the MemoryStore when persistence is enabled” on page 323](#).

See "The Database Connector IAF Adapter (ADBC)" in *Connecting Apama Applications to External Components*.

For details about the event types that provide the MemoryStore interface, see the *API Reference for EPL (ApamaDoc)*.

Introduction to using the MemoryStore

Data that the MemoryStore stores must be one of the following types: `boolean`, `decimal`, `float`, `integer` or `string`.

To use the MemoryStore, you need to add the MemoryStore bundle to your Apama project (see [“Adding the MemoryStore bundle to your project” on page 351](#)). This lets you create instances of MemoryStore events and then call actions on those events. Available actions include the following:

- Creating stores that contain tables
- Defining the schema for the rows in a table
- Creating tables and associating a schema with each table
- Storing, retrieving, updating, and committing rows of data

- Copying tables to disk to make the data persistent
- Making stored data available in DataViews for use by dashboards

You can use the MemoryStore in parallel applications. You can use the MemoryStore in a persistent monitor in a persistence-enabled correlator. See [“Using the MemoryStore when persistence is enabled” on page 323](#).

For information on using the MemoryStore in a distributed fashion, see [“Using the distributed MemoryStore” on page 365](#).

Overview of MemoryStore events

The MemoryStore defines the following events in the `com.apama.memorystore` package. Most of these events contain action fields that serve as the MemoryStore interface.

- **Storage** — The event type that provides the interface for creating stores.
- **Store** — A Store event represents a container for a uniquely named collection of tables.
- **Table** — A Table event represents a table in a store. A table is a collection of rows. Each table has a unique name within the store. A table resides in memory and you can store it on disk if you want to.
- **Schema** — A Schema event specifies a set of fields and the type of each field. Each Schema event represents the schema for one or more tables. Each table is associated with one schema. All rows in that table match the table's schema.
- **Row** — A Row event represents a row in a table. A row is an ordered and typed set of named fields that match the schema associated with the table that the row belongs to. Each row is associated with a string that acts as its key within the table. You can change the values of the fields in a row.
- **Iterator** — Provides the ability to manipulate each row of a table in turn.
- **Finished** — The MemoryStore enqueues a Finished event when processing of an asynchronous action is complete.
- **RowChanged** — The RowChanged event is used only in a distributed store. In a distributed store, the RowChanged event is sent to all applications that have subscribed to a specific table whenever changes to data in a row in that table have been successfully committed. This behavior is optional and is supported by some, but not all, third-party distributed cache providers.

For details about these events, see the information for MemoryStore in the *API Reference for EPL (ApamaDoc)*.

Adding the MemoryStore bundle to your project

To use the MemoryStore, you need only add the MemoryStore bundle to your project. The description below explains how to add the bundle using Software AG Designer, but you can also add it using the `apama_project` command-line tool as described in "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*.

Note:

To use the distributed MemoryStore, you add the Distributed MemoryStore adapter instead. The procedure for this is different and is described in [“Adding distributed MemoryStore support to a project” on page 369](#).

Adding the MemoryStore bundle to your project makes the `MemoryStore.mon` file available to the monitors in your project. When you run your project, Software AG Designer automatically injects `MemoryStore.mon`. If you want to examine this file, it is in the `monitors/data_storage` directory of your Apama installation directory. `MemoryStore.mon` is the interface between the monitors in your application and the MemoryStore plug-in. Your application creates events of the types defined in that file and calls actions on those events to use the MemoryStore's facilities. There is never any need to import or call the plug-in directly.

Note:

If you use the `engine_inject` tool to manually inject your EPL, instead of using Software AG Designer, and you want to expose MemoryStore tables to dashboards, you need to inject the `MemoryStoreScenarioImpl.mon` file, which is in the same directory as the `MemoryStore.mon` file.

➤ To add the MemoryStore bundle

1. In Software AG Designer, open the project in the Apama Developer perspective.
2. In the **Project Explorer**, right-click the **EPL Bundles** node and select **Add Bundle**.
3. In the Add Bundle dialog, select the **The MemoryStore** bundle and click **OK**.

Steps for using the MemoryStore

To use the MemoryStore, you must first add the **The MemoryStore** bundle to your project, unless you are using the distributed MemoryStore. (If you are using the distributed MemoryStore, instead of adding the **The MemoryStore** bundle, you need to add the **Distributed MemoryStore** adapter. For more information on this, see [“Adding distributed MemoryStore support to a project” on page 369](#).)

After you have added the MemoryStore bundle, you write EPL that does the following:

1. Prepare and then open a store that will contain one or more tables.
2. Define the data schema for the rows that will belong to the table.
3. Prepare and then open a table in a store.
4. For applications that will access data in a distributed store, if the underlying third-party distributed cache provider supports notifications, optionally subscribe to the table in order to receive notifications when data has changed. For further information, see [“Notifications” on page 372](#).
5. Get a new or existing row from the table.
6. Modify the row.

7. Commit the modified row to the table.
8. Repeat the three previous steps as often as needed.
9. Optionally, use an iterator to step through all rows in the table.
10. Optionally, store the in-memory table on disk.

Preparing and opening stores

The `Storage` event is used to prepare and open a store to which you can add tables. Storage events define actions that do the following:

- Request preparation of a store.
- Open a store that has been prepared.

Storage events contain no data. All Storage events are alike and exist only to provide the interface for preparing and opening stores. All actions on the `Storage` event are static; there is no need to create an instance of a `Storage` event.

If you do not require on-disk persistence, you can prepare a store in memory. If you do require on-disk persistence, you can specify the file that contains (or that you want to contain) the store. Depending on the action you call to open the store, the `MemoryStore` does one of the following:

- Opens the store for read-write access.
- Opens the store for read-only access.
- Opens the store for read-write access. Create the store if it does not already exist.

Preparation of stores is asynchronous. Actions that prepare stores return an ID immediately. When the `MemoryStore` completes preparation it enqueues a `Finished` event that contains this ID. You should define an event listener for this `Finished` event. The `Finished` event indicates whether or not preparation was successful.

You can open a store only after receiving a `Finished` event that indicates successful preparation.

For example, the following code fragment declares a `Storage` type variable and a `Store` type variable. It then calls the `prepareOrCreate()` action on the `Storage` type variable and saves the returned ID in the `Store` type variable. The name of the new store is `storename` and the store will be made persistent by saving it in the `example.dat` file. Finally, this code fragment declares a `Finished` event variable and an event listener for a `Finished` event whose ID matches the ID returned by the preparation request.

```
using com.apama.memorystore.Storage;
using com.apama.memorystore.Store;
using com.apama.memorystore.Finished;

monitor Test {
    Store store;

    action onload() {
        integer id := Storage.prepareOrCreate("storename", "/tmp/example.dat");
        on Finished(id,*,*)as f
```

```
    onStorePrepared(f);  
    ...  
}  
}
```

After a store has been successfully prepared, you can open it:

```
action onStorePrepared(Finished f) {  
    if not f.success { log "Whoops"; die; }  
    store := Storage.open("storename");  
}
```

All subsequent examples assume that the appropriate using statements have been added.

Any monitor instance can open a store after that store has been successfully prepared. However, monitor A has no information about whether or not monitor B has prepared a particular store.

Therefore, each monitor should prepare any store it needs, and then prepare any tables it needs within that store. There is no way to pass `Store` or `Table` events from one monitor to another. Multiple monitors can prepare and open the same store or table at the same time.

There are several different actions available for preparing a store:

- `Storage.prepareInMemory(string name)` returns integer prepares an in-memory store with the name you specify. All tables are empty when prepared for the first time. Persistence requests are ignored and immediately return a successful `Finished` event.
- `Storage.prepare(string name, string filename)` returns integer does the same thing as `Storage.prepareInMemory` and it also associates that store with the database file you specify. If there is data in the database file the `MemoryStore` loads the store with the data from the file when you prepare a table. Persistence requests write changes back to the file. The specified file must exist.
- `Storage.prepareOrCreate(string name, string filename)` returns integer does the same thing as `Storage.prepare()` except that it creates the file if it does not already exist.
- `Storage.prepareReadOnly(string name, string filename)` returns integer does the same thing as `Storage.prepare` and it also opens for read-only access the database file you specify. The `MemoryStore` will load the store with data from the file when you prepare the table. Persistence requests are refused and return a failure `Finished` event
- `Storage.prepareCorrelatorPersistent(string name)` returns integer prepares a store that the correlator automatically persists. Each time the correlator takes a snapshot, the snapshot includes any correlator-persistent stores along with the contents of those stores.
- `Storage.prepareDistributed(string name)` returns integer prepares a distributed store which will be available to applications running in a cluster of correlators. The *name* argument is a unique identifier that specifies the name of a configured distributed store. For information on adding a distributed store to a project, see [“Adding a distributed store to a project” on page 369](#).

Suppose a monitor instance calls one of the `Storage.prepare()` actions and the action is successful. Now suppose another monitor instance calls the same `Storage.prepare()` variant with the same table name and, if applicable, the same filename, as the previously successful call. The second call does nothing and indicates success immediately. However, if a monitor instance makes a `Storage.prepare()` call and specifies the same table name as was specified in a previously successful

`prepare()` call, that call fails immediately if at least one of the following is different from the successful call:

- The variant of the `prepare()` action called
- The specified file name or store name (if applicable)

For example, suppose a monitor made the following successful call:

```
Storage.prepare("foo", "/tmp/foo.dat")
```

After this call, the only `prepare` call that can successfully prepare the same table is

```
Storage.prepare("foo", "/tmp/foo.dat")
```

The following calls would all fail:

```
Storage.prepareInMemory("foo")
Storage.prepareOrCreate("foo", "/tmp/foo.dat")
Storage.prepareReadOnly("foo", "/tmp/foo.dat")
Storage.prepare("foo", "/tmp/bar.dat")
```

If a monitor makes a call to `prepare()` that matches a `prepare` action that is in progress, the result is the same as the result of the `prepare` that is in progress.

Description of row structures

Schemas

A schema consists of an ordered list of the names and types of fields that define the structure of a row. For example, the following schema consists of one field whose name is `times_run` and whose type is `integer`:

```
Schema schema := new Schema;
schema.fields := ["times_run"];
schema.types := ["integer"];
```

A valid schema can be created from an event type using `schemaFromAny(event)`. Types that are not supported in the event are converted to `string` types.

The `Schema` event has additional members that indicate how to publish the table. See [“Exposing in-memory or persistent data as DataViews” on page 363](#).

The schema does not include the row's key. The key is always a string and it does not have a name. Each row in a table is associated with a key that is unique within the table. The key provides a handle for obtaining a particular row. The row does not contain the key.

Two schemas match when they list the same set of field names and types in the same order and choose the same options for exposing `DataViews`.

Some distributed `MemoryStore` drivers (such as `TCStore`) support getting and setting extra fields that are present in only some individual rows, and are not named in the schema. For stores supporting this feature, it is even possible to specify an empty list of schema fields and access all fields as extra fields if desired. When getting extra fields, it is important to be aware that getting

a field that does not exist will result in an exception, so it is usually necessary to add exception handling code, or to check which keys are present in the row (using `Row.getKeys()`) before attempting to access them. It is also possible to use the `Row.getAll` or `Row.toDictionary` actions to get all fields including those named in the schema and any extra fields that are present. Note that `RowChanged` notifications are not supported for extra fields.

Tables

`Table` events define actions that do the following:

- Retrieve a row by key. The returned object is a `Row` event.
- Remove a row by key
- Remove all rows
- Obtain a sequence of keys for all rows in the table
- Obtain an iterator to iterate over the rows in the table
- Determine if any row in the table has a particular key
- Store on disk the changes to the in-memory table
- Subscribe (and unsubscribe) to a table to be notified when a row has changed. (Note, this is only supported for tables in a distributed store, and only if the underlying provider supports this feature.)
- Modify a row by key
- Modify all rows
- Obtain the position in a schema of a specified field.
- Obtain the name of the table
- Obtain the name of the store that contains the table

For details about these `Table` event actions, see the information for `MemoryStore` in the *API Reference for EPL (ApamaDoc)*.

Retrieval of a row from a table by key always succeeds (although retrieving a row from a table in a distributed store can throw an exception). If the row already exists, the `MemoryStore` returns a `Row` event that provides a local copy of the row. The content of this `Row` event does not change if another user modifies the in-memory version of the row in the table. If the row does not already exist, the `MemoryStore` populates a `Row` event with default values and returns that with field values as follows:

- `boolean` types are `false`
- `decimal` types are `0.0d`
- `float` types are `0.0`
- `integer` types are `0`

- string types are empty ("")

Rows

Row events define actions that do the following:

- Getters and setters. These actions modify only the local copy (your Row event) and not the in-memory version of the row. The in-memory version of the row is available to all monitors. If another user of the table retrieves the same row, that user receives a Row event that contains a copy of the in-memory version of the row; that user does not receive a copy of your modified, local version of the row:
 - Get and set boolean, decimal, float, integer, and string fields by name.
 - Generic get and set field by name actions which use the any type. These throw an exception if the underlying types do not match the expected field type.
 - Get and set all fields. These expect a prototype event whose fields and types match that of the table schema. An exception is thrown if the schemas do not match.
- Commit a modified Row event. That is, you modify your local Row event, and commit the changes, which updates the shared row in the table. This makes the update available to all monitors.
- Get the value of a row's key.
- Determine whether a row was present in the table when the local copy was provided.
- Obtain the name of the table the row is in.
- Obtain the name of the store the row's table is in.

The `Row.commit()` action modifies only the in-memory copy of the row so it is a synchronous and non-blocking operation. Note, in a distributed store, `Row.commit()` writes the value to the distributed store, which may be a fast, local operation or it may involve writing data to one or more remote nodes. If any other user of the table modifies the in-memory row between the time you obtain a Row event that represents that row and the time you try to commit your changes to your Row event, the `Row.commit()` action fails and the monitor instance that called `Row.commit()` dies. Therefore, if you are sharing the table with other users or using a distributed store, you should call `Row.tryCommit()` instead of `Row.commit()`. If it fails you must retry the commit operation by retrieving the row again (that is, obtaining a new Row event that contains the latest content of the in-memory row), reapplying the changes, and then calling the `Row.tryCommit()` action. This ensures that you always make changes that are consistent and atomic within the shared version of the row.

However, it is not possible to make atomicity guarantees across rows or tables.

Preparing and opening tables

After you have an open store, you can add one or more tables to that store. You call actions on Store events to create tables. Store events define actions that do the following:

- Prepare a table. You specify a table name and a schema or supply an event or type name to use as the name and schema. This call is asynchronous. The `MemoryStore` enqueues a `Finished` event that indicates success or failure. If the table does not exist, the `MemoryStore` creates an empty table.
- Open a table that has been prepared
- Store on disk the in-memory changes to tables.

If the store that contains the table is persistent and the table exists on disk then the on-disk schema must match the schema that you specify when you call the action to prepare the table. The schemas must also match if the table is a distributed table that already exists in a distributed store. If the schemas do not match, the `Finished` event that the `MemoryStore` enqueues includes an error message.

Note:

A persistent table can be an on-disk table or a table in a correlator-persistent store.

If a monitor instance calls `Store.prepare()` with the same table name and schema as those of a previously successful `Store.prepare()` call, the call does nothing and indicates success immediately. If a monitor instance calls `Store.prepare()` and specifies the same table name but the schema does not exactly match, that call fails immediately. If a monitor makes a call to `Store.prepare()` that matches a preparation that is in progress, the result is the same as the result of the preparation that is in progress.

If the table you want to prepare is persistent and it has not yet been loaded into memory then the `MemoryStore` loads the table's on-disk data into memory in its entirety. The `MemoryStore` enqueues the `Finished` event when loading the table is complete.

To use a table that is in memory, you must retrieve a handle to it from the store that contains it. Obtaining a handle to a prepared (loaded) table is a synchronous action that completes immediately and does not block. The calling monitor instance dies if you try to obtain a handle to a table that is not prepared or that is in the process of being prepared.

For example:

```
integer id := store.prepare("tablename", schema);
on Finished(id,*,*) as f onTablePrepared(f);

action onTablePrepared(Finished f) {
    if not f.success { log "Whoops"; die; }
    Table tbl := store.open("tablename");
}
```

Note:

The term “table” is a reserved keyword. Consequently, you should not use “table” as a variable name.

Preparation of a table can fail for a number of reasons including, but not limited to, the following:

- You call `prepare()` on an existing table and the schema of that table and the schema specified in the `prepare()` call do not match.

- You call `prepare()` on an existing in-memory table and the `exposePersistentView` setting is `true` for the schema you specify in the `prepare()` call.
- You call `prepare()` on a table that does not exist and the store has been opened read-only.
- You call `prepare()` on a table that does not exist in a persistent store and the attempt to create a new table in the persistent store fails, perhaps because the disk is full.
- The on-disk version of the table is corrupt in some way.
- You set `exposePersistentView` on a table in a correlator-persistent store.
- You set `exposeMemoryView` or `exposePersistentView` to `true` for a distributed store.
- The third-party distributed store implementation throws an exception for some reason such as unrecoverable network failure.

Using transactions to manipulate rows

In a monitor, any changes you make to `Row` events are local until you commit those changes. In other words, any changes you make actually modify the `Row` events that represent the actual rows in the table. After you commit the changes you have made to your `Row` events, the updated rows are available to all monitors in the correlator and to all other members of the distributed cluster if you are using a distributed store.

Note:

When you modify a `Row` event and you want to update the actual row with your changes, you must commit your changes. It does not matter whether or not the table is in a correlator-persistent store.

The `Row` event defines the following actions for committing changes:

- `Row.commit()` — Tries to commit changes to `Row` events to the table. If nothing else modified the row in the table since you obtained the `Row` event that represents that row, the `MemoryStore` commits the changes and returns. The update is available to all monitors. If the row in the table has been modified, an exception is thrown, leaving the table unchanged.
- `Row.tryCommit()` — Behaves like `commit()` except that it does not throw an exception upon failure. If the row in the table has been modified, this action returns `false` and leaves the table unchanged. If this action is successful, it returns `true`.
- `Row.tryCommitOrUpdate()` — Behaves like `tryCommit()` except that when it returns `false`, it also updates your local `Row` event to reflect the current state of the actual row in the table. In other words, if the row in the table has been modified, this action does the following:
 - Leaves the actual row in the table unchanged.
 - Updates the local `Row` event that represents this row to reflect the current state of the table. Any local, uncommitted modifications are lost.
 - Returns `false`.

- `Row.forceCommit()` — Commits the local Row event to the table even if the row in the table has been modified after you obtained the Row event.

Determining which commit action to call

If you are certain that you are the only user of a table and if it is okay for your monitor instance to be killed if you are wrong, you can use `commit()`.

If you want to use a simple loop like the one below, or if you intend to give up if your attempt to commit fails, then use `tryCommit()`.

```
boolean done := false;
while not done {
    Row row := tbl.get("foo");
    row.setInteger("a",123);
    done := row.tryCommit();
}
```

However, the loop above calls `tbl.get()` every time around. If you think there might be a high collision rate, it is worth optimizing to the following, more efficient design:

```
Row row := tbl.get("foo");
boolean done := false;
while not done {
    row.setInteger("a",123);
    done := row.tryCommit();
    if not done { row.update(); }
}
```

The `tryCommitOrUpdate()` action makes the example above a little simpler and considerably more efficient:

```
Row row := tbl.get("foo");
boolean done := false;
while not done {
    row.setInteger("a",123);
    done := row.tryCommitOrUpdate();
}
```

Alternatively, there is a packaged form of that loop that you might find more convenient:

```
action doSomeStuff(Row row) {
    row.setInteger("a",123);
}
tbl.mutate("foo", doSomeStuff);
```

The above example is equivalent to the previous one, both in behavior and performance. Which to use is a matter of context, style and personal preference.

If you want to simply overwrite the whole Row rather than updating the row based on the current value, then using `forceCommit()` would be more appropriate. It commits the local Row content to the table even if it was modified after you obtained the Row event:

```
Row row := tbl.get("foo");
row.setInteger("a", 123);
row.forceCommit()
```


Creating and removing rows

To create a row in a table, call the `get()` or `add()` action on the table to which you want to add the row. The action declaration for the `get()` action is as follows:

```
action get(string key) returns Row
```

The `Table.get()` action returns a `Row` event that represents the row in the table that has the specified key. If there is no row with the specified key, this action returns a `Row` event that represents a row that contains default values. A call to the `Row.inTable()` action returns `false`. For example:

```
boolean done := false;
integer n := -1;
while not done {
    Row row := tbl.get("example-row");
    n := row.getInteger("times_run");
    row.setInteger("times_run", n+1);
    done := row.tryCommit();
}
send Result(
    "This example has been run " + n.toString() + " time(s) before"
    to "output";
```

The `add()` action does the same as the `get()` action, except that it does not check if the row that is to be added already exists in the table until `commit()` is called and it therefore never throws an exception. If you are sure that the row does not yet exist, you can use `add()` as this is faster than `get()`.

To remove a row from a table, call the `Table.remove()` action on the table that contains the row. The action declaration is as follows:

```
action remove(string key)
```

The `Table.remove()` action removes the row with the specified key from the table. If the row does not exist, this action does nothing.

It is also possible to remove a row transactionally, by calling `Table.get()` and then `Row.remove()` and `Row.commit()`. This strategy lets you check the row's state before removal. The `Row.commit()` action fails if the shared, in-memory row has been updated since the `Table.get()` action.

In some circumstances, using `Row.remove()` is essential to guarantee correctness. For example, when decrementing a usage counter in the row and removing the row when the count reaches zero. Otherwise, another correlator context might re-increment the count between it reaching zero and the row being removed.

Iterating over the rows in a table

Iterators have operations to step through the table and determine when the end has been reached. Provided an iterator is not at the table's end, the key it is at can be obtained.

Iterator events define actions that do the following:

- Step through the rows in a table.

- Determine when the last row has been reached.
- Obtain the key of the row that the iterator is at. The iterator must not be at the end of the table for this action to be successful.
- Obtain a Row event to represent the row that the iterator is at.

The following sample code reads table content:

```
Iterator i := tbl.begin();
while not i.done() {
    Row row := i.getRow();
    if row.inTable() {
        // Put code here to read the row in the way you want.
    }
    i.step();
}
```

The following sample code modifies table content:

```
Iterator i := tbl.begin();
while not i.done() {
    Row row := i.getRow();
    boolean done := false;
    while row.inTable() and not done {
        // Put code here to modify the row in the way you want.
        done := row.tryCommitOrUpdate();
    }
    i.step();
}
```

Iterating through a table is always safe, regardless of what other threads are doing. However, if another context adds or removes a row while you are iterating in your context, it is undefined whether your iterator will see that row.

Furthermore, it is possible for another context to remove a row while your iterator is pointing at it. If this happens, a subsequent `Iterator.getRow()` returns a Row event that represents a row for which `Row.inTable()` is false.

If an EPL action loops, the correlator cannot perform garbage collection within that loop. (See [“Optimizing EPL programs” on page 418](#).) Performing intricate manipulations on many rows of a large table could therefore create so many transitory objects that the correlator runs out of memory. If this becomes a problem, you can divide very large tasks into smaller pieces, each of which is performed in response to a routed event. This gives the correlator an opportunity to collect garbage between delivering successive events.

Requesting persistence

After changing a MemoryStore table, you can call the `Table.persist()` action to store the changes on disk. Note that you can call `persist()` only on tables in an on-disk store; you cannot call `persist()` on tables in correlator-persistent, in-memory, or distributed stores. The correlator automatically persists correlator-persistent stores and their contents at the same time as the rest of the correlator runtime state. Updating a table on disk is an asynchronous action. The MemoryStore enqueues a `Finished` event to indicate success or failure of this action. The persistent

form of the database that contains the tables is transactional. Consequently, if there is a hardware failure either all of the grouped changes are made or none of them are made.

Following is an example of storing a table on disk:

```
integer id := tbl.persist();
on Finished(id,*,*) as f onPersisted(f);

action onPersisted(Finished f) {
    if not f.success { log "Whoops"; die; }
    emit "All OK";
}
```

When you update a table, the `MemoryStore` copies only the changes to the on-disk table.

To improve performance, the `MemoryStore` might group persistence requests from multiple users of a particular store. This means that calling `persist()` many times in rapid succession is efficient, but this does not affect correctness. If the `MemoryStore` indicates success, you can be certain that the state at the time of the `persist()` call (or at the time of some later `persist()` call) is on disk.

You can call the `Store.backup()` action to backup the on-disk form of a store while it is open for use by the correlator. This is an asynchronous action that immediately returns an ID. The `MemoryStore` enqueues a `Finished` event that contains this ID to indicate success or failure of this action. Be sure to define an event listener for this event.

Exposing in-memory or persistent data as DataViews

You can expose committed in-memory data or committed persistent data as `DataViews` for use by Scenario Service clients such as dashboards (see "Scenario Service API" in *Connecting Apama Applications to External Components*). Note, however that is not supported for distributed stores. The Schema event defines the following fields for this purpose:

- `exposeMemoryView` — When this field is `true`, the `MemoryStore` makes the rows in the in-memory table associated with this schema available to Apama's Scenario Service. That is, the `MemoryStore` creates `DataViews` that contain this data.
- `exposePersistentView` — When this field is `true`, the `MemoryStore` makes the rows in the on-disk table associated with this schema available to Apama's Scenario Service. That is, the `MemoryStore` creates `DataViews` that contain this data. You cannot expose a persistent view of a table in a correlator-persistent store.
- `memoryViewDisplayName` — Specifies the display name for the exposed `DataView` created from the in-memory table.
- `memoryViewDescription` — Specifies the description for the exposed `DataView` created from the in-memory table.
- `persistentViewDisplayName` — Specifies the display name for the exposed `DataView` created from the on-disk table.
- `persistentViewDescription` — Specifies the description for the exposed `DataView` created from the on-disk table.

The MemoryStore exposes in-memory changes after successfully committing them to the table. The MemoryStore exposes on-disk changes after the transaction that contains the changes is committed.

The `exposeMemoryView` and `exposePersistentView` fields have an impact on the time it takes to prepare a table for the first time. When a table is prepared the rows that are loaded from disk need to be reflected to the Scenario Service.

If you prepare the same table multiple times the display names and descriptions must match or the MemoryStore rejects the contradicting request.

When a display name or description field is blank (an empty string), the MemoryStore chooses the display name or the description for the exposed DataView. You can specify a non-empty string for one or more fields to override the default. Leave the display name and description fields blank when you are not exposing the corresponding DataView.

The fields of the exposed views are the same as those of the table, in the same order as they are defined in the table schema. The key is not part of the exposed views. Each row in the table forms a single exposed view.

See [“Making Application Data Available to Clients” on page 409](#). See also *Building and Using Apama Dashboards*.

Monitoring status for the MemoryStore

The MemoryStore provides status values via the user status mechanism. It provides the following metrics:

- `memStoreCommitTimeEwmaLongMillis` — A longer-term exponentially-weighted moving average (EWMA) of the time in milliseconds taken to commit to the MemoryStore (in-memory store and on-disk persistence).
- `memStoreCommitTimeEwmaShortMillis` — A quickly-evolving exponentially-weighted moving average (EWMA) of the time in milliseconds taken to commit to the MemoryStore (in-memory store and on-disk persistence).
- `memStoreCommitTimeMaxInLastHour` — The maximum commit latency in milliseconds since the start of the last 1 hour measurement period for the MemoryStore (in-memory store and on-disk persistence).
- `distMemStoreCommitTimeEwmaLongMillis` — A longer-term exponentially-weighted moving average (EWMA) of the time in milliseconds taken to commit to the distributed MemoryStore.
- `distMemStoreCommitTimeEwmaShortMillis` — A quickly-evolving exponentially-weighted moving average (EWMA) of the time in milliseconds taken to commit to the distributed MemoryStore.
- `distMemStoreCommitTimeMaxInLastHour` — The maximum commit latency in milliseconds since the start of the last 1 hour measurement period for the distributed MemoryStore.

The above `MaxInLastHour` values (for both the MemoryStore and the distributed MemoryStore) are updated if either of the following conditions is true:

- The latency of current commit transaction is greater than the existing maximum.
- The existing maximum value was set more than 1 hour ago.

For more information about monitor status information published by the correlator, see "Managing and Monitoring over REST" and "Watching correlator runtime status", both in *Deploying and Managing Apama Applications*.

When using Software AG Command Central to manage your correlator, see also "Monitoring the KPIs for EPL applications and connectivity plug-ins" in *Deploying and Managing Apama Applications*.

Restrictions affecting MemoryStore disk files

At any one time, only one correlator should be accessing a particular MemoryStore disk file.

To minimize the risk of data corruption in the event of a system failure, keep MemoryStore files on your local disk and not on a remote file server.

Do not create hard or symbolic links to MemoryStore files. Linking to the directory that contains a MemoryStore file is not a problem.

Using the distributed MemoryStore

With a distributed MemoryStore, you can access data shared among Apama applications running in separate correlators. Distributed stores make use of distributed caching software from a variety of third-party vendors.

The topics below describe typical use cases for the distributed MemoryStore, how to add and configure distributed stores, and how to write drivers for integrating with third-party caching software.

Overview of the distributed MemoryStore

The MemoryStore supports several types of stores as described in [“Using the MemoryStore” on page 350](#). In addition to those stores that are local to a single Apama process, Apama also supports a *distributed* store in which data can be accessed by applications running in multiple correlators. You prepare a distributed store with a `prepareDistributed` call on the `Storage` interface. When this sends a `Finished` event with `success` set to `true`, the `Store` can be opened, and `Table` objects created.

A distributed store makes use of Terracotta's `TCStore` or `BigMemory Max`, or a third-party distributed cache or datagrid technology that stores the data (table contents) in memory across a number of processes (nodes), typically across a number of machines. The collection of nodes is termed a *cluster*.

Advantages

Arranging a number of nodes into a cluster provides the following advantages:

- It is possible to store more data than would fit on one node.
- As the data is in memory, a distributed store is typically faster than persisting the store contents to disk.
- Every piece of data is typically stored on more than one node, so the failure of any one node should not cause the loss of any committed data.
- If a node fails, other nodes can access any of the data without waiting to “recover” or reload the entire datastore. Note, however, that it may take time to detect that the failed node is down.
- The number of correlators can be changed at runtime, allowing the processing capacity of the system to be increased.
- Different providers can be used, allowing a single Apama application to integrate with different distributed caches. However, each provider must have a driver. Apama provides a Service Programming Interface (SPI) with which you can write a custom driver (see [“Creating a distributed MemoryStore driver” on page 389](#) for more information).
- Data is accessible to multiple correlators. If they distribute workload appropriately, more processing capacity can use the same shared store of data. A distributed store is a building block for such a system, not a complete solution in itself.
- Applications can be notified of changes to data in the store (see [“Notifications” on page 372](#) for more information).

Disadvantages

A distributed store has the following disadvantages compared with the other types of store:

- A network request may be required to get or commit any Row. This is slower than the in-process local-memory get and commit requests made against local stores.
- The network request may fail because either more than one node has failed or there is a network failure such that the correlator cannot contact other nodes in the cluster.
- Multiple access to a single row will cause contention and will not scale (and will be slower than an in-memory store).
- It is not permitted to expose DataViews with a distributed store. A distributed store may contain a very large number of entries, which would not be practical to expose as DataViews (as it requires storing a copy of the entire table in the dashboards/Scenario Service client).

Use cases

Based on the advantages and disadvantages of distributed stores, the typical use cases for using them are:

- More data needs to be stored than will fit on any single node.
- Elastic (changing) processing capacity is required.

- A highly available system needs continuous access to data, even if some nodes fail, and with minimal recovery time.
- High throughput is required across a large number of different rows, with only a small amount of contention for a single row.

The typical use cases where a distributed store is not suitable are:

- Very low latency (sub-millisecond) access to data.
- Very high throughput (more than 10,000 requests per second) to a single row. The distributed store only scales out well if different rows are being accessed.

Supported providers

Apama includes drivers for connecting to Terracotta's TCStore and BigMemory Max, which provide unlimited in-memory data management across distributed servers. See [“TCStore \(Terracotta\) driver details” on page 378](#) and [“BigMemory Max driver details” on page 382](#) for more information.

Apama also provides an interface to integrate with third-party distributed caching software that provides compare-and-swap operations for adding, updating, and removing data (for example, software that provides methods similar to the `putIfAbsent`, `replace`, and `remove` operations on `java.util.concurrent.ConcurrentMap`).

For other distributed cache providers, you need to write a driver using the Apama Service Provider Interface (SPI) to serve as a bridge between the `MemoryStore` and the caching software. For information on creating a driver, see [“Creating a distributed MemoryStore driver” on page 389](#).

Configuration

In order to use a distributed `MemoryStore`, a set of configuration files must be created in your project and provided to the correlator. These configuration files typically come in pairs: a `storeName-spring.xml` file and a `storeName-spring.properties` file. Multiple pairs of files can be created and can make use of more than one distributed cache provider. See [“Configuring a distributed store” on page 369](#).

Distributed store transactional and data safety guarantees

The `commit()` action on a `Row` object from a distributed store by default behaves similarly to an in-memory store's `Row` object, in that the commit succeeds only if there have been no commits to the `Row` object since the most recent `get()` or `update()` of the `Row` object.

However, providers can be configured differently. For example, if using TCStore or BigMemory Max, and the `.properties` specifies `useCompareAndSwap` as `false`, then the commit will always succeed, even if another monitor committed a different value for that entry.

Unlike in-memory stores, for `Row` objects from a distributed store, a `Table.get()` or `Row.update()` may return an older value, that is, a previously committed value, even if a more recent commit has completed. This is because a distributed store may perform caching of data. After some undefined time, the `get()` should be eventually consistent; a later `get()` or `update()` of the `Row` object should retrieve the latest value. Typically, a commit of a `Row` object where the `get()` has not

retrieved the latest value will flush any local cache of the value, thus the first commit will fail, but a subsequent update and commit will succeed.

Again, providers can be configured differently. For the BigMemory Max driver, setting the `terracottaConfiguration.consistency` property to `STRONG` will ensure that after a `commit()`, a `get()` on any node will retrieve the latest version. This `STRONG` consistency mode is more expensive than `EVENTUAL` consistency.

An example: `Monitor1` gets and modifies a row and sends an EPL event to `Monitor2` which in response to the event gets and updates the row. In the table below, the event has “overtaken” the change to the row; the effects of changing the row and sending the event are observed in the reverse order (the event is seen before the change to the row).

Time	Monitor1 (on node 1)	Monitor2 (on node 2)
1	<code>Table.get("row1") = "abc"</code>	
1.2	Change row to be "abcdef"	<code>Table.get("row1") = "abc" (cached locally)</code>
1.3	<code>Row.commit("row1" as "abcdef")</code> succeeds	
1.301	Send event to node 2	
1.302		Receive event from node 1
1.303		<code>Table.get("row1") = "abc" from local cache)</code>
1.4		Update row to be "abcghi"
1.5		<code>Row.commit("row1 as "abcghi")</code> fails (not last value)
1.6		<code>Row.update() = "abcdef"</code>
1.7		Update row to be "abcdefghi"
1.8		<code>Row.commit("row1" as "abcdefghi")</code> succeeds

At 1.303, an in-memory cache (when two contexts are communicating in the same process) would be guaranteed to retrieve the latest value, "abcdef", but a distributed store may cache values locally. The commit is guaranteed to fail when a stale value is read, as it does not rely on cached values for checking whether the row is up to date or not.

Different providers may have other differences in behavior. In particular, they may differ in whether or not they use referential checking, that is, if one client reads a row, and meanwhile the row is modified but then modified back to the first value, the client with the old `Row` object may or may not succeed in performing a `commit()`. Some providers require the row to not have changed since the row was read (even if then changed back to the value at the point it was read), while others will only compare the contents of the row.

Also avoid relying on comparisons based on the `float` values `NaN` and `-0.0`. Some providers may treat `NaN` as not equal to any value, including a `NaN`, which will result in the `commit()` method

never being able to complete. Providers may differ in whether 0.0 and -0.0 are treated as the same value or not. Consider this to be undefined behavior.

Configuring a distributed store

Adding distributed MemoryStore support to a project

If you want to configure a distributed store, you first have to add Apama's **Distributed MemoryStore** adapter bundle to an Apama project.

The description below explains how to add the bundle using Software AG Designer, but you can also add it using the `apama_project` command-line tool as described in "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*.

➤ To add the Distributed MemoryStore adapter bundle to a project


1. In the Project Explorer, right-click the **Connectivity and Adapters** node and select **Add Connectivity and Adapters**.
2. Select **Distributed MemoryStore (Supports using a distributed cache from MemoryStore)** from the list of available adapters.
3. Click **OK**.

The adapter bundle is added to the project's **Connectivity and Adapters** node and the adapter instance is opened in the Distributed MemoryStore editor. The editor is initially blank and the **Distributed Stores** panel contains no distributed stores.

Adding a distributed store to a project

After the **Distributed MemoryStore** adapter bundle has been added, you can add a distributed store to the project.

➤ To add a distributed store to a project

1. In the Distributed MemoryStore editor's **Distributed Stores** panel, click the  **(Add Store)** button. The Distributed MemoryStore Configuration Wizard appears.
2. In the Distributed MemoryStore Configuration Wizard, specify the following:
 - a. From the **Store provider** drop-down list, select the third-party cache provider.

If you are using a driver supplied by Apama, select **TCStore (Terracotta)** or **BigMemory Max** from the drop-down list. Otherwise select **Others** from the drop-down list.

For queries, select **Apama Queries (using TCStore)** from the drop-down list.

- b. In the **Store name** field, specify the name of the store as it will be known in the configuration files and EPL code. The name must be unique and cannot contain spaces.

When you have selected **Apama Queries (using TCStore)**, the store name **ApamaQueriesStore** is automatically provided; this name cannot be changed in the wizard.

Note:

Support for using BigMemory Max for queries is deprecated and will be removed in a future release. It is recommended that you now use Terracotta's TCStore for queries by selecting **Apama Queries (using TCStore)** as the store provider. However, if you still want to use BigMemory Max for queries, select **BigMemory Max** as the store provider and specify "ApamaQueriesStore" as the store name.

3. Click **Finish**.


The name of the store is added to the **Distributed Stores** panel in the editor, and the resources for the store are added to the project. The default configuration settings for the store are displayed in the editor.

Configuring a distributed store


After a distributed store has been added to the project, you can configure it.

You can configure frequently-used settings for a distributed store in the Distributed MemoryStore editor. These settings are those in the `.properties` file. For other settings, you need to edit the `.xml` file directly.

➤ To configure a distributed store

1. Specify any provider-specific configuration options in either the **Standard properties** section of the editor or, for TCStore, in the **TCStore** sections of the editor.
2. In the **Classpath** section, specify the names of the required provider-specific `.jar` files. This is only required if **Other** was used as the store type (that is, if you are not using TCStore or BigMemory Max).
 - a. Click the  (**Add Classpath**) button. A new line is added to the location list.
 - b. In the new line, specify the name of the `.jar` file. When you specify the path to a `.jar` file, you should use substitution values rather than a full path name (for example, use `${installDir.mystore}/lib/my.jar`).
3. In the **Custom property substitution variables** section, specify the name and values of additional substitution variables (`${varname}`), if any, used by the distributed cache. The `.properties` file contains substitution variables that are used by the `.xml` configuration file.

You can define your own property substitution variables here, which will be written to the `.properties` file when you save. You can then edit the `.xml` file (see below) to use your own property substitution variables wherever you wish.

- a. Click the  (**Add Variable**) button. A new line is added to the list of substitution variables.
 - b. In the new line, specify the name and value of the substitution variable you want to add.
4. (If needed.) In the **Configuration files** section, you can access the Spring `.xml` and `.properties` files. Click on the file name links to open them in the appropriate editor.

For more information on specifying property values, see [“Configuration files for distributed stores” on page 373](#).

Launching a project that uses a distributed store

When you add the **Distributed MemoryStore** adapter bundle to an Apama project, the launch configuration is automatically updated to set the `--distMemStoreConfig` startup option.

In Software AG Designer, the maximum Java heap size and off-heap storage can be set in the Correlator Configuration dialog of the Run Configurations dialog. See “Adding a correlator” in *Using Apama with Software AG Designer* for more information on that dialog.

Interacting with a distributed store

Once prepared, a distributed store behaves much like other MemoryStore objects as described in [“Using the MemoryStore” on page 350](#). However, be aware of the following differences:

- The schema for tables in a distributed store is not allowed to expose DataViews.
- A distributed store (as opposed to other, non-distributed stores) supports notifications. For more information, see [“Notifications” on page 372](#).
- Exceptions. In an in-memory store, only the `Row.commit()` action can throw exceptions. However, in a distributed store, most actions can throw exceptions. An exception represents runtime error that can be caught with a `try ... catch` statement. This allows developers to choose what corrective action to take (such as logging, sending alerts, taking corrective steps, retrying later, or other actions). If no `try ... catch` block is used with these actions and an exception is thrown, the monitor instance will be terminated, the `ondie()` action will be called if one exists, and the monitor instance will lose all state and listeners. Exceptions can be thrown because of errors raised by third-party distributed cache providers. To discover what errors could be thrown because of third-party integration, you should refer to the documentation for the third-party provider in use. For more information on exceptions, see [“Exception handling” on page 276](#). The following are some of the actions that can throw exceptions:
 - `Table.get()`
 - `Table.begin()`
 - `Iterator.step()`

- `Row.commit()`
- `Row.update()`
- Performance differences. See [“Overview of the distributed MemoryStore” on page 365](#) for the advantages and disadvantages of using a distributed store as compared to an in-memory store.

Notifications

Distributed store `Table` objects may support the `subscribeRowChanged()` and `unsubscribe()` actions. If subscribed to a table, `RowChanged` events will be sent to that context. Subscriptions are reference counted per context, so multiple subscriptions to the same table in the same context will only result in one `RowChanged` event being sent for every change. Monitors should unsubscribe when they terminate (for example, in the `ondie()` action) to avoid leaking subscriptions.

The store factory bean property `rowChangedOldValueRequired` (see also [“Standard configuration properties for distributed stores” on page 375](#)) indicates whether subscribers receive previous values in `RowChanged` notification events for updated rows. When this property is set to `true` and the `RowChanged.changeType` field is set to `UPDATE` the `RowChanged.oldFieldValues` field is populated.

Notifications can impact performance, so are not recommended for tables in which a large number of changes are occurring. While `TCStore` and `BigMemory Max` support notifications, they do not support population of the old value in `RowChanged.changeType = UPDATE` events.

Within a cluster of correlators, if a table has subscriptions to `RowChanged` notifications, then all correlators must subscribe `RowChanged` notifications for that table, even if some correlators do not consume the events. This ensures all nodes receive all events correctly.

Support for notifications is optional, but if the driver does not support notifications, calls to `Table.subscribeRowChanged()` and `Table.unsubscribe()` will throw `OperationNotSupportedException` errors.

There is no support for `RowChanged` notifications for any extra fields (if supported). Only fields named in the schema are included in a `RowChanged` notification.

Some drivers do not provide `RowChanged` notifications for the rows removed by a `Table.clear()` operation. This behavior is driver-specific, therefore consult the driver documentation for more details.

Some drivers, such as `TCStore`, support sending a `MissedRowChanges` notification event in situations where an unknown number of `RowChanged` events may have been dropped, for example, due to a failure or high load.

Neither the `TCStore` driver nor the `BigMemory Max` driver set the old values with a `RowChanged` event. The old values sequence will always be empty for both of these drivers.

The `BigMemory Max` driver sets new values in the `RowChanged` event, but the `TCStore` driver does not. Therefore, if you need the new values in `TCStore`, you need to explicitly get them (using `Table.get` or similar).

See also the description of `com.apama.memorystore.RowChanged` in the *API Reference for EPL (ApamaDoc)*.

Configuration files for distributed stores

The configuration for a distributed store consists of a set of `.xml` and `.properties` files. Each distributed store in a project has the following files:

- `storeName-spring.xml`
- `storeName-spring.properties`

A distributed store is configured using a bean element in the Spring XML configuration file. The bean element has the following attributes:

- `id` – The unique name for this distributed store, which must match the name used in calls to `Storage.prepareDistributed()` and `Storage.open()` in EPL.
- `class` – The name of the `StoreFactory` implementation used by this distributed store.

When the correlator is started with the `--distMemStoreConfig dir` option (see also "Starting the correlator" in *Deploying and Managing Apama Applications*), it loads all XML files matching `*-spring.xml` in the specified configuration directory, and also all `*-spring.properties` files in the same directory. (Note, the correlator does not start unless the specified directory contains at least one configuration file.)

Note:

When the correlator is started, any properties that are specified with the `--config file` or `-Dkey=value` option take precedence and override the properties defined in a `storeName-spring.properties` file. An INFO message is then logged for all Spring properties that are being ignored.

When using Software AG Designer, these files are generated automatically. New `storeName-spring.xml` and `storeName-spring.properties` files are created when a store is added to a project. The most commonly used settings can be changed at any time using the Distributed MemoryStore editor (which rewrites the `.properties` file whenever the configuration is changed). In addition, the `storeName-spring.xml` files can be edited manually in Software AG Designer to customize more advanced configuration aspects. To edit the XML file, open the Distributed MemoryStore editor and in the **Configuration files** section, click the name of the file to open it in the appropriate editor. Once the editor for an XML file has been opened, you can switch between different views using the **Design** and **Source** tabs at the bottom of the editor window.

Some property values usually need to be changed when a development and testing configuration is deployed to a different environment such as one for production use. For more information on modifying property values when moving from a test environment to a production environment, see [“Changing bean property values when deploying projects” on page 389](#).

Making use of substitution variables is the best way to maintain different bean property values in different environments, as you can use the same XML file, with a different `.properties` file for each environment. For more details on using substitution variables to specify configuration properties, see [“Substitution variables” on page 375](#).

XML configuration file format

The configuration files for a distributed store use the Spring XML file format, which provides an open-source framework for flexibly wiring together the different parts of an application, each of which is represented by a *bean*. Each bean is configured with an associated set of *properties*, and has a unique identifier which can be specified using the `id` attribute.

It is not necessary to have a detailed knowledge of Spring to configure a distributed store, but you may wish to explore the [Spring 3.0.5](#) documentation to obtain a deeper understanding of what is going on and to leverage some of the more advanced functionality that Spring provides.

The Apama distributed `MemoryStore` configuration loads any bean that extends the Apama `AbstractStoreFactory` class.

Setting bean property values

Most bean properties have primitive values (such as string, number, boolean) which are set like this:

```
<property name="propName" value="my value"/>
```

However, it is also possible to have properties that reference other beans, such as a configuration bean defined by the third-party distributed cache provider. These property values can be set by specifying the `id` of a top-level bean as in the following example (where it is assumed that `myConfig` is the `id` of a bean defined somewhere in the file):

```
<property name="someConfigProperty" ref="myConfig"/>
```

Any top-level bean may be referenced in this way, that is, any bean that is a child of the `<beans>` element and not nested inside another bean. Referencing a bean that is defined in a different configuration file is supported.

Instead of referencing a shared bean, it is also possible to configure a bean property by creating an inner configuration bean nested inside the property value like this:

```
<property name="terracottaConfiguration">
  <bean class="net.sf.ehcache.config.TerracottaConfiguration">
    <property name="consistency" value="STRONG"/>
  </bean>
</property>
```

Note, advanced users may want to exploit Spring's property inheritance by using the `parent` attribute on an inner bean to inherit most properties from a standard top-level bean while overriding some specific subset of properties or by type-based auto-wiring.

You can use the Spring syntax for compound property names to set the value of a property held by another property. For example, to set a property `stringProp` on a bean held by the property `beanProp`, use the following:

```
<property name="beanProp.stringProp" value="myValue"/>
```

Or, to set the value of the key `myKey` in a property that holds a `Map` called `mapProp`, use the following:

```
<property name="mapProp[myKey]" value="myValue"/>
```

Substitution variables

Substitution variables in the form `${varname}` can be used to specify bean property values. Instead of specifying bean property values directly in an XML configuration file, you use `${varname}` substitution variables in the XML file and specify the values of those variables in a `.properties` file inside the configuration directory. This makes it possible to edit the variable values in Software AG Designer and to use different values during deployment to a production environment using the Apama Ant macros.

Although `.properties` and `storeName-spring.xml` files often have similar names, there is no explicit link between them, so *any* `properties` file can define properties for use by *any* `storeName-spring.xml` file. Although in some cases it may be useful to share a single substitution variable across multiple XML files, this is not normally the desired behavior, and therefore the recommendation is that all properties follow the naming convention `${varname.storeName}`.

In addition to the standard substitution variables shared by most drivers, you can add your own substitution variables for important or frequently changed properties specific to the driver specific to the cache integrated with your application. This is especially important when changing from a development environment to a production environment.

It is also possible to provide property values at runtime as Java system properties, such as specifying `-J-Dvarname=value` on the correlator command line.

The special variables `${APAMA_HOME}` and `${APAMA_WORK}` are always available.

Substitution variables are evaluated recursively. So a substitution variable can refer to another substitution variable, for example, `classpath=${installDir}/foo.jar`.

Standard configuration properties for distributed stores

The following standard properties are supported by Apama distributed cache drivers. These properties should be supported by customer-developed implementations as well.

Property Name	Description
<code>clusterName</code>	<p>This is a required property. It is a provider-specific string that is used to group together distributed store nodes that communicate with each other and share data. Store objects with the same <code>clusterName</code> value should operate as a single cluster, sharing data between them, whereas stores with different <code>clusterName</code> values should operate independently if possible.</p> <p>For providers such as Terracotta/TCStore that use other configuration properties to indicate which components to connect to, the <code>clusterName</code> is just a display name used in log messages with no impact on behavior.</p>

Property Name	Description
	<p>For BigMemory Max, the <code>clusterName</code> is a comma-separated list of <code>host:port</code> pairs that identify the servers in the Terracotta Server Array.</p> <p>Care should be taken to ensure that different clusters, and thus <code>clusterName</code> values, are used for development/testing and production environments, as serious errors would be introduced if the production and testing nodes were able to communicate with each other. For the Terracotta/TCStore driver, this is not a concern as it is only a display name. The BigMemory Max driver makes it easy to avoid this pitfall since it requires a list of <code>host:port</code>. However, if you are using another driver, then for this reason, as well as whatever firewalls may exist between development/testing and production, the recommendation is to explicitly add a suffix such as <code>_testing</code> or <code>_production</code> to the <code>clusterName</code> to indicate clearly to which environment it belongs.</p>
<code>logLevel</code>	<p>This is an optional property. The default is provider-specific, but is typically the same as the correlator log level.</p> <p>The <code>logLevel</code> property is an Apama log level string (compatible with <code>com.apama.util.Logger</code>) such as <code>ERROR</code>, <code>WARN</code>, <code>INFO</code>, <code>DEBUG</code> which is used to set the log level for the provider if possible (some providers write to the main correlator log file, through <code>log4j</code> or the Apama Logger, but others may write to a separate file).</p> <p>If not specified, the default log level is determined by the author of the driver, based on the criteria of avoiding the correlator log or <code>stdout</code> being filled with third-party distributed store messages while logging a small number of the most important messages.</p>
<code>backupCopies</code>	<p>This is an optional property. The default is 1.</p> <p>The <code>backupCopies</code> property specifies the number of additional redundant nodes that should hold a backup copy of each key/value data element. The minimum value for this property is 0 (indicating no redundancy, that is, all data is held by a single node).</p> <p>Note that some providers may allow customizing the backup count on a per-table basis, in which case this property specifies an overridable default value for tables that do not specify it explicitly.</p>

Property Name	Description
	<p>For BigMemory Max, this setting has no effect. The number of backup copies is determined by the Terracotta Server Array configuration, which is separate from the Apama configuration.</p> <p>This property is not used by the TCStore driver.</p>
<code>initialMinClusterSize</code>	<p>This is an optional property.</p> <p>The <code>initialMinClusterSize</code> property specifies the minimum number of nodes a cluster must have before the <code>Finished</code> event is sent in response to a call to <code>prepareDistributed</code>. This provides a way to make sure that a cluster is fully ready for correlator nodes to request and process data.</p> <p>The default is 1, which specifies that a <code>Finished</code> event is sent without waiting for additional nodes when preparing the distributed store.</p> <p>This property is not used by the TCStore driver.</p>
<code>rowChangedOldValueRequired</code>	<p>This property indicates whether the old value is required when there is a notification that a row has changed. The default is <code>true</code>.</p> <p>If set to <code>false</code>, the value of <code>oldFieldValues</code> is empty for <code>RowChanged.changeType.UPDATE</code> events.</p> <p>If set to <code>true</code>, the previous value is available. This cannot be set to <code>true</code> for TCStore or BigMemory Max.</p>
<code>enforceApamaSchema</code>	<p>This is an optional property.</p> <p>The <code>enforceApamaSchema</code> property sets whether Apama creates a special table to record the schema of each table and use it to prevent opening a table with a different Apama schema to the previously used schema.</p> <p>The default for most drivers (except TCStore) is to enable this checking to provide early warning of unexpected version mismatch issues. If a mismatch is detected, you should manually delete all tables used by Apama from the distributed store, typically using the <code>delete</code> command from the file system or tools provided by the distributed store. Note that this check provides no protection against non-Apama clients adding data that does not match the expected schema.</p>

If the standard properties were set, the bean configuration would look like:

```
<bean id="MyStore" class="com.foobar.MyStoreFactory">
  <property name="clusterName" value="host1:port1, host2:port2"/>
  <property name="logLevel" value="WARN"/>
  <property name="backupCopies" value="1"/>
  <property name="initialMinClusterSize" value="2"/>
  <property name="rowChangedOldValueRequired" value="true"/>
  <property name="enforceApamaSchema" value="true"/>
</bean>
```

TCStore (Terracotta) driver details

Apama provides a MemoryStore driver for Terracotta. This driver uses the TCStore API to allow Apama to read and write *records* in TCStore *datasets*, which may also be read and written by other Apama correlators and non-Apama components such as other Software AG products or custom clients written against the TCStore API.

The TCStore client libraries are installed automatically with the Apama server. So there is no need to explicitly select them at installation time, or to specify their location in the driver configuration.

Note:

Apama's TCStore driver only supports clustered mode. Currently, there is no support for embedded TCStore, for performing searches from Apama, or for the Ehcache API.

BigMemory Max is the recommended driver for caching use cases (see also [“BigMemory Max driver details” on page 382](#)). For Apama queries and applications that do *not* involve caching (such as store and "system of record" use cases), TCStore is the recommended MemoryStore driver.

Using TCStore from the Apama MemoryStore

The following table maps Apama MemoryStore terminology to TCStore classes; this may be useful when referring to the [Terracotta documentation](#):

Apama MemoryStore Event API Name	TCStore Class
Store	DatasetManager
Table	Dataset
Row	Record
Field value of a Row	Cell

Apama schemas and TCStore

Apama applications using the MemoryStore always specify a schema indicating the names and types of the fields (cell definitions) when preparing a table for use.

TCStore does not have a concept of schemas, as records in a given table are not required to have the same cell definitions. Row fields (cells) named in the schema during table preparation can always be accessed from Apama. It is not necessary for every field named in the schema to exist

in a given row (record). Apama simply returns a default value (for example, an empty value, 0 or false) when reading any field listed in the schema that is not present in a given row. It is also possible to access extra fields that are present in an individual row (record) but not listed in the schema, using any of the `getXXX` actions or `toDictionary()`. If a `getXXX` action is called with a key that is not present in the schema and is also not present as an extra field on the row, it will throw an exception. You can call `getKeys()` to see a list of all the fields (schema and non-schema) accessible.

Apama can therefore make use of non-homogenous data written to TCStore by other clients. If there is no strictly defined schema for a dataset, it is possible to not specify any fields in the schema, subject to the above restrictions. It is also possible for different Apama applications or different versions of the same application to interact with a single TCStore table using different schemas. When developing Apama applications, you should remember that all rows in a table do not necessarily contain the same fields, and ensure that all clients are using the same types for each field.

Apama can also dynamically add and remove additional non-schema fields. Any `set` method on a `Row` for a new field will create a new field on the row. Non-schema fields can be removed either with the `removeNonSchema` method or by setting the field to an empty any object.

Note:

All clients using a `Dataset` should agree on using the same types for a given cell name. The behavior when multiple clients read and write a row field (cell) with the same field name but different types is undefined, and may change in future releases of Apama.

Apama can only access cells of the Java types `Long` (maps to `integer` in EPL), `Double` (maps to `float` in EPL), `String` (for `string` or `decimal`), and `Boolean`. The `decimal` type is not supported by TCStore, so the driver converts to or from a string automatically for decimal fields defined in the schema. For non-schema fields, the EPL application needs to convert between string and decimal manually.

Configuring the TCStore driver

You can create configuration files for the TCStore driver using Software AG Designer. The only required property is the list of Terracotta servers to connect to.

The following table lists the properties that can be configured in the `storeName-spring.xml` file for the TCStore driver. The most commonly used properties also have `${varname.storeName}` substitution variables in the `storeName-spring.properties` file to make it easy to change the values by editing the properties file manually or using the Distributed MemoryStore editor in Software AG Designer.

Property Name	Description
<code>servers</code>	<p>Type: <code>string</code></p> <p>A comma-separated list of <code>host1:tsa-port1,host2:tsa-port2,...</code> values for each server from one of the stripes in the cluster, that is, all the servers defined in the Terracotta server configuration. Servers from other stripes are found automatically. A <code>terracotta://</code> URI prefix can optionally be added to the first <code>host:port</code>, but is not required. At least one server must be</p>

Property Name	Description
	specified. If high availability is required, multiple servers should be specified. See the Terracotta documentation for detailed information about cluster architecture.
useCompareAndSwap	<p>Type: boolean</p> <p>Specifies whether to provide safe atomic updating of rows by multiple clients. This means that a commit only succeeds if the value last seen by the client before it made the change matches the current value at the time of the commit. The default is true, which is the recommended setting for most use cases. Compare and swap can be disabled if each TCStore client (and thread) is known to have exclusive access to a given row, or if “best effort” rather than fully reliable behavior is required, which is likely to result in improved performance. If compare and swap is disabled, MemoryStore <code>commit()</code> operations do not fail as a result of writes made by other clients.</p> <div>Important: For correct behavior of Apama queries, you must leave the <code>useCompareAndSwap</code> property in its default (true) setting.</div>
classpath	<p>Type: string</p> <p>Specifies the path of the driver. This property should not be changed from its default value.</p>
connectionTimeoutSecs	<p>Type: integer</p> <p>Specifies the timeout used when establishing the initial connection to the Terracotta server while preparing the store. The default is 0, which uses the standard timeout supplied by TCStore. See the Terracotta documentation on <code>withConnectionTimeout</code> for further details.</p>
logLevel	<p>Type: string</p> <p>Specifies the verbosity of log messages from the Terracotta client. It does not affect the verbosity of the Apama TCStore driver. The default value for this is to remain unset, which uses the same log level for Terracotta as the correlator's main log level.</p>
clusterName	<p>Type: string</p> <p>Not recommended for use. Specifies a display name that is logged at startup. This setting has no impact on behavior, and does not have to match the name of the connected Terracotta cluster. The default is hardcoded: Terracotta.</p>
enforceApamaSchema	Type: string

Property Name	Description
	Not recommended for use. Specifies whether Apama should attempt to detect use of conflicting schemas by multiple Apama clients using the same Terracotta cluster. The default is <code>false</code> , as for typical TCStore use cases this checking is not useful.
<code>securityRootDirectory</code>	Type: string Specifies the path to the security root directory, if using TLS. For more information, see the Terracotta documentation.

The following properties are configured in the same files, but are used only when Apama creates a new dataset (table) as a result of trying to open a dataset that does not yet exist. *These settings have no effect if the dataset was already created*, whether by Apama or another TCStore client.

These settings apply to all datasets created by a given Apama store. If it is necessary to create datasets with different settings on the same Terracotta cluster, you have to configure separate Apama stores for each group of datasets needing the same configuration, or use another tool or client to create the datasets. To change the dataset settings after creation, you have to delete any data directories and restart the cluster. See the Terracotta documentation for more information.

The dataset creation configuration properties are:

Property Name	Description
<code>offheapResourceName</code>	Type: string Specifies the name of a resource defined in the Terracotta server configuration to be used for off-heap storage of the dataset contents. This is a required setting when creating datasets from Apama.
<code>enableDatasetPersistence</code>	Type: boolean Specifies whether dataset contents are persisted on disk. The default is <code>false</code> , indicating that dataset contents should be held only in memory.
<code>persistenceDataRootName</code>	Type: string Specifies the logical name of a data directory defined in the Terracotta server configuration. Note that this is a logical name; it is not a directory path. This property only has an effect if <code>enableDatasetPersistence</code> is <code>true</code> .

The following standard configuration properties are not used by this driver and are either ignored or rejected if set:

- `backupCopies`
- `initialMinClusterSize`

- `clusterName`
- `rowChangedOldValueRequired` (cannot be set to `true` for this driver).

Note:

TCStore currently sends notifications for rows removed by a `Table.clear()` operation, but this may change in a future release.

Note:

The standard `enforceApamaSchema` property is set to `false` by default for TCStore, which is the recommended setting in almost all cases.

BigMemory Max driver details

Note:

Support for using BigMemory Max for queries is deprecated and will be removed in a future release. It is recommended that you now use Terracotta's TCStore for queries. For queries and applications that do not involve caching, the TCStore driver is now the recommended driver for the distributed MemoryStore.

Apama continues to support the BigMemory Max driver. This is still the recommended driver for caching use cases, but no longer for queries.

Using BigMemory Max from the Apama MemoryStore

For reference, the following table maps Apama MemoryStore terminology to BigMemory Max classes; this may be useful when referring to the BigMemory Max documentation:

MemoryStore Event Object	BigMemory Max Class
Store	CacheManager
Table	Cache
Row	Element

By default, a distributed MemoryStore `Store` uses the BigMemory Max default cache manager. To specify the use of a different cache manager, specify the `name` property on the configuration bean. For example:

```
<property name="configuration.name" value="myCacheManager"/>
```

In a cluster, if one correlator calls `subscribeRowChanged()` for a given MemoryStore table, then all correlators in that cluster that modify the entries in that table must also call `subscribeRowChanged()` on that table even if they do not consume the events.

Iterating over a table may require pulling the entire table into memory. It may fail if the table is being modified.

If accessing a BigMemory Max table from Apama and non-Apama applications, clients will need the correct cache configuration (available from the Terracotta Management Console) and have the appropriate Apama classes available on their classpath (available in the `distmemstore` and `ap-distmemstore-bigmemory.jar` files) in order to access the cache.

Configuring the BigMemory Max driver

You can create configuration files for BigMemory Max when using Apama in Software AG Designer. The `providerDir` property should be set to the location of the BigMemory Max installation whose client libraries will be used, which is typically in the same location as Apama.

The driver for BigMemory Max is configured as follows:

- You can set BigMemory Max driver properties (described in the table below) in the `storeName-spring.xml` configuration file. Alternatively, you can specify many of these properties in an `ehcache.xml` configuration file and then specify the path for that file in the `storeName-spring.xml` configuration file using the `ehcacheConfigFile` property. If this is done, many of the properties in the `storeName-spring.xml` configuration file will be ignored; the settings derived from the `ehcache.xml` file will be used instead.
- Use the `storeName-spring.properties` file to set configuration properties for the BigMemory Max driver.
- Using off-heap storage requires setting `-XX:MaxDirectMemorySize=`. Specify this in the command line for starting the correlator as `-J-XX:MaxDirectMemorySize=`. The BigMemory Max documentation provides recommendations for specifying the value of this property. When you add a correlator to a correlator launch configuration in Software AG Designer, you can select the **Maximum Java off-heap storage in MB** option. See "Correlator arguments" in *Using Apama with Software AG Designer*.

When using the BigMemory Max driver, all correlators accessing the same data in a BigMemory Max cluster must have the same configuration.

Driver bean properties with equivalents in ehcache.xml

The following properties can be set either in the driver properties, or using a `ehcache.xml` configuration file. See the [BigMemory Max documentation](#) for more details. For more information on the Ehcache types mentioned below, see the Ehcache Javadoc and search for the required type such as `CacheConfiguration`.

Property Name	Description
<code>cacheConfiguration</code>	Type: <code>CacheConfiguration</code> Ehcache <code>CacheConfiguration</code> bean, shared by all caches (Tables). Typically used as a compound bean name, for example, <code>cacheConfiguration.overflowToOffHeap</code> .
<code>cacheConfiguration.eternal</code>	Type: <code>boolean</code>

Property Name	Description
	Disables expiration (removing old, unused values) of entries if true. Set to true in the default <i>storeName-spring.xml</i> configuration file.
cacheConfiguration.maxEntriesLocalHeap	Type: int The number of entries for each table. This is the maxEntriesLocalHeap entry in the .properties file.
cacheConfiguration.overflowToOffHeap	Type: boolean Whether to use off-heap storage. For scenarios where data is fast changing and being written from multiple correlators, the cache may perform better if this is disabled. This is the cacheConfiguration.overflowToOffHeap entry in the .properties file.
cacheDecoratorFactory	Type: String Name of a class to use as a cacheDecoratorFactory. The named class must be on the classpath and must implement Ehcache's CacheDecoratorFactory interface.
cacheDecoratorFactoryProperties	Type: Properties Properties to pass to a cacheDecoratorFactory. Allows use of the same class for many caches.
clusterName	Type: String Comma-separated list of host:port identifiers for the servers, or a tc-config.xml file name. Best practice is to list all Terracotta Server Array (TSA) nodes.
configuration	Type: Configuration Ehcache Configuration bean. Typically used as a compound bean name, for example, configuration.monitoring.
configuration.name	Type: String By default, the BigMemory Max default cache manager is used. Use this property to specify the use of a different cache manager.

Property Name	Description
<code>maxMBLocalOffHeap</code>	Type: long Number of MB of local off-heap data. Total across all tables, per correlator process.
<code>pinning</code>	Type: String Either an attribute value of "inCache" (default) or "localMemory" or a <null/> XML element (that is, <property name="pinning"><null/></property>.) Pinning prevents eviction if the cache size exceeds the configured maximum size. Recommended if the cache is being used as a system of record.
<code>terracottaConfiguration</code>	Type: TerracottaConfiguration Ehcache TerracottaConfiguration bean. Typically used as a compound bean name, for example, <code>terracottaConfiguration.consistency</code> .
<code>terracottaConfiguration. clustered</code>	Type: boolean Whether to use a TSA. Set to true in the default <code>storeName=spring.xml</code> configuration file.
<code>terracottaConfiguration. consistency</code>	Type: String Either 'STRONG' or 'EVENTUAL'. STRONG gives MemoryStore-like guarantees, while EVENTUAL is faster but may have stale values read. This is the <code>terracottaConfiguration.consistency</code> entry in the <code>.properties</code> file.
<code>terracottaConfiguration. localCacheEnabled</code>	Type: boolean Whether to cache entries in the correlator process. Set to true in the default <code>storeName=spring.xml</code> configuration file.
<code>terracottaConfiguration. synchronousWrites</code>	Type: boolean If true, then data is guaranteed to be out of process by the time a <code>Row.commit()</code> action completes. Disabling this can increase speed. This is the <code>terracottaConfiguration.synchronousWrites</code> entry in the <code>.properties</code> file.

Driver bean properties with no equivalents in ehcache.xml

You can set the following BigMemory Max driver properties in the `storeName-spring.xml` configuration file, but not in the `ehcache.xml` configuration file as they modify how the driver accesses the BigMemory Max cache.

Property Name	Description
<code>backupCopies</code>	Type: <code>int</code> Ignored. Not supported. The number of backups is governed by the TSA topology defined in the BigMemory Max documentation and used to configure the TSA nodes.
<code>converterConfig.default</code>	Type: <code>RowKeyValueConverter</code> bean Specifies the default converter to be used for all tables which do not explicitly specify a converter. A converter is a bean of type <code>RowKeyValueConverter</code> . It is used to convert non-Apama format Ehcache key/value pairs to Apama format key/value pairs. If not specified, then Apama's default converter is used which assumes that the key/value pairs in Ehcache are in Apama format. For example: <pre><property name="converterConfig.default" ref="MyDefaultConverter"/></pre> See the <i>API Reference for Java (Javadoc)</i> and the samples in the <code>samples\distmemstore\bigmemory\converters</code> directory of your Apama installation for more information about <code>RowKeyValueConverter</code> which can be found in the <code>com.apama.correlator.memstore</code> package.
<code>converterConfig.byTable</code>	Type: <code>Map(String, RowKeyValueConverter)</code> Per-table converter configuration. If a converter is not specified for a table, the default converter specified by the <code>converterConfig.default</code> property is used. For example: <pre><property name="converterConfig.byTable"> <map> <entry key="SensorTable"> <bean class="SensorDataConverter" /> </entry> </map> </property></pre>
<code>ehcacheConfigFile</code>	Type: <code>String</code>

Property Name	Description
	<p>Path to an ehcache.xml configuration file.</p> <p>Note: If this is specified, any other properties listed in this table will be ignored.</p>
exposeSearchAttributes	<p>Type: boolean</p> <p>Enable exposing search attributes. If true, then the MemoryStore schema columns are exposed as BigMemory Max search attributes and are indexed, so that other clients of BigMemory Max can perform searches on the data set. If exposeSearchAttributesSet is non-empty, then only the named columns are exposed as BigMemory Max search attributes. See notes below about non-Apama applications accessing the data in a BigMemory Max cluster.</p>
exposeSearchAttributesSet	<p>Type: Set(String)</p> <p>Limits the set of columns in each table that should be exposed as search attributes. Entries are in the form <i>tableName.columnName</i>. If empty, all schema columns are exposed as search attributes. There is an incremental cost per column that is exposed, so for performance, only expose the columns which need to be used in searches.</p> <p>For example, to expose only the Surname and FirstName columns of myTable:</p> <pre><property name="exposeSearchAttributes" value="true"/> <property name="exposeSearchAttributesSet"> <set> <value>myTable.Surname</value> <value>myTable.FirstName</value> </set> </property></pre>
initialMinClusterSize	<p>Type: int</p> <p>The minimum cluster size (number of correlators) that must be connected for prepare to finish.</p>
logLevel	<p>Type: String</p> <p>The log level.</p>
rowChangedOldValueRequired	<p>Type: boolean</p>

Property Name	Description
	Whether to expose old values in <code>rowChanged</code> events. Must be set to <code>false</code> . Note: BigMemory Max currently sends notifications for rows removed by a <code>Table.clear()</code> operation, but this may change in a future release.
<code>useCompareAndSwap</code>	Type: <code>boolean</code> Whether to use compare and swap (CaS) operations or just put/remove. Some versions of BigMemory Max support only CaS in Strong consistency. Important: For correct behavior of Apama queries, you must leave the <code>useCompareAndSwap</code> property in its default (<code>true</code>) setting.
<code>useCompareAndSwapMap</code>	Type: <code>Map(String, Boolean)</code> Per-table (cache) configuration for whether to use CaS or put/remove.

Migrating from BigMemory Max to TCStore

If you have an existing Apama application that uses BigMemory Max and you wish to move it to use TCStore instead, simply open the Distributed MemoryStore editor in Software AG Designer, remove the BigMemory Max store and add a new TCStore using the same store name. See [“Adding a distributed store to a project” on page 369](#) for more details.

Configure the new TCStore instance as described in [“TCStore \(TERRACOTTA\) driver details” on page 378](#). There is typically no need to migrate any driver configuration settings from BigMemory Max to TCStore, since the options for each driver are quite different, and because TCStore requires a lot less configuration on the client side than BigMemory Max. In most cases, the default TCStore driver settings should be left unchanged. If you had configured search attributes or were using custom converters in BigMemory Max, there is no need to do anything similar when moving to TCStore, as row values in TCStore are stored in a standard format and are automatically searchable. The `initialMinClusterSize` and `useCompareAndSwapMap` (per-table) settings are supported by BigMemory Max but cannot be set for TCStore. The TCStore driver does not support client-side caching, so there is no need to migrate any cache-related settings from BigMemory Max, or to pass an off-heap storage option (`-XX:MaxDirectMemorySize`) to the correlator.

In most cases, there will be no need to make changes to your EPL application. If you wish, you can make use of features available in the TCStore driver but not BigMemory Max, such as accessing row fields that are not in the schema. If using “row changed” notifications, the TCStore driver will sometimes send a `MissedRowChanges` event if it has detected that some of the `RowChanged` events may have been dropped; BigMemory Max does not support sending `MissedRowChanges` notifications.

The BigMemory Max driver sets new values in the RowChanged event, but the TCStore driver does not. Therefore, if you are migrating an application from BigMemory Max and you need the current or new values, you need to explicitly get them (using `Table.get` or similar).

See also the description of `com.apama.memorystore.RowChanged` in the *API Reference for EPL (ApamaDoc)*.

Note:

There is no way to automatically transfer data from BigMemory Max to a Terracotta server for use by TCStore.

Changing bean property values when deploying projects

Some bean property values will usually need to be changed when a development/testing configuration is deployed to a different environment such as production, which is typically achieved by ensuring that all such bean property values are specified using `${varname}` substitution variables specified in `.properties` files for test versus production environments. For example, for some distributed memory stores the `clusterName` should be changed so that the nodes cannot talk to each other (although Apama also recommends production nodes to be located on a different network to reduce the chance of accidental errors). For more details on using substitution variables to specify configuration properties, see [“Substitution variables” on page 375](#).

Tip:

Due to the flexibility and simplicity of `.properties` files, there are many ways this requirement can be addressed. For customers using Apama's Ant macros for deployment, one option is to maintain a separate set of `.properties` files for each environment, and customize your project's Ant script to copy the correct version of the files into the directory defined by the `--distMemStoreConfig` option just before starting the correlator. Another option is to use Ant's `<propertyfile>` task (see the Apache Ant documentation for more information on how to do this) to modify the `.properties` files in-place, overriding or adding to existing property values as required for the new deployment.

Creating a distributed MemoryStore driver

The Apama installation includes a driver for integrating the distributed MemoryStore with TCStore or the BigMemory Max distributed caching software. If you use other third-party distributed caching software, you need to write a driver that provides the bridge between Apama's MemoryStore and the third-party software in use. Apama provides a Service Provider Interface (SPI) for you to use when writing drivers.

This topic presents an introduction to the SPI and a description of its essential elements. Complete Javadoc information for the SPI is available in the *API Reference for Java (Javadoc)*; see the `com.apama.correlator.memstore` package.

Overview

A driver for a distributed cache needs to extend the following abstract classes:

- `AbstractStoreFactory`

- `AbstractStore`
- `AbstractTable`

Implementation details:

- `AbstractStoreFactory` – This is the abstract class that holds the configuration used to instantiate a distributed Store. The starting point for creating an Apama distributed cache driver is to create a concrete subclass of `AbstractStoreFactory`. The subclass should have the following:
 - A public no-args constructor.
 - JavaBeans-style setter and getter methods for all provider-specific configuration properties.
 - An implementation of `createStore()` that makes use of these product-specific properties, in addition to the generic properties defined on this factory, which are `getClusterName()`, `getLogLevel()`, and `getBackupCopies()`.
 - `afterPropertiesSet()` (optional, but useful).

Implementers are encouraged to do as much validation as possible of the configuration in the `afterPropertiesSet()` method. This method is called by Spring during correlator startup, after setters have been invoked for all properties in the configuration file. The `createStore()` action will never be called before this has happened.

The `StoreFactory` class that is implemented must then be named in the `storeName-spring.xml` configuration file for the distributed store.

- `AbstractStore` – This is the abstract class that provides access to tables whose data is held in a distributed store. Implementers should create a subclass of `AbstractStore`.

A driver's implementation of the `AbstractStore` needs to implement or override the following methods:

- `createTable()`
- `init()`
- `close()`
- `getTotalClusterMembers()`
- `AbstractTable` – This is the abstract class that holds Row objects whose data is held in a distributed store.

If the distributed store provides a `java.util.concurrent.ConcurrentMap`, Apama recommends that implementers of Apama distributed stores create a subclass of the `ConcurrentMapAdapter` abstract class for ease of development and maintenance. If the distributed store does not provide a `ConcurrentMap`, implementers should create a subclass of Apama's `AbstractTable` class.

If you are implementing from `AbstractTable`, you need to implement or override the following methods:

- `get()`

- `clear()`
- `remove()`
- `replace()`
- `putIfAbsent()`
- `containsKey()`
- `size()`

Drivers may also optionally provide support for EPL subscribing to “row changed” data notifications. To allow EPL application to subscribe to these notifications, subclasses of `AbstractTable` (or `ConcurrentMapAdapter`) must provide an instance of `RowChangedSubscriptionManager` that provides implementations of `addRowChangeListener` and `removeRowChangeListener`, and calls `fireRowChanged` when changes are detected. Also, if a subclass implements notifications, it should override the `getRowChangedSubscriptionManager` method and have it return the instance of `RowChangedSubscriptionManager` for this table. Calls to `subscribeRowChanged` and `unsubscribe` are passed to this instance. The default implementation of `getRowChangedSubscriptionManager` returns null, indicating that “row changed” notifications are not supported; in this case, calls to `subscribeRowChanged` and `unsubscribe` will throw `OperationNotSupportedException`.

Drivers may also optionally provide support for accessing extra fields that are present in an individual row but not in the table's schema. To do this, the table should implement the marker interface `TableSupportsExtraFields` and use the `RowValue.getExtraFields()` method to get and set a row's extra fields when converting between `RowValue` objects and the data type used by the distributed store to represent row values.

- **RowValue** – The `RowValue` class is not inherited from or implemented, but a driver must be able to store and retrieve objects of the Apama `RowValue` class. Typically, a cache can store any suitable Java class, but some mapping may be required as well. For more information about this class, see the *API Reference for Java (Javadoc)* for `com.apama.correlator.memstore.RowValue`.

Sample driver

To help get started writing a driver, the BigMemory Max driver is provided in source form as a sample. It implements the SPI described above and invokes the Ehcache API in order to use BigMemory Max. The sample is provided in the `samples/distmemstore_driver/bigmemory` directory of the Apama installation. To avoid confusion with the pre-compiled driver supplied in the product, the sample BigMemory Max driver uses the package name `com.apamax.memstore.provider.bigmemory`. A `README.txt` file describes how to build the sample.

Using the Management interface

The Management interface defines actions that let you do the following:

- Obtain information about the correlator
- Connect the correlator to another component

- Control logging
- Request a persistence snapshot
- Manage user-defined status values

Actions in the Management interface are defined on several event types, which are documented in the *API Reference for EPL (ApamaDoc)*.

To use the Management interface, add the **Correlator Management** EPL bundle to your Apama project (see "Adding EPL bundles to projects" in *Using Apama with Software AG Designer* or "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*). Alternatively, you can directly use the EPL interfaces provided in `APAMA_HOME\monitors\Management.mon`.

Obtaining information about the correlator

The Management interface provides actions for obtaining information about the correlator that the Management interface is being used in. These actions are defined in the `com.apama.correlator.Component` event.

There are also options of the `engine_management` tool that you can specify (see also "Shutting down and managing components" in *Deploying and Managing Apama Applications*):

- to retrieve the same information from outside the correlator,
- or to retrieve the same information for IAF processes.

The correlator also logs all of these values to its log file at startup.

Connecting the correlator to another component

The `com.apama.correlator.Component` event provides actions that allows EPL to create connections to another component, in the same way as the `engine_connect` tool (see "Configuring pipelining with `engine_connect`" in *Deploying and Managing Apama Applications*).

Controlling logging

You can configure logging using the Management interface. The `com.apama.correlator.Logging` event provides actions such as `setApplicationLogFile`, `setLogFile` and `setApplicationLogLevel`. These actions are the equivalent of using the `engine_management` options to configure logging (see also "Shutting down and managing components" in *Deploying and Managing Apama Applications*).

The `rotateLogs()` action, which is also defined in the `com.apama.correlator.Logging` event, is used for closing the log files in use, opening new log files, and then sending messages to the new log files. This action applies to:

- the main correlator log file,
- the correlator input log file if you are using one, and
- any EPL log files you are using.

For details about log file rotation, see "Rotating correlator log files" and "Rotating all correlator log files" in *Deploying and Managing Apama Applications*.

You can write an EPL monitor that triggers log rotation on a schedule. For example, the code below rotates logs every 24 hours at midnight:

```
using com.apama.correlator.Logging;

monitor Rotator {

    action onload() {
        on all at(0, 0) {
            Logging.rotateLogs();
        }
    }
}
```

Requesting a snapshot

In a persistence-enabled correlator, you can use the Management interface to request a snapshot to occur as soon as possible, and be notified of when that snapshot has been committed to disk. The Management interface lets persistent and non-persistent monitors create instances of Persistence events and then call the `persist()` action on those events.

When the correlator processes the `persist()` call it takes and commits a snapshot and executes the specified callback action at some point after the snapshot is committed. There are no guarantees about the elapsed time between the `persist()` call, the snapshot and the callback, especially when large amounts of correlator state are changing. Your code resumes executing immediately after the call to the `persist()` action. See ["Using Correlator Persistence" on page 307](#).

The Management interface defines the Persistence event:

```
package com.apama.correlator;
event Persistence {
    static action persist(action<> callback) {
        ...
    }
}
```

Consider the following sample code:

```
using com.apama.correlator.Persistence;
event Number {
    integer i;
}

persistent monitor MyApplication {
    integer counter := 0;
    sequence<integer> myNumbers;
    action onload() {
        on all Number(*) as n {
            myNumbers.append(n.i);
            counter := counter + 1;
            if(counter % 10 = 0) {
                doCommit();
            }
        }
    }
}
```

```
    }  
  }  
  
  action doCommit() {  
    Persistence.persist(logCommit);  
  }  
  
  action logCommit() {  
    log "Commit succeeded";  
  }  
}
```

Because `MyApplication` is a persistent monitor the correlator copies its state to disk as that state changes. This monitor listens for `Number` events and stores their content in the `myNumbers` sequence. After every tenth `Number` event, the code executes the `doCommit()` action, which uses the `Persistence` event in the `Management` interface to request that the correlator commits persistent state to disk. When that commit has succeeded, the `Management` interface calls the action variable that was passed to the `persist()` action. This action writes "Commit succeeded" to the correlator log.

The `Management` interface guarantees that at the moment the callback action (`logCommit()` in this example) is executed, the state of all persistent monitors at a particular point in time will have been committed. The particular point in time is guaranteed only to be between the point at which `persist()` was called and the point at which the callback action was executed. For example, suppose the following event stream is being sent into the correlator:

```
Number(1)  
Number(2)  
Number(3)  
...  
Number(10)  
Number(11)  
Number(12)
```

At the point that `Number(10)` is received, the `myNumbers` sequence contains the ten items 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and so the application initiates a snapshot commit. Suppose that the correlator suddenly terminates after notification of success appears in the log. When the correlator recovers, `MyApplication` has a `myNumbers` sequence that contains *at least* ten items. However, the sequence might contain 11 or even 12 items, if more `Number` events were received after the commit was requested but before the snapshot was actually taken. The correlator also persists state periodically, or as directed by other monitors that call the `Management` interface, so the sequence can be persisted at other points as well.

Managing user-defined status values

The `Management` interface provides actions for managing (set and return) the user-defined status values. These actions are defined in the `com.apama.correlator.Component` event and in the `com.apama.correlator.EngineStatus` event.

Status keys will have leading and trailing whitespace stripped. Keys may not be empty.

A user-defined status will only be changed if the new value differs from the current value when set using `setStatus`.

Note that the correlator status statements that appear in the log files will not have the user-defined status values, and will remain unaffected.

You can monitor the status of each component of your application using Command Central. See "Monitoring the KPIs for EPL applications and connectivity plug-ins" in *Deploying and Managing Apama Applications* for examples of using user-defined status values.

Using the JSON plug-in

You can use the JSON plug-in to convert EPL objects to JSON strings, and vice versa. When converting from JSON strings, JSON objects are converted to `dictionary<any, any>` and JSON lists are converted to `sequence<any>`.

To use the JSON plug-in, add the **JSON Support** EPL bundle to your Apama project. For details, see "Adding EPL bundles to projects" in *Using Apama with Software AG Designer* or "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*.

The JSON plug-in is provided as a `JSONPlugin` event in the `com.apama.json` package. The `JSONPlugin` event provides the following actions:

- To convert a JSON string to an EPL object:

`fromJSON(string)` returns `any`

- To convert an EPL object (including events, dictionaries and sequences) to a JSON string:

`toJSON(any)` returns `string`

For detailed information, see the *API Reference for EPL (ApamaDoc)*.

The following is a simple example:

```
using com.apama.json.JSONPlugin;

event Address {
    integer houseNumber;
    sequence<string> address;
    optional<string> postcode;
}

monitor PrintJSON {
    action onload() {
        Address a1 := new Address;
        Address a2 := new Address;
        a1.houseNumber := 107;
        a1.address.append("The Rounds");
        a1.address.append("Milton");
        a1.postcode := "CB1 2AB";
        a2.houseNumber := 196;
        a2.address.append("Exeter Road");
        a2.address.append("Newmarket");
        print JSONPlugin.toJSON(a1);
        print JSONPlugin.toJSON(a2);
    }
}
```

The above example prints the following:

```
{"postcode":"CB1 2AB","address":["The Rounds","Milton"],"houseNumber":107}  
{"postcode":null,"address":["Exeter Road","Newmarket"],"houseNumber":196}
```

Equivalent functionality is available with the JSON codec connectivity plug-in. Use the JSON codec if the JSON is coming in as an entire message of a connectivity chain. Using a codec helps to separate the application logic from the connection over which the data is being sent, and allows the use of mapping codecs if needed. See "The JSON codec connectivity plug-in" in *Connecting Apama Applications to External Components* for detailed information.

Use the JSON EPL plug-in if only one field of an event is JSON, or if the events are going to or are coming from a connection that is not using a connectivity chain.

Using MATLAB® products in an application

To use MATLAB analysis and modeling capability in an Apama application or in an application built using Apama Capital Markets Foundation, you need to add the **MATLAB** EPL bundle to your project (see "Adding EPL bundles to projects" in *Using Apama with Software AG Designer* or "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*) and ensure that the MATLAB executables and libraries are available to the correlator. The **MATLAB** bundle provides access to the MATLAB analysis and modeling toolkit from Apama EPL code and includes an EPL plug-in.

For information about the supported MATLAB versions, see the *Supported Platforms* document for the current Apama version. This is available from the following web page: <http://documentation.softwareag.com/apama/index.htm>.

This MATLAB plug-in lets you connect to and use the MATLAB engine. However, there are some functions/toolkits for which MATLAB does not support integration with C or Fortran on some operating platforms. Check the MATLAB documentation before using the MATLAB EPL plug-in.

The recommended way to use the MATLAB plug-in is to use the `MatlabManager` event, and call the relevant action and supply a callback. The call goes directly to the MATLAB plug-in, so you do not need to route a request event. `*Response` events are routed from the MATLAB plug-in to the calling context. Each request action automatically sets up a listener for the `*Response` event that will call the supplied callback. You can supply the relevant `doesNothing*Callback()` action from the `MatlabManager` event if you are not interested in the results of the callback. If you use the `MatlabManager` actions, you do not need to call the `#initialize()` action.

The legacy way to use the MATLAB plug-in is to route `*Request` events and set up listeners for the `*Response` events. If you are using the MATLAB plug-in in only the main context, injecting `MatlabService.mon` sets up all required listeners for the `*Request` events that call into the MATLAB plug-in. To use the MATLAB plug-in from another context, instantiate a `MatlabManager` variable, spawn to the other context, and call `#initialize()` on the variable. This sets up the required listeners in the current (non-main) context, and the `*Response` events are routed to this context.

Note:

The MATLAB plug-in is asynchronous (except for the `OpenSession` requests), so the processing of the input queue, or calling the request actions, does not block.

Due to a shutdown timing issue in MATLAB, if you repeatedly open and close MATLAB sessions, you may see the error message - A primary message table for module 77. This is due to a race with shutting down the MATLAB COM server. The workaround for this is to have an unused idle MATLAB instance running that will keep the COM server running.

The MATLAB plug-in is multi-context aware. The *Response events are routed to the calling context.

➤ To include MATLAB capabilities in your application

1. Ensure that the directory containing the MATLAB plug-in library is included in the library search path: %APAMA_HOME%\bin should be in the PATH on Windows platforms. Or for deployment on Linux operating systems, \$APAMA_HOME/lib should be in the LD_LIBRARY_PATH.
2. Import the MATLAB plug-in in the application's EPL code.
3. Set the appropriate values for your PATH environment variable:
 - 64-bit Windows: Add `MATLAB_HOME/bin` and `MATLAB_HOME/bin/win64` to %PATH%.
 - 64-bit Linux: Add `MATLAB_HOME/bin` to \$PATH. Also, add `MATLAB_HOME/sys/os/glnxa64` and `MATLAB_HOME/bin/glnxa64` to the end of \$LD_LIBRARY_PATH (after the Apama library).

Note:

On Linux, if you have problems starting Apama and MATLAB, especially with library incompatibilities such as with libssl, libcrypto or libz, ensure the LD_LIBRARY_PATH is ordered to load the newest version of the library first (which may be in the Apama or MATLAB installation, or the version in the operating system). It may help to try running a standalone MATLAB from an Apama Command Prompt, or to use the `ldd` utility to diagnose.

MatlabManager actions

The `MatlabManager` event provides the following actions. For complete reference information, see the *API Reference for EPL (ApamaDoc)*.

Action	Description
<code>openSession(string sessionId, string messageId, boolean singleUse, integer precision, action<string, string, boolean, string> callback)</code>	Starts a MATLAB process for the purpose of using MATLAB as a computational engine. Uses the MATLAB API function <code>engOpen()</code> if <code>singleUse</code> is false and <code>engOpenSingleUse()</code> if <code>singleUse</code> is true. Single use is unavailable on Linux. The response to this action call is an <code>OpenSessionResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.
<code>closeSession(string sessionId,</code>	Closes a MATLAB session. Uses the MATLAB API function <code>engClose()</code> . The response to this action call

Action	Description
<pre>string messageID, action<string, string, boolean, string> callback)</pre>	is a CloseSessionResponse event routed from the plug-in to the calling context and the supplied callback is invoked.
initialize()	You must call this action when you are using MATLAB by means of routed events in a context other than the main context. Spawn to another context, set up the relevant listeners in the new context, and then call initialize(). You do not need to call initialize() when you are calling the MatlabManager actions.
<pre>putFloat(string sessionID, string messageID, string name, float value, action<string, string, boolean, string> callback)</pre>	<p>Puts a float variable into a MATLAB engine workspace. Uses the MATLAB API function engPutVariable(). The response to this action call is a PutFloatResponse event routed from the plug-in to the calling context and the supplied callback is invoked.</p> <p>Note: By default, this event creates a local variable in the MATLAB session. If you need the variable to have a global scope, call evaluate() before you call the putFloat() action. In the evaluate() call, declare the variable as being global (for example, "global x").</p>
<pre>getFloat(string sessionID, string messageID, string name, action<string, string, float, boolean, string> callback)</pre>	Gets a float variable from the MATLAB engine workspace. Uses the MATLAB API function engGetVariable(). The response to this action call is a GetFloatResponse event routed from the plug-in to the calling context and the supplied callback is invoked.
<pre>putFloatSequence(string sessionID, string messageID, string name, sequence<float> values, action<string, string, boolean, string> callback)</pre>	Puts a float sequence variable in a MATLAB engine workspace. Uses the MATLAB API function engPutVariable(). The response to this action call is a PutFloatSequenceResponse event routed from the plug-in to the calling context and the supplied callback is invoked.
<pre>getFloatSequence(string sessionID, string messageID, string name, action<string, string, sequence<float>, boolean, string></pre>	Gets a float sequence variable from the MATLAB engine workspace. Uses the MATLAB API function

Action	Description
<code>callback()</code>	<code>engGetVariable()</code> . The response to this action call is a <code>GetFloatSequenceResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.
<code>putFloatMatrix(string sessionId, string messageId, string name, sequence<sequence<float>> values, action<string, string, boolean, string> callback)</code>	Puts a two-dimensional matrix variable into a MATLAB engine workspace. Uses the MATLAB API function <code>engEval()</code> . The response to this action call is a <code>PutFloatMatrixResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.
<code>getFloatMatrix(string sessionId, string messageId, string name, action<string, string, sequence<sequence<float>>, boolean, string> callback)</code>	Gets a two-dimensional matrix variable from the MATLAB engine workspace. Uses the MATLAB API function <code>engGetVariable()</code> . The response to this action call is a <code>GetFloatMatrixResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.
<code>evaluate (string sessionId, string messageId, string expression, integer outputSize, action<string, string, string, sequence<string> boolean, string> callback)</code>	Evaluates an expression for the MATLAB engine session and returns textual output from evaluating the expression, including possible error messages. Uses the MATLAB API function <code>engEvalString()</code> . The response to this action call is an <code>EvaluateResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.
<code>setVisible(string sessionId, string messageId, boolean value, action<string, string, boolean, string> callback)</code>	Makes the window for the MATLAB engine session either visible or invisible on the Windows desktop. Uses the MATLAB API function <code>engSetVisible()</code> . The response to this action call is a <code>SetVisibleResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.
<code>getVisible(string sessionId, string messageId, action<string, boolean, boolean, string> callback)</code>	Returns the current visibility setting for the MATLAB engine session. Uses the MATLAB API function <code>engGetVisible()</code> . The response to this action call is a <code>GetVisibleResponse</code> event routed from the plug-in to the calling context and the supplied callback is invoked.

MATLAB examples

To use MATLAB features in your Apama or Apama Capital Markets Foundation application, you must create a MATLAB session. The following examples show how to create a MATLAB session and how to use it to set or get floating point scalar values, arrays or matrices. Each get or set request has an associated response that indicates whether the request successfully completed.

Creating a MATLAB session

The following example creates a MATLAB session. A boolean value indicates whether MATLAB should open a new session or re-use an existing session.

```
monitor MatlabExample2
{
    // ***** Creating a MATLAB session:
    com.apamax.matlab.MatlabManager matlabManager;

    action onload() {
        // Spawn to a new context:
        spawn run() to context("New Context");
    }

    action run() {
        // Running in a context other than main, open a MATLAB session:
        matlabManager.openSession(
            "Session1", "openSessionRequest", false, 6, sessionOpened);
    }

    action sessionOpened(
        string sessionID, string messageID, boolean success, string error) {
        if (success) {
            log "Session Opened";
        } else {
            log "Session failed to open - " + sessionID + ", "
              + messageID + ", " + success.toString() + ", " + error;
        }
    }
}
```

Working with scalar values

The following example shows how to set a scalar value:

```
action putFloatExample() {
    matlabManager.putFloat(
        "Session1", "putFloatRequest", "x", 10.0, putFloatCallback);
}

action putFloatCallback(
    string sessionID, string messageID, boolean success, string error) {
    if (success) {
        log "Put Float Succeeded";
    } else {
        log "Put Float Failed - " + sessionID + ", " + messageID + ", "
          + success.toString() + ", " + error;
    }
}
```



```
}
```

The following example shows how to get a scalar value:

```
action getFloatExample() {
    matlabManager.getFloat(
        "Session1", "getFloatRequest", "x", getFloatCallback);
}

action getFloatCallback(string sessionID, string messageID, float value,
    boolean success, string error) {
    if (success) {
        log "Get Float Succeeded - value = " + value.toString();
    } else {
        log "Get Float Failed - " + sessionID + ", " + messageID + ", "
            + success.toString() + ", " + error;
    }
}
```

Working with arrays

To set an array:

```
action putFloatSequenceExample() {
    sequence<float> y := [0.0, 1.0, 2.71828, 3.14159];
    matlabManager.putFloatSequence("Session1", "putFloatSequenceRequest",
        "y", y, putFloatSequenceCallback);
}

action putFloatSequenceCallback(
    string sessionID, string messageID, boolean success, string error) {
    if (success) {
        log "Put Float Sequence Succeeded";
    } else {
        log "Put Float Sequence Failed - " + sessionID + ", "
            + messageID + ", " + success.toString() + ", " + error;
    }
}
```

To get an array:

```
action getFloatSequenceExample() {
    matlabManager.getFloatSequence(
        "Session1", "getFloatSequenceRequest", "y", getFloatSequenceCallback);
}

action getFloatSequenceCallback(string sessionID, string messageID,
    sequence<float> value, boolean success, string error) {
    if (success) {
        log "Get Float Sequence Succeeded - value = " + value.toString();
    } else {
        log "Get Float Sequence Failed - " + sessionID + ", "
            + messageID + ", " + success.toString() + ", " + error;
    }
}
```

Working with matrices

To set a matrix:

```
action putFloatMatrixExample() {
    sequence<sequence<float>> matrix := [];
    sequence<float> row1 := [-2.1, 3.5];
    sequence<float> row2 := [5.0, 1.0, 7.9, 17.0];
    sequence<float> row3 := [-20.0, -90.0, 25.0];

    matrix.append(row1);
    matrix.append(row2);
    matrix.append(row3);

    matlabManager.putFloatMatrix("Session1", "putFloatMatrixRequest",
        "m", matrix, putFloatMatrixCallback);
}

action putFloatMatrixCallback(
    string sessionId, string messageId, boolean success, string error) {
    if (success) {
        log "Put Float Matrix Succeeded";
    } else {
        log "Put Float Matrix Failed - " + sessionId + ", "
            + messageId + ", " + success.toString() + ", " + error;
    }
}
```

To get a matrix:

```
action getFloatMatrixExample() {
    matlabManager.getFloatMatrix(
        "Session1", "getFloatMatrixRequest", "m", getFloatMatrixCallback);
}

action getFloatMatrixCallback(string sessionId, string messageId,
    sequence<sequence<float>> value, boolean success, string error) {
    if (success) {
        log "Get Float Matrix Succeeded - value = " + value.toString();
    } else {
        log "Get Float Matrix Failed - " + sessionId + ", "
            + messageId + ", " + success.toString() + ", " + error;
    }
}
```

As well as setting MATLAB variables, applications may also send requests to the MATLAB plug-in to evaluate any appropriate MATLAB expressions using the `evaluate()` action.

The following example shows how to use the MATLAB plug-in to add two matrices and get the result:

```
action evaluateRequestExample() {
    // First matrix:
    sequence<sequence<float>> matrix1 := [];
    sequence<float> m1row1 := [1.0, 2.0, 3.0];
    sequence<float> m1row2 := [4.0, 5.0, 6.0];
    sequence<float> m1row3 := [7.0, 8.0, 9.0];
    matrix1.append(m1row1);
```

```

matrix1.append(m1row2);
matrix1.append(m1row3);

// The MATLAB manager also provides 'doesNothing*' callbacks that can
// process the returns silently if the response is not needed.
matlabManager.putFloatMatrix("Session1", "putFloatMatrixRequest1",
    "matrix1", matrix1, matlabManager.doesNothingCallback);

// Second matrix:
sequence<sequence<float>> matrix2 := [];
sequence<float> m2row2 := [2.0,5.0,8.0];
sequence<float> m2row3 := [3.0,6.0,9.0];
matrix2.append(m2row1);
matrix2.append(m2row2);
matrix2.append(m2row3);
matlabManager.putFloatMatrix("Session1", "putFloatMatrixRequest1",
    "matrix2", matrix2, matlabManager.doesNothingCallback);

// Use MATLAB to add the two matrices.
// The expected size of the string to be returned:
integer STANDARD_OUTPUT_SIZE := 256;

// Although use of the MATLAB plug-in is asynchronous, requests are
// queued. This guarantees that the two putFloatMatrix() actions
// have already been processed.
matlabManager.evaluate("Session1", "evaluateRequest",
    "matrix3 = matrix1 + matrix2", STANDARD_OUTPUT_SIZE, evaluateCallback);
}

action evaluateCallback(
    string sessionID, string messageID, string output,
    sequence<string> outputLines, boolean success, string error) {
    if (success) {
        matlabManager.getFloatMatrix(
            "Session1", "getMatrixRequest", "matrix3", getMatrix3Callback);
    } else {
        log "Evaluate Failed - " + sessionID + ", "
            + messageID + ", " + success.toString() + ", " + error;
    }
}

action getMatrix3Callback(string sessionID, string messageID,
    sequence<sequence<float>> value, boolean success, string error) {
    if (success) {
        log "Get Float Matrix Succeeded - value = " + value.toString();
    } else {
        log "Get Float Matrix Failed - " + sessionID + ", "
            + messageID + ", " + success.toString() + ", " + error;
    }
}
}

```

Using the R plug-in

Introduction to using the R plug-in

R is a free software environment for statistical computing. See <https://www.r-project.org/> for more information on R.

The Apama R plug-in connects to R using the Rserve package. See <https://rforge.net/Rserve/index.html> for more information on Rserve.

Prerequisites

You must install R, and you must install the Rserve package into R. After that, you can run Rserve using an appropriate configuration.

The Apama R plug-in uses a TCP connection to communicate with R. A host and port are supplied when opening the connection. The R server process can run locally on the same machine as Apama, or on a remote server. Rserve uses 6311 as the default port. You can change this default in the Rserve configuration and when creating the connection within Apama. See “[Steps for using the R plug-in](#)” on page 405 for further information.

Note:

Rserve has different session functionality, depending on whether the server is running on Windows or Linux. On Windows, new sessions are not created, if you re-connect to Rserve; previously set variables will still be valid.

To use the R plug-in, you need to add the **R Support** bundle to your Apama project (see also “[Adding the R Support bundle to your project](#)” on page 404). This lets you interact with a local or remote R running Rserve, setting and getting variables and evaluating R script from within EPL.

Adding the R Support bundle to your project

To use the R plug-in, you need only add the **R Support** EPL bundle to your project. For details, see “Adding EPL bundles to projects” in *Using Apama with Software AG Designer* or “Creating and managing an Apama project from the command line” in *Deploying and Managing Apama Applications*.

Adding the **R Support** bundle to your project makes the `RPlugin.mon` file available to the monitors in your project. When you run your project, Software AG Designer automatically injects the `RPlugin.mon` file. If you want to examine this file, you can find it in the `monitors/R` directory of your Apama installation.

The file `RPlugin.mon` is the interface between the monitors in your application and the R plug-in. Your application creates an `RConnection` event using the static `open()` action in the `RFactory` event, and it then calls `set()` or `evaluate()` actions on the `RConnection` event to communicate with R. There is never any need to import or call the plug-in directly.

Steps for using the R plug-in

The events within the **R Support** bundle are in the `com.apama.r` package. It is assumed that R has been installed with Rserve and that it is running on a known host and port (see [“Introduction to using the R plug-in” on page 404](#) for more information).

After you have added the **R Support** bundle, you write EPL that does the following:

1. Create an `RConnection` event using the static `open()` or `openDefault()` action in the `RFactory` event with the known host and port. On Linux, the connection can be made using local sockets if the port is set to `-1` and if the host is set to the socket filename.
2. Use the `set()` action in `RConnection` to set named variables in R. The `set()` action uses the any type, which means that all types can be passed into it.

Supported types are `integer`, `float`, `string`, `boolean` and `sequence`, including nested sequences or sequences of different types using `any`.

`decimal` and `dictionary` types are not supported. All unsupported types will cause an exception to be thrown.

3. Use the `evaluate()` action to evaluate an R expression and return the result. This action returns an any type and the user must process the result to the correct type.

Or use the `evaluateAstype()` action to evaluate an R expression, casting the result to the requested type. If the type is not correct, an exception will be thrown.

4. Close the connection using `close()`.

See the *API Reference for EPL (ApamaDoc)* for more information on these actions.

Note:

`open()` can be called from different contexts. However, a matching host/port to an open connection will share the same connection.

Within R, everything is an array and R does not differentiate between a single value or an array which holds a single value. Because of this, the Apama R plug-in converts an array of a single value into a single value of the correct type. Therefore, if you set a `sequence<integer>` variable to `[123]`, it will just be an integer of 123 when you retrieve this value.

Calls to the R plug-in are synchronous and block execution in the current context. You can create an asynchronous service in a dedicated context; a sample is provided to demonstrate this. See also [“R plug-in samples” on page 405](#).

R plug-in samples

Sample code for using the R plug-in is provided in the `samples/epl/RPlugin` directory of your Apama installation. See the `README.txt` file in that directory for more information.

- The file `RPluginSample.mon` demonstrates how to set and get variables of different types from R, how to create a matrix using an R script and get the dimensions, and how to load an R script from an external file.
- The file `AsyncR.mon` demonstrates an asynchronous R service running in a dedicated context where requests and responses are sent via events.

Interfacing with user-defined EPL plug-ins

Although EPL is very powerful and enables complex applications, it is foreseeable that some applications might require additional specialized operations. For example, an application might need to carry out advanced arithmetic operations that are not provided in EPL. You can address this situation by writing custom EPL plug-ins using Apama's C++, Java, or Python plug-in development kits. For detailed information on developing your custom EPL plug-in, see .

Note:

The correlator's plug-in interface is versioned. If you upgrade the major or minor version of Apama, you may need to recompile your plug-ins against the new libraries to be compatible with the newer version of the correlator.

In order to access a function implemented in an EPL plug-in, you must import the plug-in. If the plug-in is written in Java, you must first inject the jar file containing the plug-in. For any plug-in, an EPL monitor or event must use the `import` statement to load it:

```
import "apama_math" as math;
```

For a Java plug-in, this will load the plug-in from the injected jar file. For a C++ plug-in, this will look for the Apama plug-in file `libapama_math.so` (on Linux) or for `apama_math.dll` (on Windows). These must be located on the standard library path (by default, in `$APAMA_WORK/lib`). It will then map it to the internal alias `math`.

Note:

Insert the `import` statement in the monitor that uses the plug-in functions.

If the `apama_math` plug-in defines a method called `cos` that takes a single floating point value as an argument and returns a `float` value, this would be called from EPL as follows:

```
float a, b;  
// ... some other EPL  
a := math.cos(b);
```

Standard `float`, `integer` and `boolean` types are passed by-value to external functions while `string` and sequence types (which map to native arrays in the plug-in) are passed by-reference. In addition, the `chunk` type can be used to “pass-through” data returned from one function call to another plug-in function, as shown in [“About the chunk type” on page 406](#).

About the chunk type

The `chunk` type allows data to be referenced from EPL that has no equivalent EPL type. It is not possible to perform operations on data of type `chunk` from EPL directly; the `chunk` type exists purely

to allow data output by one external library function to pass through to another function. Apama does not modify the internal structure of chunk values in any way. As long as a receiving function expects the same type as that output by the original function, any complex data structure can be passed around using this mechanism.

To use chunks with plug-ins, you must first declare a variable of type chunk. You can then assign the chunk to the return value of an external function or use the chunk as the value of the *out* parameter in the function call.

The following example illustrates this. The functions `setRealVal` and `setImagVal` set internal values of the chunk, while the functions `getRealVal` and `getImagVal` retrieve values from the chunk. The function `addComplexNumber` adds the second chunk to the first chunk. The source code for `complex_plugin` is in the `samples\correlator_plugin\cpp` directory of your Apama installation directory.

```
Monitor ComplexPlugin {
    import "complex_plugin" as plugin;

    action onload {
        // Creates a first complex number chunk
        chunk complexNumberFull := plugin.makeComplexNumberFull(4.0, -1.4);
        printComplexNumber(complexNumberFull);

        // Creates a second complex number chunk
        chunk complexNumberEmpty := plugin.makeComplexNumberEmpty();

        // Get the real and imaginary values of the number
        plugin.setRealVal(complexNumberEmpty, 2.0);
        plugin.setImagVal(complexNumberEmpty, 3.0);
        printComplexNumber(complexNumberEmpty);

        // Add the second complex number to the first complex number
        plugin.addComplexNumber(complexNumberFull, complexNumberEmpty);
        printComplexNumber(complexNumberFull);
    }

    action printComplexNumber(chunk complexNumber)
    {
        float real := plugin.getRealVal(complexNumber);
        float imag := plugin.getImagVal(complexNumber);
        string sign := "";
        if(imag >= 0.0) {
            sign := "+";
        }
        log real.toString() + sign + imag.toString() + "i";
    }
}
```

Although the chunk type was designed to support unknown data types, it is also a useful mechanism to improve performance. Where data returned by external plug-in functions does not need to be accessed from EPL, using a chunk can cut down on unnecessary type conversion. For example, suppose the output of a `localtime()` method is a 9-element array of type `float`. While you could declare this output to be of type `sequence<float>`, there is no need to do so because the EPL never accesses the value. Consequently, you can declare the output to be of type chunk and avoid an unnecessary conversion from native array to EPL sequence and back again.

11 Making Application Data Available to Clients

■ Adding the DataView Service bundle to your project	411
■ Creating DataView definitions	411
■ Deleting DataView definitions	412
■ Creating DataView items	412
■ Deleting DataView items	413
■ Updating DataView items	414
■ Looking up field positions	415
■ Using multiple correlators	415

DataViews are an Apama abstraction that allows an application to present a read-only view onto some of their data for easy consumption by external clients that use Apama's Scenario Service API (see "Scenario Service API" in *Connecting Apama Applications to External Components*), such as Dashboard Builder dashboards.

External clients using the Scenario Service can be notified when instances are added and deleted, and when the outputs of a specific instance change. However, they cannot create, delete or edit instances themselves, as DataViews are read-only views and instance management is under the control of the application.

This topic is about the DataView Service, which provides an EPL API for creating/deleting/updating DataView definitions, instances and instance outputs such that they can be consumed by Scenario Service clients. The DataView Service is not the only way Apama exposes DataViews to the Scenario Service. MemoryStore tables, for example, can be configured to expose their rows as DataView instances (see [“Exposing in-memory or persistent data as DataViews” on page 363](#)).

The DataView Service uses two central concepts:

■ **DataView definition**

A DataView definition specifies a unique DataView name, a set of field names and field types (each type is one of `string`, `decimal`, `float`, `integer`, and `boolean`), and optionally a set of key fields.

■ **DataView item**

Each DataView item is associated with a DataView definition, and specifies values for the defined fields.

Note:

The topics below briefly describe the event types that are used for managing the DataView definitions and items. For detailed information on the fields that are available for these events, see the `com.apama.dataview` package in the *API Reference for EPL (ApamaDoc)*.

Note that a DataView definition is not intended to serve as a central data structure for your application, but rather is intended merely to expose your application's data to remote client applications.

The programming interface is defined by the `DataViewService_Interface.mon` file in the `monitors` directory of your Apama installation directory. It defines the API for working with DataView definitions and DataView items.

Using the DataView Service, you can create DataViews in only the main context. You cannot create them in any contexts you create.

Metadata properties can be specified for a DataView by adding keys with the prefix `DataViewDefinition.EXTRA_PARAMS_METADATA_PREFIX` to the `extraParams` dictionary of `DataViewAddDefinition` when adding the new DataView definition.

Adding the DataView Service bundle to your project

To use the `DataViewService`, you have to add the **DataView Service** bundle to your Apama project. Adding this bundle ensures that the following EPL files are loaded before any monitors that use them. These monitors are in the `monitors` directory of your Apama installation:

- `ScenarioService.mon`
- `DataViewService_Interface.mon`
- `DataViewService_Impl_Dict.mon`

Note:

The `DataViewService` is designed primarily to interact with other EPL or JMon applications that reside in the same correlator. However, it can also be used with multiple correlators. See [“Using multiple correlators” on page 415](#) for further information.

The description below explains how to add the bundle using Software AG Designer, but you can also add it using the `apama_project` command-line tool as described in "Creating and managing an Apama project from the command line" in *Deploying and Managing Apama Applications*.

➤ To add the DataView Service bundle to your Apama project

1. In Software AG Designer, go to the Apama Developer perspective.
2. In the **Project Explorer**, right-click the **EPL Bundles** node and select **Add Bundle**.
3. Select the **DataView Service** bundle and click **OK**.

Creating DataView definitions

Use the following event types to create DataView definitions.

DataViewAddDefinition

Create and route an event of this type in order to create a DataView definition. The response is provided by a `DataViewDefinition` or `DataViewException` event.

DataViewDefinition

These events are responses to `DataViewAddDefinition` events. They indicate the successful creation of a DataView definition. The contents of the fields are exactly those of the `DataViewAddDefinition` event to which this is a response, except possibly for *extraParams*.

DataViewException

These events occur under exceptional circumstances in response to `DataViewAddDefinition` or `DataViewDeleteDefinition` events, or any circumstance under which a `DataView` cannot be identified.

Here is an example of creating a `DataView` definition and handling `DataViewException` events:

```
using com.apama.dataview.DataViewAddDefinition;
using com.apama.dataview.DataViewException;
...
DataViewAddDefinition add := new DataViewAddDefinition();
add.dvName := "Weather";
add.dvDisplayName := "Weather";
add.fieldNames := ["location", "temperature", "humidity", "visibility"];
add.fieldTypes := ["string", "integer", "integer", "integer"];
add.keyFields := ["location"];
route add;
on all DataViewException() as dvException {
  log "*** Weather monitor error: " +
    dvException.toString() at ERROR;
}
```

Deleting DataView definitions

Use the following event types to delete `DataView` definitions.

DataViewDeleteDefinition

Create and route events of this type in order to delete a `DataView` definition. The response is provided by a `DataViewDefinitionDeleted` or `DataViewException` event.

DataViewDefinitionDeleted

These events are responses to `DataViewDeleteDefinition` events. They indicate the successful deletion of a `DataView` definition.

Creating DataView items

Use the following event types to create `DataView` items.

DataViewAddItem

Create and route an event of this type to create a `DataView` item. This item must not exist already. A response is provided by a `DataViewItem` or `DataViewException` event.

Here is an example that creates and routes a `DataViewAddItem` event, and handles the `DataViewItem` response by logging the addition of the item:

```
using com.apama.dataview.DataViewAddItem;
```

```

using com.apama.dataview.DataViewItem;
...
string location ;
integer temp;
integer humidity;
integer visibility;
...
DataViewItem item := new DataViewItem();
item.dvName := "Weather";
item.fieldValues :=
    [location,temp.toString(),humidity.toString(),
     visibility.toString()];
route item;
on DataViewItem (dvName="Weather") as added {
    log("Weather monitor - DataViewItem: " +
        added.dvItemId.toString());
}

```

DataViewItemAddOrUpdateItem

Create and route an event of this type to create a `DataViewItem` item if it does not already exist, or update a `DataViewItem` item if it already exists. A response is provided by a `DataViewItem` or `DataViewItemException` event.

This will only work when `keyFields` are used. Any attempts to change the owner of an existing item will be rejected with a `DataViewItemException`.

DataViewItem

These events are responses to `DataViewItemAddItem` events. They indicate the successful creation of a `DataViewItem` item. The contents of the fields are exactly those of the `DataViewItemAddItem` event to which this is a response, except possibly *extraParams*, and with the addition of the `dvItemId` field.

DataViewItemException

These events occur under exceptional circumstances in response to `DataViewItemDeleteItem`, `DataViewItemUpdateItem` or `DataViewItemUpdateItemDelta` events.

Deleting DataView items

Use the following event types to delete `DataViewItem` items.

DataViewItemDeleteItem

Create and route an event of this type to delete a `DataViewItem` item. A response is provided by a `DataViewItemDeleted`, `DataViewItemException` or `DataViewItemException` event.

Here is an example that creates and routes a `DataViewItemDeleteItem` event and handles the `DataViewItemDeleted` response by logging the deletion of the item:

```

using com.apama.dataview.DataViewItemDeleteItem;
using com.apama.dataview.DataViewItemDeleted;

```

```
string location;
...
DataViewDeleteItem delete := new DataViewDeleteItem;
delete.dvName := "Weather";
delete.dvItemId := -1; // Set the ID to -1 when using keyFields
delete.keyFields := [location];
route delete;
on DataViewItemDeleted (dvName="Weather") as deleted {
    log("Weather monitor - DataViewItemDeleted:
        "+deleted.dvItemId.toString());
}
```

DataViewItemDeleted

These events are responses to `DataViewDeleteItem` events. They indicate the successful deletion of a `DataView` item.

DataViewDeleteAllItems

Create and route an event of this type to delete all `DataView` items associated with a specified `DataView` definition. A response is provided by a `DataViewAllItemsDeleted`, `DataViewException` or `DataViewItemException` event.

DataViewAllItemsDeleted

These events are responses to `DataViewDeleteAllItem` events. They indicate the successful deletion of all items associated with a given `DataView` definition.

Updating DataView items

Use the following event types to update `DataView` definitions.

Note:

In addition to the event types listed below, you can also use the `DataViewAddOrUpdateItem` event to either create new `DataView` items or to update existing ones. See [“Creating DataView items” on page 412](#).

DataViewUpdateItem

Create and route an event of this type to update a data item by specifying a sequence of new field values. If the update does not succeed, a response is provided by a `DataViewItemException` event.

Here is an example of creating and routing a `DataViewUpdateItem` event:

```
using com.apama.dataview.DataViewUpdateItem;
...
string location;
integer temp;
integer humidity;
integer visibility;
...
```

```

DataViewUpdateItem update := new DataViewUpdateItem;
update.dvName := "Weather";
update.dvItemId := -1; // Set the ID to -1 when using keyFields
update.fieldValues :=
    [location,temp.toString(),humidity.toString(),visibility.toString()];
route update;

```

DataViewUpdateItemDelta

Create and route an event of this type to update a data item by specifying a dictionary of field-position/field-value pairs. If the update does not succeed, a response is provided by a `DataViewItemException` event.

Here is an example of creating and routing a `DataViewUpdateItemDelta` event:

```

using com.apama.dataview.DataViewUpdateItemDelta;
...
string location;
integer temp;
integer humidity;
integer visibility;
...
DataViewUpdateItemDelta update := new DataViewUpdateItemDelta;
update.dvName := "Weather";
update.dvItemId := -1; // Set the ID to -1 when using keyFields.
update.fieldValues :=
    {0:location,1:temp.toString(),2:humidity.toString(),
     3:visibility.toString()};
route update;

```

Looking up field positions

Use the following event types to look up the numerical position of a given field-name for a given `DataView` definition.

DataViewGetFieldLookup

Create and route an event of this type to request a helper dictionary that supports lookup of field position for a given field name. The response is provided by a `DataViewFieldLookup` or `DataViewItemException` event.

DataViewFieldLookup

These events are responses to `DataViewGetFieldLookup` events. They contain a dictionary that supports lookup of the field position for a given field name.

Using multiple correlators

The `DataViewService` is designed to primarily interact with other EPL or JMon applications that reside in the same correlator. Therefore, the `DataViewService` implementation does not emit any

events. You can inject the following optional additional monitors, which are in the `monitors` directory of your Apama installation, to emit the events when that is required:

- `DataService_ServiceEmitter.mon`
- `DataService_ApplicationEmitter.mon`

This enables Dashboard Builder clients to visualize the state of a number of applications, each of which is running in a separate correlator, and each of which may fail-over to another correlator. Since configuring all of the dashboards to know about each of these correlators might be difficult and fragile, you can designate an additional single correlator as the “view correlator”, which holds the `DataService` and `ScenarioService` to which any client dashboard can connect.

With this architecture, the individual applications in the separate correlators need to emit `DataService` request events to a channel that has been connected to the view correlator. These applications can either emit the events directly, or with the Application Emitter injected they can route the events and the extra monitor will emit them to the channel. The `DataService` in the view correlator routes its responses (as normal), but the Service Emitter monitor will then emit those events out on the `com.apama.dataview` channel so that the originating correlators can receive them.

Note that these two emitters are entirely optional, and are not required for most deployments. Moreover, you normally do not inject these two monitors into the same correlator. Also, there is no bundle in Software AG Designer that provides these monitors.

12 Testing and Tuning EPL monitors

■ Optimizing EPL programs	418
■ Best practices for writing EPL	418
■ Structure of a basic test framework	420
■ Using event files	420
■ Handling runtime errors	421
■ Capturing test output	423
■ Avoiding listeners and monitor instances that never terminate	423
■ Handling slow or blocked receivers	424
■ Diagnosing infinite loops in the correlator	424
■ Tuning contexts	425

This section provides information about testing and tuning your EPL monitors.

Optimizing EPL programs

Best practices for optimizing EPL programs include:

- Minimize cost of spawning — avoid repeated spawning of monitors that contain a large number of variables.
- Allocate events — but not unnecessarily. See [“Avoiding unnecessary allocations” on page 419](#).
- Specify wildcard on non-essential event fields. See [“Wildcard fields that are not relevant” on page 419](#).
- Use plug-ins when you cannot write efficient EPL to accomplish your purpose. See [“When to use plug-ins” on page 340](#).
- Minimize the effect of garbage collection

EPL, like languages such as Java or C#, relies on garbage collection. Intermittently, the correlator analyses the objects that have been allocated, including events, dictionaries and sequences, and allows memory used by objects that are no longer referenced to be re-used. Thus, the actual memory usage of the correlator might be temporarily above the size of all live objects. While running EPL, the correlator might wait until a listener, `onload()` action or stream network activation completes before performing garbage collection. Therefore, any garbage generated within a single action, listener invocation or stream network activation might not be disposed of before the action/listener/activation has completed. It is thus advisable to limit individual actions/listeners/activations to performing small pieces of work. This also aids in reducing system latency.

The cost of garbage collection increases as the number of events a monitor instance creates and references increases. If latency is a concern, it is recommended to keep this number low, dividing the working set by spawning new monitor instances if possible and appropriate. Reducing the number of object creations, including string operations that result in a new string being created, also helps to reduce the cost of garbage collection. The exact cost of garbage collection could change in future releases as product improvements are made.

Best practices for writing EPL

EPL is a programming language with some special features. As such, it shares the characteristic with every other programming language that it is possible to write poor, inefficient code. All the techniques that apply to other languages to minimize wasted cycles can also be applied to EPL.

Basic programming optimization techniques all apply:

- Move code out of tight loops
- Avoid unnecessary allocation, for example, strings
- Put common tests first in `if ... else` form

There is no substitute for empirical evaluation of the performance of your application. You must measure performance and compare measurements when modifications are made. Also, ensure that you are comparing like-with-like. Understanding performance implications is invaluable and it helps in avoiding unnecessary performance costs.

You should know how fast your application needs to be.

Wildcard fields that are not relevant

Once a design has stabilized and event interfaces are well defined, it is possible to wildcard fields that do not need to be matched on in event listeners. Designating an event field as a wildcard prevents the correlator from creating an index for that field. Most importantly, a wildcard field means that the correlator does not need to traverse that index when receiving an event of that type to try to find interested event listeners (as there will not be any). This can give tangible performance benefits, particularly with large events.

Premature wildcarding is not advised but is not harmful. You can easily remove the `wildcard` annotation from event fields with no impact on existing code. The compiler gives an error if any code attempts to match on a field that is a wildcard.

The correlator can index up to 32 fields for each event type. If you are using an event that has more than 32 fields, you must designate the additional fields as wildcards.

See [“Improving performance by ignoring some fields in matching events”](#) on page 159.

Avoiding unnecessary allocations

You should eliminate unnecessary allocations, especially when the size of an event is very large. For example:

```
event LargeEventWith1000Fields {}           // field definitions omitted

integer i := 0;
while (i < 1000) {
    route LargeEvent(0,0,i, ...);           // bad
    i := i + 1;
}

LargeEvent le := new LargeEvent();          // good
while (i < 1000) {
    le.foo := i;
    route le;
    i := i + 1;
}
```

Implementing states

When you want to write a process that passes through one or more states it is good practice to have one action per state. For example:

```
action inAuction() {
    on AuctionClosed outOfAuction();
}
```

```
}  
  
action outOfAuction() {  
    on all Price (stock,*) as p and not InAuction() {  
        on Price(stock,>p.price*1.01) and not InAuction() {  
            sellStock();  
        }  
    }  
    on InAuction() inAuction();  
}
```

Structure of a basic test framework

Apama lends itself to automated testing because

- You can define test cases in event files that you feed into the correlator.
- Apama includes a comprehensive set of command line utilities, all of which are scriptable using standard scripting languages on different platforms.
- The correlator is deterministic when there is only the main context. When there is more than one context, each context is deterministic but the correlator as a whole is not.

If the advocated event interface pattern is employed for encapsulation, then modules can be tested in isolation (unit testing) as well as in more comprehensive integration-level tests.

A basic test case includes the following:

- EPL files (.mon) to deploy (or references to them).
- Input event files (.evt) to send to the correlator.
- Reference event files (.evt) to compare to actual output.
- Script to orchestrate execution of the test-case.

You should assemble all of these files in an Apama project in Software AG Designer and then use Software AG Designer to launch the test case.

Each test-case can reside in its own project with all relevant files local to it. The basic test process is to launch the application, send in some events, capture outputs, then compare to expected output, printing the results of the test to the console or log file at the minimum.

Using event files

The following example shows how to use &TIME (Clock) events to explicitly set the correlator clock. To do this, the correlator must have been started in external clocking mode (the &TIME events give errors otherwise). Times are in seconds since the midnight, Jan 1970 epoch.

```
#seed initial time (seconds since Jan 1970 epoch)  
&TIME(1)  
  
# Send in configuration of heartbeat interval to  
# 5 SecondsSetHeartbeatInterval(5.0)
```

```
# Advance the clock (5.5 seconds)
&TIME(6.5)

# Correlator should have sent heartbeat with id 1 -
# acknowledge all is well
HeartbeatResponse(1,true)
```

Notice that the input event file has a lot of knowledge regarding the way in which the module will (should) respond. For example, the `HeartbeatResponse` event expects that the first `HeartbeatRequest` will have the ID of 1. There is necessarily a close coupling between the input scripts and the implementation of the module being tested. This is another reason why as much of this information should be extracted into the module's message exchange protocol and made explicit, and perhaps enforced by one or more interface intermediaries.

A single correlator context is guaranteed to generate the same output in the same order, even when EPL timers (such as `on all wait()`) are employed. This is a benefit of correlator determinism, and makes regression testing, even of temporal logic, possible.

Note:

The correlator's behavior can be nondeterministic when events are sent between multiple contexts, or when plug-ins are used.

Handling runtime errors

EPL eliminates many runtime errors because of the following:

- Strict, static typing, so there are no class cast exceptions.
- No implicit type conversion so there are no number format exceptions.
- Values can only be “null” (or “empty” as it is called in EPL) if they are explicitly declared as `optional`, and can be accessed safely without risk of null pointer exceptions. See the description of the `optional` type in the *API Reference for EPL (ApamaDoc)* and [“The ifpresent statement” on page 653](#) for more information.

However, EPL cannot entirely eliminate runtime errors. For example, you receive a runtime error if you try to divide by zero or specify an array index that is out of bounds. Some runtime errors are obscure. For example:

```
mySeq.remove(mySeq.indexOf("foo"));
```

If `foo` is not in `mySeq`, `indexOf()` returns `-1`, which causes a runtime error.

See also [“Exception handling” on page 276](#).

What happens

When the correlator detects a runtime error, an exception is raised. If that is not caught by a `catch` block, the correlator kills the monitor instance that contains the code that caused the error. This protects the other monitor instances that are running in the same correlator. Upon an unhandled

runtime error, the correlator also terminates any listeners that were set up by the monitor instance being killed, and the state of the killed monitor is lost.

Using `ondie()` to diagnose runtime errors

The preferred method for handling runtime errors is using a catch block; see also [“Exception handling” on page 276](#).

If you are not sure where the catch block is necessary, you can specify some logging in the `ondie()` action to help diagnose the problem and to alert other system modules that a problem occurred. For example:

```
action ondie() {  
    log "monitor instance terminating for " + myId;  
    route InternalError("Foo");  
}
```

In some circumstances, you can move into a suspended or safe state, or initiate damage limitation activities, for example, such as pulling all active orders from the market. For example, Apama scenarios use the `ondie()` action to route an `InstanceDied()` event to a `ScenarioService` monitor. This in turn sends the event to connected clients so the termination of the instance can be handled, perhaps displayed, in a dashboard.

An alternative to using `ondie()` in this manner is to use a basic ACK, NACK, and timeout message exchange protocol so that a client is robust against its services being unavailable.

See [“About executing `ondie\(\)` actions” on page 44](#) for information about how `ondie()` can optionally receive exception information if an instance dies due to an uncaught exception.

Using logging to diagnose errors

Logging is an effective means of generating diagnostic information. When writing log entries, consider the overhead of string allocation, garbage collection, and writing data to disk. Use conditional tests to reduce this overhead and minimize unnecessary logging.

The EPL `log` statement is a simple means of generating logging output. The EPL `log` statement writes to the correlator log file by default so any messages your program sends to the log file are mixed in with all other correlator logging messages. However, you can configure the correlator to send your EPL logging to a separate file. See [“Setting EPL log files and log levels dynamically” in *Deploying and Managing Apama Applications*](#). The logging attributes you can specify include a particular target log file and a particular log level for any number of individual packages, monitors and events.

When sending messages to the correlator log file, consider the following:

- Log messages can be lost if the correlator is logging to `stdout`.
- Using the correlator log is relatively expensive if there are many `log` statements in the critical path.
- Anything you send to the log might be lost if the correlator log level is `OFF`.

See also [“Logging and printing” on page 279](#).

Standard diagnostic log output

By default, the correlator outputs diagnostic information every five seconds, and sends it to the correlator log file at `INFO` log level. You can use this information to diagnose common problems. See “Descriptions of correlator status log fields” in *Deploying and Managing Apama Applications* for further information.

The correlator sends this information to its log file during normal operation. While it is possible to disable this output (by setting the correlator's log level to `WARN`), doing so is not advisable. In the unlikely event that you run into a problem, Apama Technical Support always ask for a copy of this log file, as the information in it is often useful for diagnosing the nature of a failure.

Capturing test output

All receivers should be started before any events are sent in to the correlator and set to write events to file. The file(s) can be easily compared to reference output using standard operating system tools.

Other tools are also useful in checking the output. The `engine_inspect` correlator utility is good for verifying that the right number of monitor instances and listeners is present after (stages of) a test. Also, you can use this utility to detect listeners and monitor instances that never terminate, or premature existence of monitor instances.

Use the `engine_receive` utility to capture event output. You can specify the `-f` option to pipe received events to a file. Start multiple receivers on different channels as required

The `engine_inspect` utility provides useful data for testing including the number of monitor instances, listeners, receivers, events generated and so on. Split input event files and run the `engine_inspect` utility after each file.

Capture the correlator log and compare to reference data. This is useful if your application logs errors or there are interesting diagnostics.

Avoiding listeners and monitor instances that never terminate

An Out of Memory condition causes the correlator to exit. This condition can be caused by listeners and monitor instances that never terminate — also referred to as listener leaks. For example, the following on statement defines event listeners that never terminate:

```
on all ( Foo(id=1) or all Foo(id=2) ) {      // second "all" is bogus
...
}
```

The following example spawns monitor instances that never terminate:

```
on all Trade() as t spawn handle();          // missing "unmatched" action
```

```
...  
action handle() {  
    on all Trade(symbol=t.symbol) as t {  
        ...  
    }  
}
```

The `sm` (number of monitor instances) and `ls` (number of listeners) counts in the log file are often revealing in the case of a memory leak. An increasing trend can be seen in these counts over a period of time, when there is no valid reason for this given the intended logic of the application.

Handling slow or blocked receivers

You can use correlator diagnostic output to identify slow or blocked receivers.

- The `oc` (number of events on the output queue) can grow to 10,000 maximum. If you see a steady trend that it is growing, it probably indicates a slow receiver.
- The `tx` (number of events transmitted) should always be increasing. If it is static, or not increasing as fast as it should, it probably indicates a slow receiver.

Slow receivers include:

- Receivers that are not consuming events as quickly as the correlator is generating them.
- Blocked receivers that are not accepting new events.

When the correlator's output queue fills, operations that are sending events from the processing thread (or threads, if there is more than one context) are blocked. If the output queue remains filled, and the processing thread(s) remain blocked, the input queue(s) start(s) to fill. Events are never dropped.

If you specify the `-x` correlator option when you start the correlator, it causes the correlator to disconnect any receiver that becomes slow. If you discover that your application is producing events at too high a rate for a particular receiver you might be able to publish the events to separate channels so that the receiver needs to handle fewer events. Alternatively, or in addition, you might be able to modify your application to throttle the rate at which it sends events to this receiver.

If you cannot speed the receiver up, or install faster hardware, you can partition the correlator's output event flow into channels so that the receiver needs to handle fewer events. Alternatively, you can use throttling in the correlator to output events less frequently.

See also "Determining whether to disconnect slow receivers" in *Deploying and Managing Apama Applications*.

Diagnosing infinite loops in the correlator

A correlator live lock occurs when events are recursively routed without a termination mechanism. The following example shows this in its simplest form:

```
on all Foo() {  
    route Foo();  
}
```



```
}
```

More complex forms might recurse after a connected chain of several events being routed between different monitors.

There are no limits on how many routed events can be queued. Consequently, depending on the nature of the bug, the correlator might run out of memory. Note that an overloaded correlator would show similar symptoms, but can be distinguished by the fact that work is still being done (events are being sent out from the correlator).

When the correlator is in an infinite loop, it quickly uses an entire CPU and if there are events being routed as part of the loop then the correlator will run out of memory. Use the following correlator diagnostics to diagnose an infinite loop:

- `rq` — sum of the number of routed events on the input queues of all contexts. When the correlator is in an infinite loop, this will always be 1 or it will always be increasing. It depends on the application.
- `iq` — sum of the number of entries on the input queues of all contexts. When the correlator is in an infinite loop, this number is continuously increasing.
- `tx` — number of transmitted events. This number is static when the correlator is in an infinite loop.

To identify an infinite loop in a particular context, run `engine_inspect -x` a few times. This lists each context along with the number of events on its input queue. See if there are contexts that have input queues that are getting bigger and bigger.

Tuning contexts

You should implement contexts whenever you want the correlator to perform concurrent processing. Work to be divided among contexts should have minimum or no interdependencies and no ordering requirements. Many applications present a natural way to partition work that is largely independent. For example, you could partition a financial application by stock symbol, or by user, or by strategy.

The following topics describe common ways to optimize use of contexts.

Parallel processing for instances of an event type

A candidate for implementing parallel processing is when an application performs calculations for a number of events that are of the same type, but that have different identifiers. For example, different stock symbols from a stock market data feed. You can use either of the following strategies to implement parallel processing for this situation:

- Create multiple public contexts. Each context listens for one identifier, operates on the events that have that identifier, and discards events that have any other identifier.
- Have one context distribute data to multiple contexts, which are each dedicated to processing the events that have a particular identifier.

The performance of these strategies varies according to the work being done. A distributor can be a bottleneck. However, there is a cost in every context discarding events for which it is not interested. In the following situations, the distributor strategy is likely to be more efficient:

- There is a very large set of identifiers but a relatively low overall rate of arriving events.
- Events must be pre-processed.
- Events are not arriving from external sources. Instead, you must explicitly send events.

The sample code below shows the distributor strategy.

```
event Tick {
    string symbol;
    integer price;
}

/** In the main context, the following monitor distributes Tick events
    to other contexts. There is one context to process each unique symbol. */
monitor TickDistributor {

    /** The dictionary maps each unique Tick symbol to the (private)
        context that ultimately processes it. */
    dictionary<string, context> symbolMapping;

    action onload() {
        on all Tick() as t {
            // If the context for this symbol does not yet exist, create it.
            if(not symbolMapping.hasKey(t.symbol)) {
                context c := context("Processing-"+t.symbol);
                symbolMapping[t.symbol] := c;
                spawn processSymbol(t.symbol) to c;
            }

            // Send each Tick event to the context that handles its symbol.
            send t to symbolMapping[t.symbol];
        }
    }

    /** The following action handles Tick events with the given symbol.
        This action executes in a private context that processes all Tick
        events that have one particular symbol. */
    action processSymbol(string symbol) {
        // Because this context receives a homogeneous stream of Tick events
        // that all have the same particular symbol, there is no need to specify
        // an event listener that discriminates based on symbol.
        on all Tick() as t {
            ...
        }
    }
}
```

Parallel processing for long-running calculations

Suppose a required calculation takes a relatively long time. You can do the calculation in a context while the main context performs other operations. Or, you might want multiple contexts to concurrently perform the long calculation on different groups of the incoming events.

The following code provides an example of performing the calculation in another context.

```
monitor parallel {
    integer numTicks;
    action onload() {
        on all Tick() {
            numTicks:=numTicks+1;
            send NumberTicks(numTicks) to "output";
        }
        on all Calculate() as calc {
            integer atNumTicks:=numTicks;
            integer calcId:=integer.getUnique();
            spawn doCalculation(calc, calcId, context.current())
                to context("Calculation");
            on CalculationResponse(id=calcId) as resp {
                send CalculationResult(resp, atNumTicks, numTicks) to "output";
            }
        }
    }
    action doCalculation(Calculate req, integer id, context caller) {
        float value:=actual_calculation_function(req);
        send CalculationResponse(id, value) to caller;
    }
}
```

For each Calculate event found, the event listener specifies a `spawn...to` statement that creates a new context. All contexts have the same name — Calculation — and a different context ID. All contexts can run concurrently.

A Calculation context might send a CalculationResponse event to the main context before the main context sets up the CalculationResponse event listener. However, the correlator completes the operations, including setting up the CalculationResponse event listener, that result from finding a Calculate event before it processes the sent CalculationResponse event.

While the calculations are running, other Tick events might arrive from external components and the correlator can process them.

The order in which CalculationResponse events arrive on the main context's input queue can be different from the order of creation of the contexts that generated the CalculationResponse events. The order of responses depends on when the calculation started and how long it took to complete the calculation. The monitor instance in the main context uses the `calcId` variable to distinguish responses.

13 Generating Documentation for Your EPL Code

■ Code constructs that are documented	430
■ Steps for using ApamaDoc	430
■ Inserting ApamaDoc comments	431
■ Inserting ApamaDoc tags	432
■ Inserting ApamaDoc references	435
■ Inserting EPL source code examples	436
■ Generating ApamaDoc from the command line	437
■ Generating ApamaDoc from an Ant script	439

Just as you can use the Javadoc tool to generate documentation for Java, you can use ApamaDoc to generate documentation for EPL. ApamaDoc, which is based on Javadoc, generates reference documentation from EPL source code. To enhance what ApamaDoc automatically generates, you can insert annotations in block comments. Annotations are a mixture of text and tags.

ApamaDoc is an export wizard in Software AG Designer. It generates static HTML pages that document the structure of all EPL code in a project. This includes the `.mon` files that you create as well as all `.mon` files in all bundles that have been added to a project.

Alternatively, you can generate ApamaDoc using the `apamadoc` command-line tool or from an Ant script.

Note:

ApamaDoc does not operate on `.qry` files. That is, you cannot use ApamaDoc to generate reference documentation for Apama queries.

Code constructs that are documented

The ApamaDoc generates documentation for the following code constructs:

- Monitors
- Events (defined outside monitors)
- Fields, constants and actions on events
- Custom aggregate functions
- Fields, constants and actions on custom aggregate functions
- Plug-in import statements

By default, the ApamaDoc export wizard does not generate documentation for inner fields or events of a monitor. If you want to include inner fields and events of a monitor, generate ApamaDoc without a user interface using the `--includeMonitorMembers` option. See [“Generating ApamaDoc from the command line” on page 437](#) or [“Generating ApamaDoc from an Ant script” on page 439](#) for more information.

To find out how ApamaDoc can be useful for documenting the storage and flow of personal data within your application, see “Documenting personal data flows within an Apama application” in *Developing Apama Applications*.

Steps for using ApamaDoc

The general steps for using ApamaDoc are as follows:

1. Create an Apama project in Software AG Designer.
2. Add a `.mon` file to your project.

3. In the `.mon` file, enhance the automatically generated documentation by adding annotations. See [“Inserting ApamaDoc comments” on page 431](#), [“Inserting ApamaDoc tags” on page 432](#), and [“Inserting ApamaDoc references” on page 435](#).
4. Save and build the project.
5. Right-click the project name and select **Export** from the context menu.
6. In the **Export** dialog, expand **Software AG**, select **ApamaDoc Export**, and click **Next**.
7. Identify the folder that you want to contain the ApamaDoc output, and click **Finish**.

To view the ApamaDoc output, go to the output folder you identified and double-click the `index.html` file. The generated ApamaDoc opens in your browser.

Try this with any project you already have, or with one of the demo projects. Even if you have not added any ApamaDoc annotations, you can see that ApamaDoc automatically generates a lot of documentation.

Inserting ApamaDoc comments

To augment the documentation automatically generated by ApamaDoc, insert comments in your EPL files in the following format:

1. Start the comment with the `/**` characters, rather than the usual `/*` notation.
2. Enter the text you want to appear in the generated documentation.
3. After each newline, to continue the ApamaDoc comment, insert a `*` character at the beginning of the next line.
4. As needed, insert one or more tags for particular constructs. See [“Inserting ApamaDoc tags” on page 432](#). Any tags must occur at the beginning of a newline (ignoring `*` and whitespace characters). Documentation for a tag ends when you declare another tag or end the comment.
5. End the comment with the usual `*/` characters.

For example, your EPL code might look like this:

```
/**
 * Called by the monitor when it executes the onload() action.
 * This action maintains the configuration for this scenario.
 * @param sId The scenario ID.
 * @param updateCallback The callback after the configuration is updated.
 */
action init(string sId, action<> updateCallback) {
    scenarioId:=sId;
    route GetConfiguration(scenarioId);
    listener l:=on Configuration(scenarioId=scenarioId) as c {
        config := c.configuration;
```

```
        defaultConfig := c.defaults;
        configurationUpdated();
        updateCallback();
    }
    listeners.append(l);
}
```

When ApamaDoc processes these comments, it removes initial and trailing whitespace and * characters. For example, the ApamaDoc output would look like this:

```
init
void init(string sId, action< > updateCallback)
Called by the monitor during execution of the onload() action. This action
maintains the configuration for this scenario.
Parameters:
    sId - The scenario ID.
    updateCallback - The callback after the configuration is updated.
Listens:
com.apama.scenario.Configuration
```

Inserting ApamaDoc tags

ApamaDoc automatically generates documentation for EPL code constructs. To enhance the quality of the documentation, you can insert tags that let you provide and link to additional information. A tag begins with an @ symbol and is immediately followed by a name and other information. The following table describes the tags you can insert.

Tag	Description	Use For
@author <i>name</i>	There are no restrictions on the name. It can span multiple lines. It is ended by the start of the next tag.	Events, monitors, and custom aggregate functions
@deprecated [<i>description</i>]	The optional description can be anything pertinent to the deprecated construct. For example, you might want to suggest a newer equivalent or provide a reason for the deprecation.	All code constructs
@emits <i>eventRef</i> [<i>description</i>]	Documents events that are emitted. <i>eventRef</i> specifies a link to an event definition. The optional description can be anything pertinent to the emitting of the event. See “Inserting ApamaDoc references” on page 435 .	Actions and their enclosing types
@enqueues <i>eventRef</i> [<i>description</i>]	Documents events that are enqueued. <i>eventRef</i> specifies a link to an event definition. The optional description can be anything pertinent to the enqueueing of the event. See	Actions and their enclosing types

Tag	Description	Use For
	“Inserting ApamaDoc references” on page 435.	
<code>@listens eventRef [description]</code>	Documents events that are being listened for. <i>eventRef</i> specifies a link to an event definition. The optional description can be anything pertinent to the event listener. See “Inserting ApamaDoc references” on page 435.	Actions and their enclosing types
<code>@param codeRef [description]</code>	Documents arguments to actions and custom aggregate functions. <i>codeRef</i> specifies the parameter name. The description should be a sentence describing the purpose of the parameter and any constraints on the permitted values that may be specified.	Actions and custom aggregate functions
<code>@private</code>	Hides constructs from ApamaDoc. On a line by itself, immediately precede the construct that you do not want to generate documentation for with the following: <code>/** @private */</code>	All code constructs
<code>@returns description</code>	Documents the return value of an action or custom aggregate function. The description should be a sentence describing the purpose of the return value, and any pertinent information about the possible values that may be returned. Note that for backwards compatibility reasons <code>@return</code> is maintained as an alias for <code>@returns</code> .	Actions and custom aggregate functions
<code>@routes eventRef [description]</code>	Documents events that are being routed. <i>eventRef</i> specifies a link to an event definition. The optional description can be anything pertinent to the routing of the event. See “Inserting ApamaDoc references” on page 435.	Actions and their enclosing types
<code>@see</code>	There are three forms of this tag. Each form documents a relationship	All code constructs

Tag	Description	Use For
	between a code fragment and some other information. <code>@see "description"</code> Lets you insert text that explains the relationship. <code>@see codeRef description</code> Lets you reference an EPL code construct and describe the relationship between this construct and that construct. <code>codeRef</code> specifies a link to some other EPL code. See “Inserting ApamaDoc references” on page 435 . <code>@see linkText [description]</code> Lets you specify an HTML link to an external resource. Optionally, you can add more information.	
<code>@sends eventRef [description]</code>	Documents events that are sent to a channel. <code>eventRef</code> specifies a link to an event definition. The optional description can be anything pertinent to the sending of the event to a channel. See “Inserting ApamaDoc references” on page 435 .	Actions and their enclosing types
<code>@since version</code>	Documents when a code construct was introduced. Replace <code>version</code> with a particular version number, for example, 9.9.	All code constructs
<code>@spawns actionRef [description]</code>	Lets you document the lifecycle of a monitor. <code>actionRef</code> specifies a link to an action definition. See “Inserting ApamaDoc references” on page 435 .	Actions and their enclosing types
<code>@throws [description]</code>	Documents exceptions which may be thrown from an action and the conditions which would cause the action to throw an exception.	Actions
<code>@version version</code>	Lets you specify a version of the current incarnation of this code.	Events, monitors, custom aggregate

Tag	Description	Use For
	Replace <i>version</i> with a particular version number, for example, 9.9.	functions, and import statements

Inserting ApamaDoc references

Many ApamaDoc tags contain links to other parts of the EPL code. These tags specify one of the following link types:

- Code references
- Type references
- Event references
- Action references

A *code reference* is a link to a monitor definition, an event type definition, an action definition, a member (variable or named constant) declaration or an import declaration. A code reference has two forms.

The first form links to constructs that are in the monitor definition or event type definition that contains this ApamaDoc comment. The target of the link can be a variable declaration, named constant declaration, import declaration, or action definition. The format for this code reference is as follows:

```
[ # ] (member | import | ( action() ) )
```

The hash symbol is optional. You must specify one of the following:

- Name of a member (variable or named constant) that is in the monitor or event type definition that contains this ApamaDoc comment.
- Name of an item that is being imported in the monitor or event type definition that contains this ApamaDoc comment.
- Name of an action that is in the monitor that contains this ApamaDoc comment. If you specify an action, the name of the action must end with parentheses. For example:

```
#updateOrder()
```

The second form links to constructs that are not in the monitor or event type definition that contains this ApamaDoc comment. You can link to code constructs that are in the same package or in other packages. The format for this code reference is as follows:

```
[ package [. monitor ].]type[ #(member | import | ( action() ) )]
```

Replace *type* with the name of a monitor or event type definition. If the ApamaDoc comment is in the same package as the link target, the package specification is optional. If you replaced *type* with the name of an event that is defined in a monitor, you must replace *monitor* with the name of that monitor and you must specify the package name.

The hash symbol followed by a name is required when the link target is a variable declaration, named constant declaration, import declaration, or action definition. If you specify an action, the name of the action must end with parentheses.

If the code reference is valid the rendered HTML output contains a hyperlink to the referenced code construct's documentation followed by the descriptions text, if any. If the reference is not valid, the output displays only the tag's description text if you provided it.

A *type reference* is a subset of a code reference. It always links to a monitor or event type definition.

An *event reference* is a subset of a type reference. It always links to an event type definition.

An *action reference* is a subset of code reference. It always links to an action. The action can be in an event type definition, in a monitor, or in an event type definition that is in a monitor.

Inserting EPL source code examples

ApamaDoc supports the `<code>...</code>` tag in ApamaDoc comments. You can use this tag to specify code snippets as part of the comment. In this case, you need not specify the comment character (*) at the beginning of each line.

The ApamaDoc `<code>` tag is a block element, and is not the same as the HTML `<code>` tag that is used for inline markup. Unlike the HTML `<code>` tag, the ApamaDoc `<code>` tag preserves the indentation and line breaks of the code snippet. You need not use the HTML `
` tag to force a line break.

If you want to mark up pieces of program code within the running text, use the HTML tag `<tt>`.

In many cases, you can enter the less-than (<) and greater-than (>) signs as they are. However, you need to take care when you have text that is enclosed in these signs in your comment and which has no spaces between the text and these signs (for example, `sequence<string>`). This will be interpreted as an HTML tag. To avoid this, we recommend that you always add spaces (for example, `sequence< string >`). If this is not possible for your particular use case, you have to use the `<` and `>` entities instead (for example, `sequence<string>`). So check the generated result carefully if you are using these signs.

Example:

```
/**
 * This is an example of a comment.
 *
 * When using the <tt>code</tt> tag, all indentation and line breaks
 * are preserved.
<code>
sequence< string > s :=
    ["Something", "Completely", "Different"];
print ", ".join(s);
</code>
 * Note that the above < and > signs will be generated unchanged
 * because spaces have been added.
 */
```

Note:

If you have to use the special symbol @ within the ApamaDoc `<code>...</code>` tag, you must use the HTML ASCII code `@` instead of the symbol. Otherwise, this will be interpreted as an ApamaDoc tag (such as `@param`).

Generating ApamaDoc from the command line

The `apamadoc` tool generates HTML documentation in ApamaDoc format from EPL files (`.mon` files).

The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt or using the `apama_env` wrapper (see "Setting up the environment using the Apama Command Prompt" in *Deploying and Managing Apama Applications*) ensures that the environment variables are set correctly.

Synopsis

To generate HTML documentation from EPL files using the `apamadoc` tool, run the following command:

```
apamadoc [options] output_folder input...
```

When you run this command with the `--help` option, the usage message for this command is shown.

Description

The `apamadoc` tool generates the HTML documentation in the specified output folder, using the input method described below.

You can view the ApamaDoc by opening `output_folder/index.html` in a web browser.

Options

The `apamadoc` tool takes the following options:

Option	Description
<code>-t title</code>	Uses the specified title in the generated documentation.
<code>-p</code>	Generates documentation for elements marked as private.
<code>-v</code>	Generates verbose output on standard output.
<code>--includeMonitorMembers</code>	Generates documentation for monitor members. By default, ApamaDoc is only generated for the API exposed by the specified EPL files. The members of monitors are therefore not generated (for example,

Option	Description
	inner events, variables and constants, fields in events, and actions).
<code>--help -help -h</code>	Displays usage information.

Operands

The `apamadoc` tool takes the following operands:

Operand	Description
<code>output_folder</code>	The name of the folder in which the documentation is to be generated.
<code>input</code>	<p>The input method. Specify one or more of the following (you can specify any combination of these):</p> <ul style="list-style-type: none">■ A file containing a list of EPL files, separated by newlines. The filename must be preceded by an ampersand (<code>@</code>). For example, <code>@epl_list.txt</code>.■ A folder that is recursively searched for the EPL files.■ An EPL file. <p>If the same EPL file occurs more than once in the input, this file is only processed once.</p> <p>If an input file or folder is not found, the <code>apamadoc</code> tool reports an error. Documentation is not generated in this case.</p>

Examples

The following examples show the different ways in which the `apamadoc` tool can be started:

- Generate documentation in the `doc` folder, using the EPL files listed in the `epl_list.txt` file:

```
apamadoc doc @epl_list.txt
```
- Generate documentation as above, but with “Example” as the title in the generated documentation:

```
apamadoc -t "Example" doc @epl_list.txt
```
- Generate documentation in the `doc` folder, recursively processing the EPL files in the `src/epl` folder with verbose output enabled:

```
apamadoc -v doc src/epl
```

- Generate documentation in the `doc` folder from the `src/epl/example.mon` file only:

```
apamadoc doc src/epl/example.mon
```

- Generate documentation in the `doc` folder, using the following: the EPL files listed in the `ep1_list.txt` file, the EPL files recursively found in the `src/epl` folder, and the `myep1.mon` file from the parent folder:

```
apamadoc doc @ep1_list.txt src/epl ../myep1.mon
```

Generating ApamaDoc from an Ant script

Generating ApamaDoc from an Ant script is useful if you want to control what ApamaDoc generates without user-interface intervention, for example, when you are running nightly build integrations. Also, using the Ant script, you can control which files are exported and which files are omitted. See the `generate-apamadoc` macro definition in the `apama-macros.xml` Ant script for more details. You can find this Ant script in the `etc` directory of your Apama installation.

To generate ApamaDoc from an Ant script, you have to create a `build.xml` Ant script that uses the `generate-apamadoc` macro to build ApamaDoc for the EPL files you wish to document. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="" default="apamadoc" basedir=".">
  <property environment="env"/>
  <import file="${env.APAMA_HOME}/etc/apama-macros.xml"/>
  <target name="apamadoc">
    <generate-apamadoc outputDir="./apamadoc-output" title="ApamaDoc Title">
      <!-- standard Ant fileset to specify which files should be documented -->
      <fileset dir="./monitors">
        <include name="**/*.mon"/>
      </fileset>
    </generate-apamadoc>
  </target>
</project>
```

By default, ApamaDoc is only generated for the API exposed by the specified EPL files. The members of monitors are therefore not generated (for example, inner events, variables and constants, fields in events, and actions). This can be enabled, however, using the `includeMonitorMembers` attribute.

Note:

The script that you use to generate ApamaDoc requires Apache Ant. To set the path appropriately, it is recommended that you run your script from the Apama Command Prompt (see "Setting up the environment using the Apama Command Prompt" in *Deploying and Managing Apama Applications*). If you do not use the Apama Command Prompt, then you must ensure that the `PATH` variable contains an entry for the Ant installation folder (such as `C:\ant`), which makes the Apama `ant.bat` file accessible to ApamaDoc generation.

II Developing Apama Applications in Java

14	Overview of Apama JMon Applications	443
15	Defining Event Expressions	463
16	Concept of Time in the Correlator	487
17	Developing and Deploying JMon Applications	493

14 Overview of Apama JMon Applications

■	Introducing JMon API concepts	445
■	About event types	446
■	About monitors	452
■	About event listeners and match listeners	453
■	Description of the flow of execution in JMon applications	455
■	Parallel processing in JMon applications	456
■	Identifying external events	459
■	Optimizing event types	459
■	Logging in JMon applications	461
■	Using EPL keywords as identifiers in JMon applications	462

This part provides information and instructions for using Apama's in-process API for Java, called JMon, to write applications that run on the correlator. To develop an Apama application you can use the correlator's native Event Processing Language (EPL) or JMon. This part focuses exclusively on how to use JMon to write an application that runs on the correlator.

JMon reference documentation is provided in Javadoc format.

Important:

Apama provides both EPL and JMon as ways to create powerful Complex Event Processing (CEP) applications. Both types of application can use Apama's event expressions to set up listeners that detect patterns of events. With JMon, however, there are restrictions on the Apama capabilities that can be used. Therefore, for new applications Software AG strongly recommends choosing the more powerful EPL language over JMon. A common pattern for writing the main logic and event expressions for a CEP application is to use EPL (not JMon), with the EPL code calling Java plug-ins (see [“Writing EPL Plug-ins in Java” on page 527](#)) as needed for specific operations such as invoking a third-party library. Keep in mind that a Java plug-in is not the same as a JMon application, and JMon - though not deprecated - should be considered a legacy feature. Any future enhancements in this area will be made to the Java plug-in mechanism and not to JMon. Any customer application that needs to tightly integrate CEP logic with Java libraries should use the more modern pattern of EPL calling into Java plug-ins.

JMon restrictions include:

- Streams cannot be defined.
- Multiple contexts are not supported.
- Channels are not supported.
- Decimal data types are not supported.
- Other EPL plug-ins cannot be called.

The correlator is Apama's core event processing and correlation engine. Interfaces to the correlator let you inject monitors that

- Analyze incoming event streams to find patterns of interest
- Specify the actions to undertake when the correlator identifies such patterns

You can use the Apama JMon API to write applications that are to be deployed on the correlator.

The correlator embeds a Java Virtual Machine in which Apama JMon applications can be loaded and run.

The JMon API provides a suite of Java classes that allow a developer to build a Java application, and then inject it into the correlator. Apama JMon applications can define *listeners*, which specify patterns and sequences of events to look for and actions to carry out when the correlator detects those events.

You can develop Apama JMon applications in Software AG Designer. When you use Software AG Designer to develop an application, it can automatically generate a framework for your JMon event and JMon monitor files.

For more information on developing JMon applications in Software AG Designer, see, "Adding a new JMon application", "Adding a JMon monitor", and "Adding a JMon event" in *Using Apama with Software AG Designer*.

Note:

Apama includes the in-process API for Java (JMon) and the client API for Java. In most cases, the context makes it clear which API the discussion is addressing. When this is not clear, the APIs are referred to as the JMon API or Apama client API for Java.

Introducing JMon API concepts

This section introduces the main concepts behind programming the functionality within Apama using JMon. It describes how events are modeled in JMon and how they are used to drive and trigger *listeners* within JMon *monitor* classes.

Apama is designed to fit within an event (or message) driven world. In event driven systems information is propagated through units of information termed events or messages. Conceptually, an event typically represents an occurrence of a particular item of interest at a specific time, and is usually encoded as an asynchronous network message.

Apama is designed to process thousands of these event messages per second, and to sift through them for sequences of interest based on their contents as well as their temporal relationships to each other. When writing Apama applications using JMon, the Java code you write informs the correlator of the sequences of interest and, when matching event sequences are detected, these are passed to your JMon code for handling. Apama's correlator component is capable of looking for hundreds of thousands of different event sequences concurrently.

In order to program the correlator using JMon, a developer must write their application as a set of Java classes that implement the JMon APIs. This programming model is similar to writing Enterprise JavaBeans intended for use in an application server. These Java classes then need to be loaded (or “injected”) into the correlator, which instantiates and executes them immediately.

Almost all of the standard language functionality provided by Java and its libraries can be used in JMon applications, just as in any other Java applications. However, the power of the correlator is only truly leveraged by invoking its event matching, correlation and event generation capabilities. As streams of events are passed into a correlator, the *listeners* defined in JMon applications sift through the events looking for specific sequences of interest matching a variety of temporal constraints. Once a listener triggers, a method is invoked on a Java object, the *Match Listener* object. The developer specifies this object when the listener is created.

Three kinds of Java class objects can be loaded into the correlator; event types, monitors and match listeners.

- Event type classes serve to define the event types that the correlator can accept from external sources and carry out correlations on.
- Monitor classes program the correlator. They define what event patterns the correlator must look for and allow arbitrary Java code to be executed.
- Match listeners provide a method that is called when a specific event sequence is detected.

These three Java class types will be now be discussed in detail.

About event types

Apama events are strongly typed. Each event must be of a specific known type, henceforth called the *event type*. An event type defines the name of the event, and its particular set of parameters. Every parameter is named and can be one of a selection of types. Every event instance of a given event type is therefore identical in structure; every instance has the same set (and order) of parameters.

Before the correlator can understand and process events of a specific event type, it needs to have been provided with an *event type definition*. This allows it to understand the event messages it is passed, create optimal indexing structures, and allows listeners to be set up to look for event sequences involving events of that type.

An event type definition defines the event type's name and the name, type and order of each of its parameters. Parameters can be of any of the following types:

- Java standard types `String`, `long`, `double`, `boolean` or `Map`.
- Java arrays.
- `com.apama.jmon.Location` type — This type corresponds to either a spatial point represented by two coordinates, or a rectangular space expressed in terms of its two bounding corners.

Apama's JMon API supports Java generic maps. Apama recommends that you use these when possible instead of the `Event.getMapFieldTypes()` method. Doing so lets you gain the benefits of compile-time type safety as well as a simpler class definition.

However, while it is valid to declare a parameter to be an array of generic maps, assignment of values to the map elements is not type-safe, and will be rejected by the Java compiler. If you need a parameter that is an array of maps, use the `Event.getMapFieldTypes()` method instead of generic maps.

You can nest a plain `Map` as a value (not a key) at any depth in a parameterized `Map`. You cannot nest a parameterized `Map` in a plain `Map`. This is because you would not be able to specify the parameterized types to be returned from the `getMapFieldTypes()` method. Of course, you can nest a parameterized `Map` as a value (but not a key) in a parameterized `Map`. For example:

EPL:

```
event BadComplexEventExample {
    dictionary <string , dictionary <string, SimpleEvent>> complex;
}
```

Java:

```
import java.util.Map;
import java.util.HashMap;
import com.apama.jmon.Event;
public class BadComplexEventExample extends Event {
    // By using a non-parameterized map you lose the information that the
    // field is a dictionary with values that are also dictionaries.
    public Map complex;
```

```

public BadComplexEventExample() {
    this(new HashMap());
}

public BadComplexEventExample(Map complex) {
    this.complex = complex;
}
}

```

See also the definition of `ComplexEvent` in [“About event parameters that are complex types” on page 449](#).

An event can embed an event (potentially of a different type) as a parameter.

Simple example of an event type

An event type is defined as a Java class as per the following example,

```

/*
 * Tick.java
 *
 * Class to abstract an Apama stock tick event. A stock tick event
 * describes the trading of a stock, as described by the symbol
 * of the stock being traded, and the price at which the stock was
 * traded
 */

import com.apama.jmon.Event;

public class Tick extends Event {
    /** The stock tick symbol */
    public String name;

    /** The traded price of the stock tick */
    public double price;

    /**
     * No argument constructor
     */
    public Tick() {
        this("", 0);
    }

    /**
     * Construct a tick object and set the name and price
     * instance variables
     *
     * @param name The stock symbol of the traded stock
     * @param price The price at which the stock was traded
     */
    public Tick(String name, double price){
        this.name = name;
        this.price = price;
    }
}

```

By Java programming conventions, the previous definition would need to be provided on its own in a stand-alone file, for example, `Tick.java`.

The definition must import the definition of the `Event` class. This is provided as part of the `com.apama.jmon` package provided with your Apama distribution. See [“Developing and Deploying JMon Applications” on page 493](#) on installation and deployment for details of where to locate this package.

`Event` is the abstract superclass of all user classes implementing desired event types. Then we must define our new event class as a subclass of the `Event` type.

The user-defined event class must have three primary elements:

- A set of `public` variables that define the event's parameters
- A “no argument” constructor, whose purpose is to construct an instance of the event with the parameters set to default values
- A constructor whose parameter list corresponds (in type and order) to the event's parameters. This constructor allows creation of an instance of the event with specific parameter values.

In the above example the event is called `Tick`, and it has two parameters, `name`, of type `String`, and `price`, of type `double`. The previous definition may be considered a simple template for how to write all event definitions.

Note:

Non-public (like `private` and `protected`) variables are not considered to be part of the event schema.

Extended example of a JMon event type

Let us now consider an extended example:

```
package test.jmon.example;

import java.util.Map;
import com.apama.jmon.*;

/*
 * TestEvent.java
 *
 * Class to abstract an Apama event whose primary purpose is to
 * showcase how to define an event class containing parameters of
 * all the allowed types, including arrays and Maps.
 */

public class TestEvent extends Event {

    // example of parameters of the basic types
    public long primitiveInteger;
    public double primitiveFloat;
    public boolean primitiveBoolean;
    public String referenceString;

    // example of parameters consisting of arrays of the basic types
    public long[] sequenceInteger;
    public double[] sequenceFloat;
    public boolean[] sequenceBoolean;
```



```

public String[] sequenceString;

// a nested event of type EmbeddedTestEvent
public EmbeddedTestEvent referenceNestedTestEvent;

// a parameter of type Location
public Location referenceLocation;

// a parameter of type Map
public Map<long, String> dictionaryIntegerString;
. . .
}

```

Comparing JMon and EPL event type parameters

You might already be familiar with EPL, the Apama complex event processing scripting language through which the correlator can be programmed as an alternative to JMon. Event types defined in JMon can be used in EPL, and vice-versa. JMon event type parameters map to EPL parameter types as follows:

JMon Type	Equivalent EPL Type
long	integer
double	float
boolean	boolean
String	string
Location	location
array	sequence (of the same type)
Map	dictionary (with the same key and value types)
com.apama.jmon.Event or its subclass	event (with the same equivalent subset of fields as defined in this table)

The correlator's performance can be optimized by *wildcarding* event type definitions where appropriate. This procedure is described in [“Optimizing event types” on page 459](#).

About event parameters that are complex types

It is possible in both EPL and JMon to declare a field of an event definition to be a complex type. For example, the SequenceEvent definition below defines an event that is constructed from a sequence of DataHolder events, which in turn contain a string and an integer. This is defined in EPL in two events thus:

```

event DataHolder {
    string name;
    integer age;
}

```

```
}  
  
event SequenceEvent {  
    sequence <DataHolder> complex;  
}
```

An example constructed SequenceEvent event is show below:

```
SequenceEvent([DataHolder("kap", 1), DataHolder("gbs", 2)])
```

The equivalent event definitions for the above in Java are defined below:

```
import com.apama.jmon.Event;  
  
public class DataHolder extends Event {  
    /** Event fields */  
    public String name;  
    public long age;  
  
    /** No argument constructor  
    */  
    public DataHolder () {  
        this("", 0);  
    }  
    /** Construct a DataHolder object and set the instance variables  
    */  
    public DataHolder (String n, long a) {  
        name = n;  
        age = a;  
    }  
}  
  
import com.apama.jmon.Event;  
  
public class SequenceEvent extends Event {  
  
    /** Event field */  
    public DataHolder[] people;  
  
    /** No argument constructor  
    */  
    public SequenceEvent() {  
        this( new DataHolder[]{} );  
    }  
  
    /** Construct a SequenceEvent object and set the instance variable  
    */  
    public SequenceEvent(DataHolder[] p) {  
        this.people = p;  
    }  
}
```

Sample Java code to create and emit a SequenceEvent event is shown below:

```
s = new SequenceEvent(new DataHolder[] {new DataHolder("kap", 1),  
    new DataHolder("gbs", 2)} );  
s.emit();
```

Events can also include Map types, which are equivalent to EPL dictionary types. When you use Map types, Apama recommends that you use generic maps whenever you can. For example, in EPL the following event is a dictionary of dictionaries and each internal dictionary is a sequence of SimpleEvent types:

```
event ComplexEvent {
    dictionary <string,
        dictionary <string, sequence<SimpleEvent>>> complex;
}
```

You can implement this in Java as follows:

```
import java.util.Map;
import java.util.HashMap;
import com.apama.jmon.Event;
import com.apama.jmon.annotation.EventType;

@EventType(description = "Event that contains a field with a complex structure")
public class ComplexEvent extends Event {

    /** Event field */
    public Map<String, Map<String, SimpleEvent[]>> complex;
    /**
     * No argument constructor
     */
    public ComplexEvent() {
        this(new HashMap<String, Map<String, SimpleEvent[]>>());
    }

    /**
     * Construct a ComplexEvent object, set the instance variable complex
     *
     * @param complex The dictionary/Map to use as the field value
     */
    public ComplexEvent(
        Map<String, Map<String, SimpleEvent[]>> complex) {
        this.complex = complex;
    }
}
```

This example is provided in its complete form as a sample. It is distributed in the folder `samples/java_monitor/complex_event/`.

Non-null values for non-primitive event field types

When the correlator creates an event to pass to the JMon code, it ensures that all fields of a non-primitive type have a non-null value. Note that this is different from the Java default, which is to allow null values for non-primitive types.

The `com.apama.jmon.Event` default constructor uses reflection to initialize non-primitive null fields with the following values:

- `sequence` — an empty array of the specified type.
- `dictionary` — an empty `java.util.HashMap` object.

- `string` — an empty `java.lang.String` object.
- `event` — a default construction of the event, with recursive initialization for any of its non-primitive fields that have null values.

In your application, if you explicitly assign a null value to a non-primitive event field, and your application tries to emit, enqueue, or route that event, the correlator logs an error and terminates your application.

About monitors

Monitor classes configure the activity of the correlator. This is analogous to how an Enterprise JavaBean effectively defines the activity of an application server.

All monitor classes must implement the `com.apama.jmon.Monitor` interface and define an `onLoad` method. When a monitor class is loaded into the correlator, it is instantiated as an object and its `onLoad` method is executed. In Java parlance, this would be equivalent to the `static void main (args[])` method.

Most Java code (with certain limitations) can be executed within the `onLoad` method, although its primary purpose is probably to configure one or more asynchronous *listeners* for specific events or event sequences.

A monitor class must define a “no argument” constructor. The Java code within the correlator uses this when the class definition is loaded.

Below is a minimal monitor:

```
import com.apama.jmon.*;

public class Simple implements Monitor {
    /**
     * No argument constructor used by the jmon framework on
     * application loading
     */
    public Simple() {}

    /**
     * Implementation of the Monitor interface onLoad method.
     * Does nothing.
     */
    public void onLoad() {
    }
}
```

The above monitor class does nothing and is shown here as a template for how to define a monitor class.

EPL. Although there are similarities, the concept of a monitor in EPL and in JMon is not the same. The EPL monitor is a very powerful custom programming structure, whereas in JMon a monitor class is primarily a standard Java class with an entry method that gets automatically executed upon loading (as described in the topics below).

About event listeners and match listeners

For a monitor class to leverage the intrinsic features of the correlator, it must set up one or more *listeners*.

A listener is a conceptual entity whose function is to sift through all incoming event streams looking for a particular event or sequence of events. The event or sequence of events of interest is represented as an *event expression*.

The simplest way of setting up a listener is by creating an instance of an `EventExpression` and then specifying a `MatchListener` object that gets triggered when the expression becomes true, that is, when a suitable event or event sequence is detected. A more efficient alternative is to use a *prepared event expression*, which is described in [“Optimizing event types” on page 459](#).

A match listener is a Java object that implements the `com.apama.jmon.MatchListener` interface and implements the `match` method. This method is called by the correlator when the event expression it is registered with is detected.

Example of a MatchListener

The following example illustrates this functionality:

```
import com.apama.jmon.*;
public class Simple implements Monitor, MatchListener {

    /**
     * No argument constructor used by the jmon framework on
     * application loading
     */
    public Simple() {}

    /**
     * Implementation of the Monitor interface onLoad method. Sets up
     * a single event expression looking for all Tick events
     * with a trade price of greater than 10.0. This class instance
     * is added as a match listener to the event expression.
     */
    public void onLoad() {
        EventExpression eventExpr = new EventExpression("Tick(*, >10.0)");
        eventExpr.addMatchListener(this);
    }

    /**
     * Implementation of the MatchListener interface match method.
     * Prints out
     * a message when the listener triggers
     */
    public void match(MatchEvent event) {
        System.out.println("Pattern detected");
    }
}
```

This example illustrates several new concepts.

Consider the `onLoad` method. Firstly it creates an event expression object variable. This object, of type `com.apama.jmon.EventExpression`, represents an event, or sequence of events, to look for. The constructor of an `EventExpression` is passed a string that defines the actual event expression.

As the syntax of an event expression will be illustrated in the next section it is enough to say that this event expression is specifying “the “first” “Tick” event whose “price” parameter is greater than the value “10.0.”

Then, a match listener is registered with the newly created event expression object. A match listener can be any object that implements the `com.apama.jmon.MatchListener` interface and defines the `match(MatchEvent event)` method. For the sake of simplicity, the `Simple` monitor class has here been written to also implement the `MatchListener` interface, and therefore the statement,

```
eventExpr.addMatchListener(this);
```

is passing `this` as the reference to a suitable `MatchListener`.

Once a match listener has been registered with an event expression the correlator creates a listener entity to start looking for the specified event expression.

Listeners are asynchronous. Hence the `match` method may be invoked at any time subsequent to the activation of the listener, but always after all Java code in the current method finishes executing. Therefore in this case all Java statements in the `onLoad` method would finish being executed before `match` is called after a match.

Defining multiple listeners

A monitor can define any number of event expressions, and create any number of listeners. The following code,

```
public void onLoad() {
    EventExpression eventExpr1 = new EventExpression("Tick(*, >10.0)");
    EventExpression eventExpr2 =
        new EventExpression("NewsItem(\"ACME\", *)");

    eventExpr1.addMatchListener(this);
    eventExpr2.addMatchListener(this);
}
```

is creating two event expressions, `eventExpr1` and `eventExpr2`. Then each is assigned a match listener, thus activating two distinct listeners. The fact that both are being assigned the same match listener object, i.e. `this` same object `this`, is inconsequential. It just means that the same method, the `match` method of this object, will be called when the correlator detects either of the event expressions.

As already described, creating a listener is an asynchronous operation that returns immediately. In the above code, in practice both listeners are created concurrently. It is not possible for the `eventExpr1` listener to trigger before the `eventExpr2` listener is created. However, once the enclosing method's code has completed execution, the listeners can trigger at any time, and independently of each other.

Removing listeners

A `MatchListener` instance that is no longer connected to an event expression, and to which there are no references, is garbage collected in the usual way. In some situations, you might want to be notified when the correlator removes its reference to the `MatchListener` (when it can no longer fire). For example, you might need this notification if the `MatchListener` has unmanaged resources (for example, open files) that need to be explicitly cleaned up when it is no longer needed, or your application has other references to the `MatchListener` that need to be removed when the listener can no longer fire so that it can be garbage collected. In those situations, you can define your listener so that it implements the `com.apama.jmon.RemoveListener` interface. There is no requirement to implement this interface. It is up to you to determine whether you need it.

The `RemoveListener` interface extends the `MatchListener` interface by providing one additional method: `removed()`. If you implement the `RemoveListener` interface, the correlator calls your implementation of the `removed()` method in the following situations:

- The application removes your listener from the event expression it is attached to.
- The event expression your listener is attached to is in a state that will never match. For example, on `A()` within `(10.0)` after 10 seconds have elapsed without an `A()`.

In the following example, the `removed()` method is called because the event expression dies after 10 seconds.

```
import java.util.HashMap;
import com.apama.jmon.*;

public class Test implements Monitor {
    public Test() {}
    public void onLoad() {
        EventExpression e = new EventExpression("TestEvent() within(10.0)");

        e.addMatchListener(new RemoveListener() {
            public void match(MatchEvent event) {
                System.out.println(Correlator.getCurrentTime() +
                    ": Received match");
            }
        });
        public void removed(EventExpression e) {
            System.out.println(Correlator.getCurrentTime() +
                ": Received removed");
        }
    }
}
```

Description of the flow of execution in JMon applications

The flow of execution of JMon applications through the correlator at any given time is single threaded. All the listeners of JMon applications are fired in a single-threaded manner. However, during the lifetime of a JMon application, its execution may be moved among a number of threads

by the correlator. This is particularly important since thread-local variables will not behave in the same way as you would expect them to in a conventional Java application.

When a number of monitor classes are loaded into the JVM within the correlator their `onLoad` methods are executed in turn, in the same order as the injected classes, and any listeners created are set up and activated.

Control then reverts to the correlator, which takes in one event from its input queue. This event is examined by each of the active listeners in turn (the order is undefined), and each one that triggers immediately calls the `match()` method in its registered `MatchListener` object.

Once all the listeners have processed the event (and hence all `match` methods terminated), control reverts to the correlator to process the next input event. Note that since events can also match listeners in EPL monitors, these would also be processed before control reverts.

However, JMon applications can create other Java threads. In such multi-threaded JMon applications, the correlator has no control of these additional Java threads. Consequently, you should never route or emit an event from a Java thread that was not the thread in which the correlator invoked the JMon application. Doing this results in unpredictable behavior. To communicate from your JMon application to other parts of the correlator, use the `enqueue()` method or preferably, the `enqueueTo()` method.

Parallel processing in JMon applications

By default, the correlator operates in a serial manner. If you want, you can implement contexts for parallel processing. You can create contexts only with EPL but you can then use those contexts from your Apama JMon code. This section provides information about how to use contexts in Apama JMon applications.

For general information about contexts and instructions for creating contexts, see [“Implementing Parallel Processing” on page 285](#).

You can find a sample JMon application that implements the use of contexts in the `samples\java-monitor\context` directory of your Apama installation directory.

Overview of contexts in JMon applications

The Apama JMon API provides the `com.apama.jmon.Context` type. This class corresponds to the EPL context type, but with a more limited set of features:

- A JMon event definition can contain a `Context` type field. This lets you transfer a reference to a context to and from an Apama JMon application. You cannot pass context references between the correlator and your Apama JMon application on their own.
- You can enqueue events to
 - Particular contexts: `Event.enqueueTo(Context c)`
 - A list or array of contexts:

```
Event.enqueueTo(java.util.List<Context> ctxList)
```



```
Event.enqueueTo(Context[] ctxList)
```

See [“Emitting, routing, and enqueueing events” on page 469](#).

- You can call `Context.getCurrent()` to obtain a reference to the context that a piece of code is running in. See [“Obtaining context references” on page 290](#).
- The `Context` class provides accessor methods for context properties such as context name and context ID.

Using contexts in JMon applications

➤ To use EPL contexts in JMon applications

1. In EPL code, create a context that you want to use in your JMon application.
2. In your JMon application, define an event type that contains a `Context` field.
3. Use this event type to obtain a reference to the context you created in EPL.
4. Use the context reference to enqueue events to that context.

For an example, see the `samples\java-monitor\context` directory in your Apama installation directory.

Using the `Context` class default constructor

The `com.apama.jmon.Context` class default constructor, `public Context()`, creates a dummy context that provides the same functionality as an uninitialized context variable in EPL. A JMon dummy context does not correspond to an actual correlator context. The JMon dummy context corresponds to the implicit context that is created in EPL for uninitialized context variables. The default constructor is provided for convenience. Use it when you want to enqueue an event to another context from a JMon application and the event happens to have a context field that contains an irrelevant value. As with other JMon types, this value cannot be null. Following is an example, beginning with the event definition:

```
import com.apama.jmon.*;

public class ContextEvent extends Event {
    public long id;
    public boolean req;
    public Context c;
    public ContextEvent(long id) {
        this.id = id;
        this.req = true;
        this.c = new Context();
    }
}
```

Here is the JMon application:

```
public class SampleJMonApp implements Monitor {
    ...
    public void onLoad() {
        ...
        ContextEvent req = new ContextEvent(service_id);
        ...
        // send requests here
        req.route();
        ...
        EventExpression cexpr =
            new EventExpression("all ContextEvent(*,false,*):ackEvt");
        cexpr.addMatchListener(new MatchListener() {
            public void match(MatchEvent event){
                ContextEvent ackEvt =
                    (ContextEvent)event.getMatchingEvents().get("ackEvt");
                // extract the context here
                Context serviceContext = ackEvt.evt;
                ...
            }
        });
    }
    ...
}
```

Here is the EPL application:

```
monitor ContextFactory
{
    ...
    action onload() {
        ...
        on all ContextEvent(*, true, *) as req {
            integer svcid;
            ...
            context serviceContext := context("svc");
            ContextEvent ack :=
                ContextEvent(svcid, false, serviceContext);
            route ack;
            ...
        }
    }
    ...
}
```

Descriptions of methods on the Context class

You can call the following methods on a Context object. For more information, see the description of the context type in the *API Reference for EPL (ApamaDoc)*.

■ `public long getId()`

Returns the unique identifier for the context. For a Context instance that would return the following `toString()` result: `"context(2,"context_name",false)"`, the `getId()` method returns the value 2. This method returns 0 for a Context instance created with the default constructor.

■ `public String getName()`

Returns the name of the context. For example, suppose you create a context with the following EPL code:

```
context c := context("test");
```

If you transfer a reference to this context into your JMon application, a call to the `getName()` method on this context instance returns "test".

This method returns an empty string for a `Context` instance created with the default constructor.

■ `public String toString()`

Returns a string representation of the context instance. This method produces a string that is identical to the string that EPL produces. For example: "context(2, "context_name", false)". The first item in the string, 2 in this example, is the context's unique identifier. The second item in the string, "context_name", is the name of the context. The third item in the string is the value of the `receivesInput` boolean flag, which indicates whether the context is public or private.

This method returns "context(0, "", false)" for a `Context` instance created with the default constructor.

For details about public and private contexts, see [“Implementing Parallel Processing” on page 285](#) and [“Creating contexts” on page 288](#).

■ `public static Context getCurrent()`

Returns a `Context` instance that corresponds to the current correlator context. This is the context that contains the code that you are calling. Apama executes single-threaded JMon applications in the main correlator context. Consequently, this method always returns a `Context` instance that references the main correlator context.

During execution, JMon applications can create new Java threads. Do not confuse new threads with correlator contexts. The `Context.getCurrent()` method returns null when you call it inside newly created Java threads.

Identifying external events

In some situations, you might want to determine whether an event originated outside the correlator. To do this, call the `Event.isExternal()` method:

```
public boolean isExternal()
```

This method returns `true` if the event was sent to the correlator by some external process and that event was then passed into your JMon application.

Optimizing event types

[“About event types” on page 446](#) introduced event type classes.

The correlator creates several indexing data structures for every event type. The complexity and efficiency of these data structures depends on the number of parameters an event has, and therefore “smaller” (with less parameters) events are processed more rapidly.

Therefore, if possible, when designing an application it is preferable to control it using a number of “smaller” event types rather than through a single event type with a large number of parameters.

Wildcarding parameters in event types

Alternatively, if large event types are unavoidable, you can optimize performance by reviewing the usage of these event types in JMon, specifically within event templates in event expressions.

If a parameter of an event is never matched against directly within any event expressions, that is only * (or wildcard) ever appears against it in event templates, then the event type's definition can be amended to indicate this. This tells the correlator to ignore this parameter in its internal indexing.

Consider the event type definition presented in [“About event types” on page 446](#).

```
/*
 * Tick.java
 *
 * Class to abstract an Apama stock tick event. A stock tick event
 * describes the trading of a stock, as described by the symbol
 * of the stock being traded, and the price at which the stock was
 * traded
 *
 */
import com.apama.jmon.Event;

public class Tick extends Event {
    /** The stock tick symbol */
    public String name;

    /** The traded price of the stock tick */
    public double price;

    /**
     * No argument constructor
     */
    public Tick() {
        this("", 0);
    }

    /**
     * Construct a tick object and set the name and price
     * instance variables
     *
     * @param name The stock symbol of the traded stock
     * @param price The price at which the stock was traded
     */
    public Tick(String name, double price){
        this.name = name;
        this.price = price;
    }
}
```

If all references to this event type in event expressions look similar to this,

```
Tick("ACME", *)
```

that is, where the second parameter `price` is always specified as a `*`, then this parameter could be *wildcarded* in the event type definition.

This can be done by annotating the field in the event type class, as shown here

```
/** The traded price of the stock tick */
@com.apama.jmon.annotation.Wildcard
public double price;
```

This definition in the `Tick` class will override the default behavior, and it lets the correlator know that it can optimize its indexing by ignoring the `price` parameter.

As many parameters as desired can be wildcarded in this way. For example, if both `price` and `name` were to be wildcarded in `Tick`, they should be defined as follows,

```
/** The stock tick symbol */
@com.apama.jmon.annotation.Wildcard
public String name;

/**The traded price of the stock tick */
@com.apama.jmon.annotation.Wildcard
public double price;
```

Of course, if you were to do this, then

```
Tick(*, *)
```

would be the only valid event template that can be expressed in JMon. Any other expression would cause a Java runtime error.

Logging in JMon applications

The logging facilities in JMon are provided by Log4j, a publicly available logging library for Java. These logging facilities are included in `com.apama.util.Logger`. See the *API Reference for Java (Javadoc)*.

Note:

Full documentation for Log4j 2 can be found at <https://logging.apache.org/log4j/2.x/>.

It is recommended that you do not use the `FATAL` or `CRIT` log levels provided by the `Logger` class, which are present only for historical reasons. It is better to use `ERROR` for all error conditions regardless of how fatal they are, and `INFO` for informational messages. By default, the JMon classes log at `WARN` level. See "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications* for information about configuring log levels in the correlator.

To ensure that the correlator can serialize logging behavior, specify that instances of `Logger` are static.

Using EPL keywords as identifiers in JMon applications

If you use EPL keywords as event name or field identifiers, then in the following situations you must escape such identifiers by preceding them with hash (#) symbols:

- You refer to the JMon identifier in EPL code: You must escape the identifier in the EPL code that contains the reference.
- You refer to the JMon identifier in a JMon event expression: You must escape the identifier in that JMon event expression.

For example, consider the following Java code:

```
class test extends Event {  
    int id;  
    float price;  
    int integer;  
}
```

Now suppose you want to write the following EPL code:

```
on all test(id=7) as f {  
    print f.toString();  
    emit f;  
}
```

No escaping is necessary. However, suppose you want to write this EPL code:

```
print f.integer.toString();
```

In this case, you must escape `integer` as follows:

```
print f.#integer.toString();
```

Likewise, you must escape `integer` in the following JMon event expression:

```
new EventExpression("all test(#integer > 5)");
```

See also [“Keywords” on page 687](#).

15 Defining Event Expressions

■ About event templates	464
■ Specifying parameter constraints in event templates	466
■ Obtaining matching events	468
■ Emitting, routing, and enqueueing events	469
■ Specifying temporal sequencing	471
■ Defining advanced event expressions	473
■ Optimizing event expressions	485
■ Validation of event expressions	486

Consider this code snippet from the previous example:

```
public void onLoad() {
    EventExpression eventExpr =
        new EventExpression("Tick(*, >10.0)");
    eventExpr.addMatchListener(this);
}
```

The highlighted code is creating an event expression, and embeds the following event expression definition string:

```
Tick(*, >10.0)
```

This is the simplest form of an event expression; specifically it contains a single *event template*.

In this case, the event expression is specifying “the first “Tick” event whose “price” parameter contains a value greater than 10.0”.

If you are already familiar with EPL, the syntax for writing JMon event expressions is the same as for EPL event expressions.

About event templates

The first part of an event template defines the event type of suitable events (in this case `Tick`), while the section in brackets describes filtering criteria that must be applied to the contents of events of the desired type for them to match.

In the example at the beginning of the chapter, the first parameter within the event template has been set to a wildcard (*), specifying that all `Tick` events, regardless of the value of their `name` parameter, are suitable. That is, as long as their second parameter, `price`, is greater than 10. The filtering criteria supplied are applied to the event's contents in the same order as within the event definition for that event type. This is known as *positional syntax*.

[“Specifying parameter constraints in event templates” on page 466](#) lists all the filtering operators (like `>`) that can be applied to the value of a parameter within an event template.

Specifying positional syntax

In positional syntax, the event template must define a value (or a wildcard) to match against for every parameter of that event's type, in the same order as the parameter's definition in the event type definition. Therefore, for the event type,

```
public class MobileUser extends Event {
    public long userID;
    public Location position;
    public String hairColour;
    public String starsign;
    public long gender;
    public long incomeBracket;
    public String preferredHairColour;
    public String preferredStarsign;
    public long preferredGender;
    // ... Constructors
```



```
}
```

a suitable event template definition might look like

```
MobileUser(*,*, "red", "Capricorn", *, *, *, *, 1)
```

This can get unwieldy when you are working with event types with a large number of parameters and very few of them are actually being used to filter on. An alternative syntax can be used that addresses this. The above can instead be expressed as:

```
MobileUser(hairColour="red", starsign="Capricorn",  
preferredGender=1)
```

This is known as *named parameter syntax* and in this style all other non-specified fields are set to wildcard.

Given the following event types:

```
public class A extends Event {  
    public long a;  
    public String b;  
  
    // ... Constructors  
}  
  
public class B extends Event {  
    public long a;  
  
    // ... Constructors  
}  
  
public class C extends Event {  
    public long a;  
    public long b;  
    public long c;  
  
    // ... Constructors  
}
```

Here are some equivalent event expressions that demonstrate how to use the two syntaxes:

	Positional Syntax	Name/Value Syntax
Using constants and literals	on A(3,"string")	on A(a=3,b="string")
	on A(=3,="string")	on A(b="string",a=3)
Relational comparisons	on B(>3)	on B(a>3)
Ranges	on B([2:3])	on B(a in [2:3])
Wildcards	on C(*,4,*)	on C(b=4)
	on C(*,*,*)	on C(a=*,b=4,c=*)
		on C()

More details about the operators and expressions possible within event templates are given in the next section.

Note that it is possible to mix the two styles as long as you specify positional parameters before named ones. There cannot be any positional parameters after named ones. Therefore the following syntax is legal:

```
D(3,>4,i in [2:4])
```

while the following is not:

```
E(k=9,"error")
```

Specifying completed event templates

In some situations, you want to ensure that the correlator completes all work related to a particular event before your application performs some other work. In your event template, specify the `completed` keyword to accomplish this. For example:

```
on all completed A(f < 10.0) {}
```

When an event that matches the template comes into the correlator, the correlator

1. Runs all of the event's normal and unmatched listeners.
2. Processes all routed events that result from those listeners.
3. Calls the `completed` listeners.

Specifying parameter constraints in event templates

The first part of an event template defines the event type of the event the listener is to match against, while the section in brackets describes further filtering criteria that must be satisfied by the contents of events of that type for a match.

Event template parameter operators specify constraints that define what values, or range of values, are acceptable for a successful event match.

Operator	Meaning
<code>[value1 : value2]</code>	<p>Specifies a range of values that can match. The values themselves are included in the range to match against. For example:</p> <pre>stockPrice(*, [0 : 10])</pre> <p>This event template will match a <code>stockPrice</code> event where the price is between 0 and 10 inclusive. This range operator can only be applied to <code>double</code> and <code>long</code> types.</p>

Operator	Meaning
<code>[value1 : value2)</code>	<p>Specifies a range of values that can match. The first value itself is included while the second is excluded from the range to match against. For example:</p> <pre>stockPrice(*, [0 : 10))</pre> <p>This example will match a <code>stockPrice</code> event where the price is between 0 and 9 inclusive (assuming the parameter was of <code>long</code> type).</p> <p>This range operator can only be applied to <code>double</code> and <code>long</code> types.</p>
<code>(value1 : value2]</code>	<p>Specifies a range of values that can match. The first value itself is excluded from while the second is included in the range to match against. For example:</p> <pre>stockPrice(*, (0 : 10])</pre> <p>This example will match a <code>stockPrice</code> event where the price is between 1 and 10 inclusive (assuming the parameter was of <code>long</code> type).</p> <p>This operator can only be applied to <code>double</code> and <code>long</code> types.</p>
<code>(value1 : value2)</code>	<p>Specifies a range of values that can match. The values themselves are excluded in the range to match against. For example:</p> <pre>stockPrice(*, (0 : 10))</pre> <p>This example will match if a <code>stockPrice</code> event where the price is between 1 and 9 inclusive (assuming the parameter was of <code>long</code> type).</p> <p>This operator can only be applied to <code>double</code> and <code>long</code> types.</p>
<code>> value</code>	<p>All values greater than the value supplied will satisfy the condition and match.</p> <p>This operator can only be applied to <code>double</code> and <code>long</code> types.</p>
<code>< value</code>	<p>All values less than the value supplied will satisfy the condition and match. This operator can only be applied to <code>double</code> and <code>long</code> types.</p>
<code>>= value</code>	<p>All values greater than or equal to the value supplied will satisfy the condition and match.</p> <p>This operator can only be applied to <code>double</code> and <code>long</code> types.</p>
<code><= value</code>	<p>All values less than or equal to the value supplied will satisfy the condition and match.</p>

Operator	Meaning
	This operator can only be applied to double and long types.
<i>value</i>	Only a value equivalent to the value supplied will satisfy the condition and match. A String value must be enclosed in double quotes (" "), and therefore these need to be preceded with an escape character inside event expression definitions in an EventExpression constructor (\") A Location value must consist of a structure with four doubles representing the coordinates of the corners of the rectangular space being represented.
*	Any value for this parameter will satisfy the condition and match.

Obtaining matching events

An event template provides a definition against which several event instances could match. Once a listener triggers, sometimes it is necessary to get hold of the *actual* event that matched the template.

This can be achieved through *event tagging*.

If you are familiar with EPL, event tagging in JMon is similar in principle to variable coassignment in EPL. For this reason the term *coassigned* is sometimes used to refer to event tagging.

Consider this revised Simple monitor:

```
import com.apama.jmon.*;

public class Simple implements Monitor, MatchListener {

    /**
     * No argument constructor used by the jmon framework on
     * application loading
     */
    public Simple() {}

    /**
     * Implementation of the Monitor interface onLoad method. Sets up
     * a single event expression looking for all stock trade events
     * with a trade price of greater than 10.0. This class instantiation
     * is added as a match listener to the event expression.
     */
    public void onLoad() {
        EventExpression eventExpr = new EventExpression("Tick(*, >10.0):t");
        eventExpr.addMatchListener(this);
    }

    /**
     * Implementation of the MatchListener interface match method.
     * Extracts the tick event that caused the event expression to
```

```

    * trigger and emits the event onto the default channel
    */
    public void match(MatchEvent event) {
        Tick tick = (Tick)event.getMatchingEvents().get("t");
        System.out.println("Event details: " + tick.name
            + " " + tick.price);
        tick.emit();
    }
}

```

Note the revised event expression

```
Tick(*, >10.0):t
```

This specifies that when a suitable `Tick` event is detected, it must be recorded with the `t` tag. This allows a developer to get hold of the actual event that matched the event expression within the registered match listener's `match` method.

Once the `eventExpr` listener detects a suitable event it will trigger and call `match`, passing to it a `MatchEvent` object. This object embeds within it all the individual event instances that together caused the event expression to be satisfied and were tagged.

In this example our event expression still consists of a single event template, and since this is tagged, then the `MatchEvent` object will contain the single `Tick` event that triggered the `eventExpr` listener. This will be tagged as `t`.

A `MatchEvent` object has two methods:

- `HashMap getMatchingEvents()` - Get the set of tagged Events that caused the match. This method returns a Map of the tagged Event objects that hold the values that matched the source EventExpression.
- `Event getMatchingEvent(String key)` - Get one of the tagged Events that caused the match. This method returns the tagged Event object that matched in the source EventExpression.

For complete class and method signatures, refer to the *API Reference for Java (Javadoc)*.

The lines:

```
Tick tick = (Tick)event.getMatchingEvents().get("t");
```

or

```
Tick tick = event.getMatchingEvent("t");
```

show how the tagged event can be extracted by using the tag as a key.

Emitting, routing, and enqueueing events

Once the event has been extracted it can also be *emitted*, *routed*, or *enqueue*d.

This functionality is provided by the following methods of the `Event` class:

- `route()` — Route this event internally within the correlator.

- `emit()` — Emit this event from the correlator onto the default channel.
- `emit(String channel)` — Emit this event from the correlator onto the named channel.
- `enqueue()` — Route this event internally within the correlator to a special queue just for enqueued events.
- `enqueueTo()` — Route this event internally within the correlator to the input queue of the specified context or contexts.

The `route` method generates a new event that is dispatched back into the correlator. Any active listeners seeking that event then receive this. There is no difference between an externally sourced event (passed in through a live message feed) and an event that was issued internally through a `route` method, other than that internally routed events are placed at the front of the input queue, although in the same order as they are routed within an action.

The `emit` method dispatches events to external registered event receivers, that is, sends them out from the correlator. Active listeners will not receive events that are emitted.

Events are emitted onto named *channels*. For an application to receive events from the correlator it must register itself as an event receiver and *subscribe* to one or more channels. Then if events are emitted to those channels they will be forwarded to it.

Channels effectively allow both *point-to-point* message delivery as well as through *publish-subscribe*. Channels can be set up to represent topics. External applications can then subscribe to event messages of the relevant topics. Otherwise a channel can be set up purely to indicate a destination and have only one application connected to it.

The `enqueue()` method generates an event and places the event on a special queue just for events generated by the `enqueue()` method. A separate thread moves each enqueued event to the input queue of each public context. This arrangement ensures that if a public context's input queue is full, the event generated by `enqueue()` still arrives on its special queue, and is moved to that context's input queue when there is a room. Active listeners will eventually receive events that are enqueued, once those events make their way to the head of the context's input queue alongside normal events.

Use the `enqueue()` method when you want to ensure that the correlator processes the generated event after it processes all routed events. This means that you want the correlator to finish processing the current external event. Completion of processing the current external event means that all routed events that resulted from that external event have been processed.

In a parallel application, you can enqueue an event to a particular context by calling the following method on an instance of `com.apama.jmon.Event`:

```
public void enqueueTo(Context ctx)
```

This method provides the same functionality provided by the EPL `enqueue ... to` statement. See [“Sending an event to a particular context” on page 294](#).

However, it is important to mention that when you enqueue an event to a particular context the event goes on that context's input queue and not on the special queue for enqueued events. Consequently, when you call this method from an application thread that was created from the

main JMon application and the destination context's input queue is full, this method blocks until the queue is able to accept the event.

Call the following method to enqueue an event to a array of contexts:

```
public void enqueueTo(Context[] ctxArray)
```

Call the following method to enqueue an event to a list of contexts:

```
public void enqueueTo(List < Context> ctxList)
```

Specifying temporal sequencing

If you want to search for a temporal sequence of two events, for example, “locate the sequence of a “NewsItem” event followed by a “Tick” event”, there are two ways you can proceed in JMon.

Chaining listeners

You can *chain* listeners, as follows:

```
// Code within the monitor class

public void onLoad() {
    EventExpression eventExpr = new EventExpression("NewsItem(*, *)");
    eventExpr.addMatchListener(matchListener1);
}

// Code within the first Match Listener class - matchListener1
public void match(MatchEvent event) {

    // Arbitrary additional code ...
    EventExpression eventExpr = new EventExpression("Tick(*, *)");
    eventExpr.addMatchListener(matchListener2);
}

// Code within the second Match Listener class - matchListener2

public void match(MatchEvent event) {
    System.out.println("Detected a NewsItem followed"
        + " by a Tick event, both regarding any company.");
}
```

The Java code above shows how to set up a listener to seek the first event, and then once that is located, start searching for the second. This programming style is particularly appropriate when further actions need to be taken at each stage of the event detection, in this case between detecting the NewsItem and seeking the Tick.

It is also the only way in which the event templates can be “linked” together. If the desired effect was to locate “any” first NewsItem and then seek a Tick specifically for the same company mentioned in the NewsItem, you could amend the example as follows,

```
// Code within the monitor class
public void onLoad() {
    EventExpression eventExpr
        = new EventExpression("NewsItem(*, *):n");
```

```
    eventExpr.addMatchListener(matchListener1);
}

// Code within the first Match Listener class - matchListener1
public void match(MatchEvent event) {

    NewsItem newsItem = (NewsItem)event.getMatchingEvents().get("n");
    EventExpression eventExpr
        = new EventExpression("Tick(\"" + newsItem.name + "\", *)");
    eventExpr.addMatchListener(matchListener2);
}

// Code within the second Match Listener class - matchListener2
public void match(MatchEvent event) {
    System.out.println("Detected a NewsItem, followed"
        + " by an Tick event regarding the same company.");
}
```

Note how the above code seeks out a `NewsItem` on any company, but then extracts the actual `NewsItem` event detected, and uses its name parameter to create the event template for seeking the Tick event.

Using temporal operators

Let us return to how to express searching for a temporal sequence. If there is no requirement to execute any arbitrary code in between events and there is no requirement to link searches as illustrated above, then you can embed a temporal event expression within a single listener.

The first code excerpt could be re-written as follows,

```
// Code within the monitor class
public void onLoad() {
    EventExpression eventExpr
        = new EventExpression("NewsItem(*,*) -> Tick(*,*)");
    eventExpr.addMatchListener(matchListener1);
}

// Code within the first (and only) Match Listener class - matchListener1
public void match(MatchEvent event) {
    System.out.println("Detected a NewsItem followed"
        + " by a Tick event, both regarding any company.");
}
```

The event expression definition for `eventExpr` no longer consists of a single event template. It now has multiple clauses and contains a temporal operator.

In this case, the operator used is `->`, or the *followed-by* operator. This is the primary temporal operator for use in event expressions. It allows a developer to express a sequence of events to match against within a single listener, with the listener triggering once the whole sequence is encountered.

In Java, an event sequence does not imply that the events have to occur right after each other, or that no other events are allowed to occur in the meantime.

For the sake of brevity, let `A`, `B`, `C` and `D` represent event templates, and `A'`, `B'`, `C'` and `D'` be individual events that match those templates, respectively. If a listener is created to seek the event

expression `(A -> B)`, the event feed `{A', C', B', D'}` would result in a match once the `B'` is received by the correlator.

Followed-by operators can be chained to express longer sequences. Therefore you could write,

```
A -> B -> C -> D
```

within an event expression definition.

The next section focuses on the use of temporal operators in event expressions.

Defining advanced event expressions

An event template is the simplest form of an event expression. All event expression operators, including `->`, can themselves take entire event expressions as operands.

It is useful to think of event expressions as being Boolean expressions. Each clause in an event expression can be `true` or `false`, and the whole event expression must evaluate to `true` before the listener triggers and calls the match listener's `match` method.

As before, for the sake of brevity, let us use the letters `A`, `B`, `C` and `D` to represent event templates, and `A'`, `B'`, `C'` and `D'` to represent individual events that match those templates, respectively.

Once more, consider this representation of an event expression,

```
A -> B -> C -> D
```

When the listener is first activated it is helpful to consider the expression as starting off by being `false`. When an event that satisfies the `A` clause occurs, the `A` clause becomes `true`. Once `B` is satisfied, `A -> B` becomes `true` in turn, and evaluation progresses in a similar manner until eventually all `A -> B -> C -> D` evaluates to `true`. Only then does the listener trigger and call the associated match listener's `match` method. Of course, this event expression might never become `true` in its entirety (as the events required might never occur) since no time constraint (see [“Specifying the timer operators” on page 481](#)) has been applied to any part of the event expression.

Specifying other temporal operators

For a listener to trigger on an event sequence, the event expression defining what to match against must evaluate to `true`.

The `or` operator allows you to specify event expressions where a variety of event sequences could lead to a successful match. It effectively evaluates two event templates (or entire nested event expressions) simultaneously and returns `true` when either of them become `true`.

For example,

```
A or B
```

means that either `A` or `B` need to be detected to match. That is, the occurrence of one of the operand expressions (an `A` or a `B`) is enough to satisfy the listener.

The `and` operator specifies an event sequence that might occur in any temporal order. It evaluates two event templates (or nested event expressions) simultaneously but only returns `true` when they are both `true`.

```
A and B
```

will seek “an “A” followed by a “B or “a “B” followed by an “A. Both are valid matching sequences, and the listener will seek both concurrently. However, the first to occur will terminate all monitoring and trigger the listener.

The following example code snippets indicate a few patterns that can be expressed using the three operators presented so far.

Example	Meaning
<code>A -> (B or C)</code>	Match on an A followed by either a B or a C.
<code>(A -> B) or C</code>	Match on either the sequence A followed by a B, or just a C on its own.
<code>A -> ((B -> C) or (C -> D))</code>	Find an A first, and then seek for either the sequence B followed by a C or C followed by a D. The latter sequences will be looked for concurrently, but the monitor will match upon the first complete sequence that occurs. This is because the <code>or</code> operator treats its operands atomically; that is, in this case it is looking for the sequences themselves rather than their constituent events.
<code>(A -> B) and (C -> D)</code>	<p>Find the sequence A followed by a B (A -> B) followed by the sequence C -> D, or else the sequence C -> D followed by the sequence A -> B. The <code>and</code> operator treats its operands atomically; that is, in this case it is looking for the sequences themselves and the order of their occurrence, rather than their constituent events. It does not matter when a sequence starts but it occurs when the last event in it is matched.</p> <p>Therefore {A', C', B', D'} would match the specification, because it contains an A -> B followed by a C -> D. In fact the specification would match against either of the following sequences of event instances; {A', C', B', D'}, {C', A', B', D'}, {A', B', C', D'}, {C', A', D', B'}, {A', C', D', B'}, and {C', D', A', B'}.</p>

The `not` operator is unary and acts to invert the truth value of the event expression it is applied to.

```
A -> B and not C
```

therefore means that the correlator will match only if it encounters an A followed by a B without a C occurring at any time before the B is encountered.

Note:

The not operator can cause an event expression to reach a state where it can never evaluate to true any more, that is, it will become *permanently false*.

Consider this listener event sequence:

```
on (A -> B) and not C
```

The listener will start seeking both `A -> B` and `not C` concurrently. If an event matching `C` is received at any time before one matching `B`, the `C` clause will evaluate to `true`, and hence `not C` will become `false`. This will mean that `(A -> B) and not C` will never be able to evaluate to `true`, and hence this listener will never trigger. In practice the correlator cleans out these *zombie* listeners periodically.

Note:

It is possible to write an event expression that always evaluates to `true` immediately, without any events occurring.

Consider this listener:

```
on (A -> B) or not C
```

Assuming that `A`, `B`, and `C` represent event templates, their value will start off as being `false`. However, that means that `not C` will become `true` immediately, and hence the whole expression will become `true` right away. This listener will therefore trigger immediately as soon as it is instantiated. If any of `A`, `B` or `C` were nested event expressions the same logic would apply for their own evaluation.

Specifying a perpetual listener for repeated matching

So far all the examples given have created listeners that will trigger on the first occurrence of an event (or sequence of events) that satisfies the supplied event expression.

For example,

```
public void onLoad() {
    EventExpression eventExpr = new EventExpression("Tick(*, >10.0)");
    eventExpr.addMatchListener(this);
}
```

locates the *first* occurrence of a `Tick` event that satisfies the `Tick(*, >10.0)` event template. This first suitable event triggers the listener and calls the `match` method of the registered match listener object.

However, you might want to detect *all* `Tick` events that satisfy the above event template (or event expression). To do this you must create a *perpetual* listener, that is, one that does not terminate on the first suitable occurrence, but instead stays alive and triggers repeatedly on every subsequent occurrence.

This effect can be achieved through use of the `all` event expression operator.

If the above is rewritten as follows,

```
public void onLoad() {
    EventExpression eventExpr =
        new EventExpression("all Tick(*, >10.0)");
    eventExpr.addMatchListener(this);
}
```

the listener created will now seek the first `Tick` event whose price is greater than 10. Upon detecting such an event it will trigger and call the `match` method. It will then return to monitoring the incoming event streams to look for the next suitable occurrence. This behavior will be repeated indefinitely until the listener is explicitly deactivated. This means that potentially the `match` method could be invoked multiple times.

Deactivating a listener

A listener whose event expression embeds an `all` operator will stay active indefinitely and trigger repeatedly. It will continue doing this until it is explicitly deactivated. This can be done using the `removeMatchListener` method on the `EventExpression` object.

For complete class and method signatures, refer to the *API Reference for Java (Javadoc)*.

Temporal contexts

Imagine that we have seven event templates defined, which for the sake of brevity are represented by the letters A, B, C, D, E, F and G in the following text. Now, consider a stream of incoming events, where x_n indicates an event instance that matches the event template x . Likewise, x_{n+1} indicates another event instance that matches against x , but which need not necessarily be identical to x_n .

Consider the following sequence of incoming events:

```
C1 A1 F1 A2 C2 B1 D1 E1 B2 A3 G1 B3
```

Given the above event sequence, what should the event expression

```
A -> B
```

match upon?

In theory the combinations of events that correspond to “an “A” followed by a “B” are:

```
{A1, B1}, {A1, B2}, {A1, B3}, {A2, B1}, {A2, B2}, {A2, B3}, {A3, B3}
```

In practice it is unlikely that a developer wanted their monitor to match seven times on the above example sequence, and it is uncommon for all the combinations to be useful.

In fact, consistent with the truth-value based matching behavior already described, the event expression `A -> B` will only match on the first event sequence that matches the expression. Given the above event sequence the listener will trigger only on `{A1, B1}`, call the associated `match` method, and then terminate.

If a developer wishes to alter this behavior, and have the monitor match on more of the combinations, they can use the `all` operator within the event expression.

If the listener's specification was rewritten to read:

```
all A -> B
```

the listener would match on “every” A and the first B that follows it.

The way this works is that upon encountering an A, a second *child* listener (or *sub-listener*) is created to seek for the next A. Both listeners would continue looking for a B to successfully match the sequence specified. If more As are encountered the procedure is repeated; this behavior continues until the *parent* listener is explicitly deactivated.

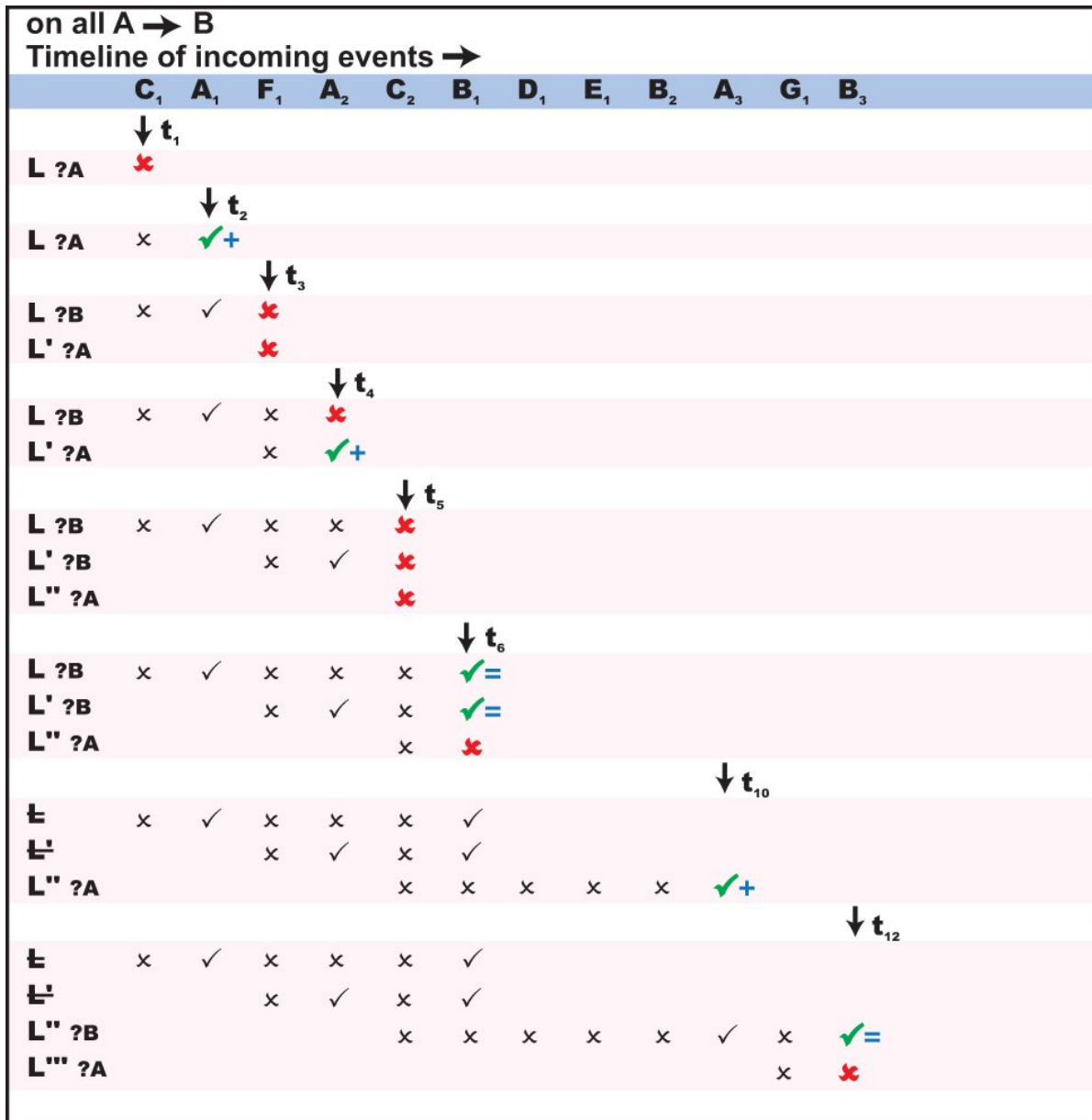
Therefore `all A -> B` would match on {A1, B1}, {A2, B1} and {A3, B3}.

Note that `all` is a unary operator and has higher precedence than `->`, `or` and `and`. Therefore `all A -> B` is the same as `(all A) -> B` or `((all A) -> B)`.

The following table illustrates how the execution of `all A -> B` proceeds over time as the above sequence of input events is processed by the correlator. The timeline is from left to right, and each stage is labeled with a time t_n , where t_{n+1} occurs after t_n . To the left are listed the listeners, and next to each one (after the ?) is shown what event template that listener is looking for at that point in time. In the example, assuming `L` was the initial listener, `L'`, `L''` and `L'''` are other sub-listeners that are created as a result of the `all` operator.

Guide to the symbols used:

This symbol	indicates the following
↓	A specific point in time when a particular event is received.
✗	No match was found at that time.
✓	The listener has successfully located an event that matches its current active template.
=	A listener has successfully triggered.
+	A new listener is going to be created.



The parent listener denoted by $\text{all } A \rightarrow B$ will never terminate as there will always be a sub-listener active looking for an A.

If, on the other hand, the specification is written as,

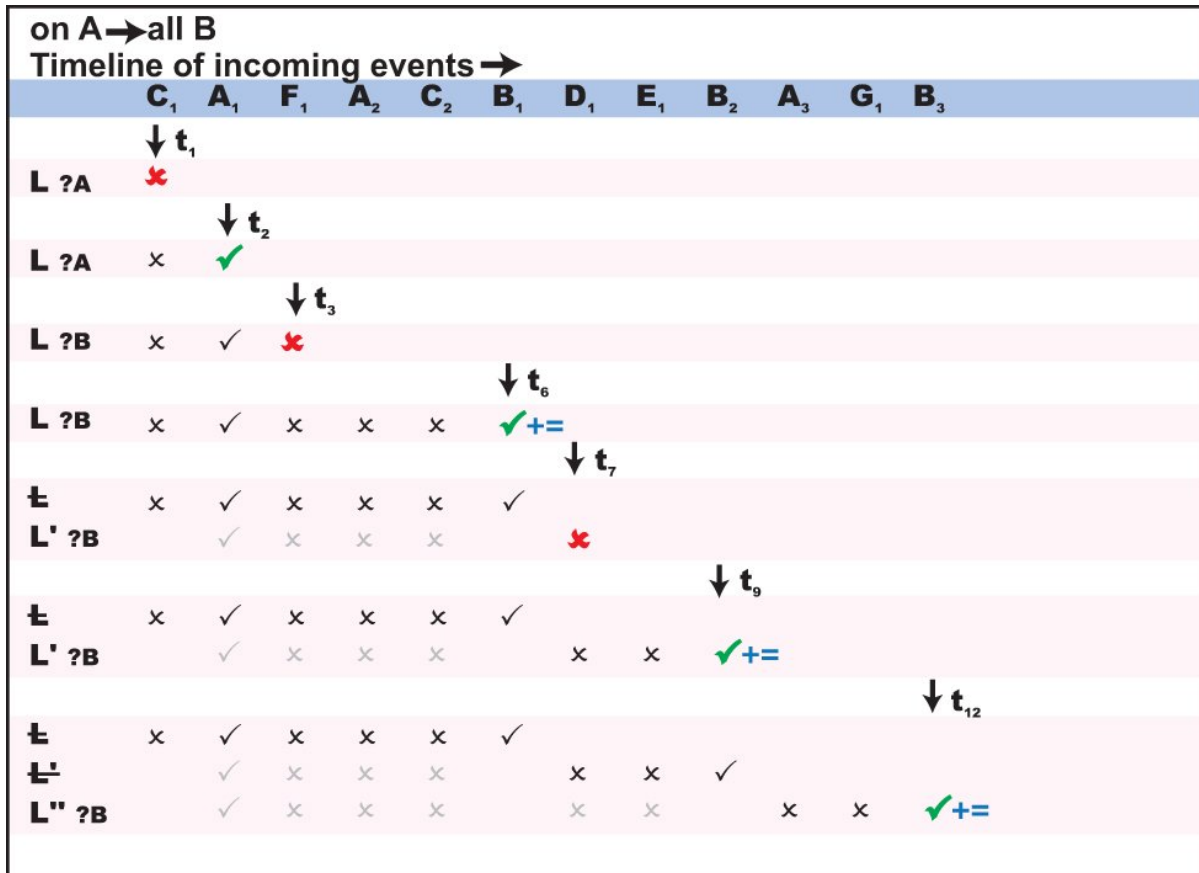
$A \rightarrow \text{all } B$

the listener would now match on all the sequences consisting of the first A and each possible following B.

The way this works is by creating a second listener upon matching a B that then goes on to search for an additional B, and so on repeatedly until the listener is explicitly killed.

Therefore $A \rightarrow \text{all } B$ would match $\{A_1, B_1\}$, $\{A_1, B_2\}$ and $\{A_1, B_3\}$.

Graphically this would now look as follows:



The table shows the early states of L' and L'' in light color because those listeners actually never really went through those states themselves. However, since they were created as a clone of another listener, it is as though they were.

The parent listener denoted by A → all B will never terminate, as there will always be a sub-listener looking for a B.

The final permutation is to write the monitor as,

```
all A -> all B
```

Now the listener would match on an A and create another listener to look for further As. Each of these listeners will go on to search for a B after it encounters an A. However, in this instance all listeners are duplicated once more after matching against a B.

The effect of this would be that all A → all B would match {A₁, B₁}, {A₁, B₂}, {A₁, B₃}, {A₂, B₁}, {A₂, B₂}, {A₂, B₃} and {A₃, B₃}, i.e. all the possible permutations. This could cause a very large number of sub-listeners to be created.

Note:

The all operator must be used with caution as it can create a very large number of sub-listeners, all looking for concurrent patterns. This is particularly applicable if multiple all operators are nested within each other. This can have an adverse impact on performance.

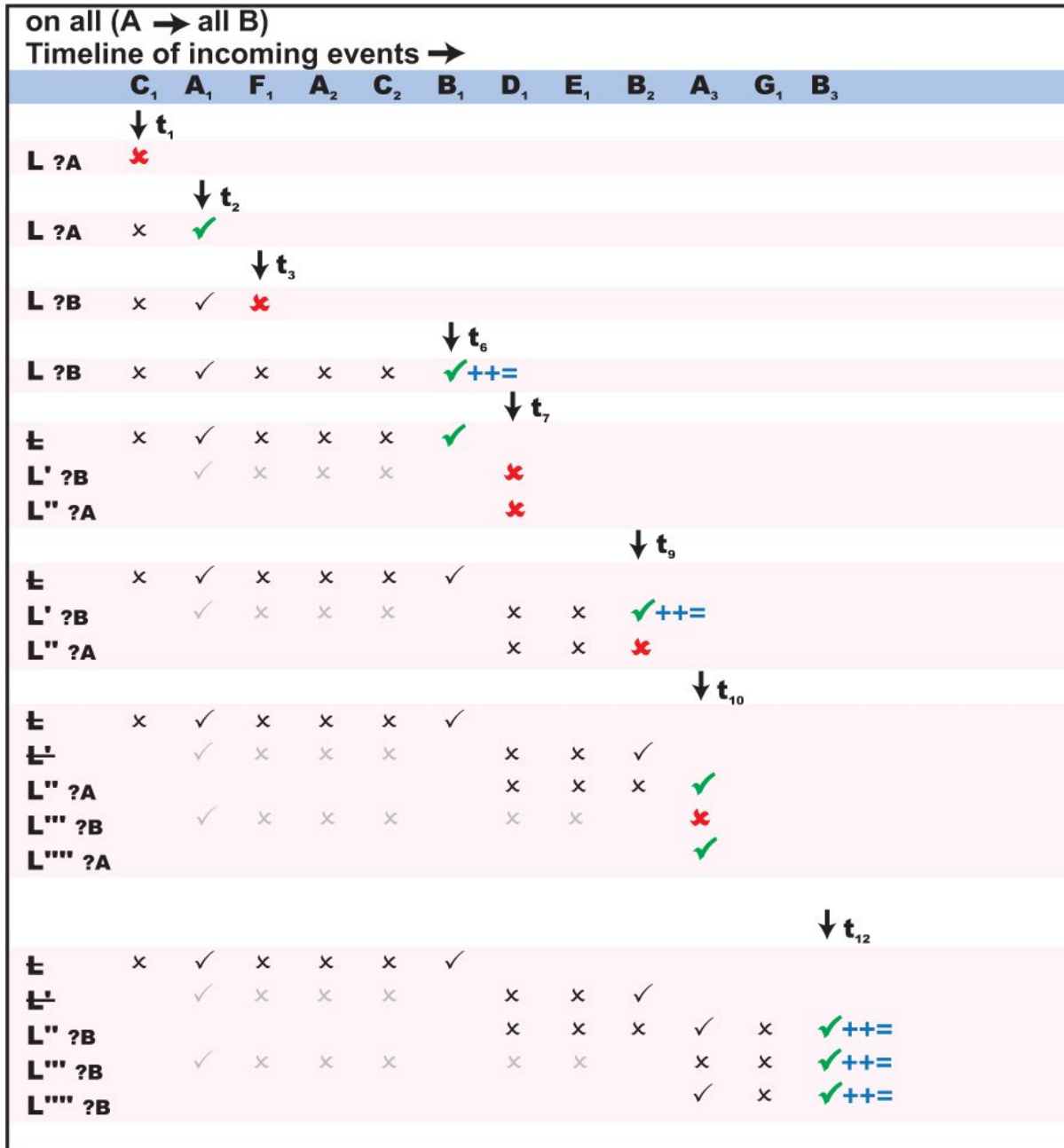
As with all other event expression operators, the `all` operator can be used within nested event expressions, and be nested within the operating context of another `all` operator. This can have a dramatic effect on the number of sub-listeners created.

Consider the example,

```
all (A -> all B)
```

This will match the first A followed by all subsequent Bs. However, as on every match of an A followed by B, `(A -> all B)` becomes `true`, then a new search for the *next* A followed by all subsequent Bs will start. This will repeat itself recursively, and eventually there could be several concurrent sub-listeners that might match on the same sequences, thus causing duplicate triggering.

On the same event sequence as previously, graphically, this would be evaluated as follows:



Thus matching against $\{A_1, B_1\}$, $\{A_1, B_2\}$, $\{A_1, B_3\}$, and twice against $\{A_3, B_3\}$. Notice how the number of active listeners is progressively increasing, until after t_{12} there would actually be six active listeners, three looking for a B and three looking for an A.

Specifying the timer operators

So far we have shown how to use event expressions to define interesting sequences of events to look for, where the events of interest depend not only on their type and content, but also on their temporal relationship to (whether they occur before or after) other events.

Being able to define temporal relationships can be useful, but typically it also needs to be constrained over some temporal interval.

Looking for event sequences within a set time

Consider this earlier example:

```
// Code within the monitor class
public void onLoad() {
    EventExpression eventExpr = new EventExpression(
        "NewItem(\"ACME\",*) -> Tick(\"ACME\",*)");
    eventExpr.addMatchListener(matchListener1);
}

// Code within the first (and only) Match Listener
// class - matchListener1

public void match(MatchEvent event) {
    System.out.println("Detected a NewItem followed"
        + " by an Tick event, both regarding the ACME company.");
}
```

This will look for the event sequence of a news item about a company followed by a stock price tick about that company. Once improved this could be used to detect the beginning of a rise (or fall) in the value of shares of a company following the release of a relevant news headline.

However, unless a temporal constraint is put in place, the monitor is not going to be that pertinent, as it might trigger on an event sequence where the price change occurs weeks after the news item. That would clearly not be so useful to a trader, as the two events were most likely unrelated and hence not indicative of a possible trend.

If the event expression above is rewritten as follows,

```
EventExpression eventExpr = new EventExpression(
    "NewItem(\"ACME\",*) -> Tick(\"ACME\",*) within(30.0)");
```

the `Tick` event would now need to occur within 30 seconds of `NewItem` for the listener to trigger.

The `within(float)` operator is a postfix unary operator that can be applied to an event expression (the `Tick` event template in the above example). Think of it like a stopwatch. The clock starts ticking as soon as the event expression it is attached to becomes active, i.e. when the listener actually starts looking for it. If the stopwatch reaches the specified figure before the event expression evaluates to true the event expression becomes permanently false.

In the above code, the timer is only activated once a suitable `NewItem` is encountered. Unless an adequate `StockTick` then occurs within 30 seconds and makes the expression evaluate to true, the timer will fire and fail the whole listener.

As already specified, the `within` operator can be applied to any event expression, hence `A within(x)`, where `A` represents just an event template and `x` is a float value specifying a number in seconds, is perfectly valid.

Waiting within a listener

The second timer operator available for use within event expressions is `wait(float)`.

`wait` allows you to insert a “temporal pause” within an event expression. Once activated, a `wait` expression becomes `true` automatically once a set amount of time passes. For example,

```
A -> wait(x seconds) -> C
```

will proceed as follows: activate the listener and look for the `A` event expression or template, then once `A` becomes `true`, pause (that is, wait) for `x` seconds, then finally start looking for the `C` event expression or template.

In addition to being part of an event expression, `wait` can also be used on its own,

```
wait(20.0)
```

is a valid event expression in its own right. When its listener activates it just waits for the number of seconds specified (here being 20), then it evaluates to `true` and calls any registered `match` methods.

Therefore a `wait` clause starts off being `false`, and then turns to `true` once its time period expires. This behavior can be inverted through use of `not`. The expression

```
not wait(20.0)
```

would start off being `true`, and stay `true` for 20 seconds before becoming `false`.

The following,

```
B and not wait(20.0)
```

is an interesting example. It effectively means that this listener will trigger only if a `B` occurs within 20 seconds of its activation. After that, the `not wait(20)` clause would become `false` and prevent the listener from ever triggering.

By using `all` with `wait`, you can easily implement a periodic repeating timer:

```
all wait(5.0)
```

This listener will trigger every 5 seconds and calls any registered `match` methods.

Working with absolute time

The final temporal operator is the `at` operator. This operator allows you to express temporal activity with regards to absolute time.

The `at` operator allows triggering of a timer:

- **At a specific time**; for example, at 12:30pm on April, 5th.
- **Repeatedly** with regards to the calendar when used in conjunction with the `all` operator, across seconds, minutes, hours, days of the week, days of the month, and months; for example, on every hour, or on the first day of the month, or every 10 minutes past and 40 minutes past.

Important:

Triggering using the `at` operator always uses the time zone in which the correlator is running.

The syntax is as follows:

```
at(minutes, hours, days_of_the_month, month, days_of_the_week [, seconds])
```

where the last operand, *seconds*, is optional.

Valid values for each operand are as follows:

Timer operand	Values
<i>minutes</i>	0 to 59, indicating minutes past the hour.
<i>hours</i>	0 to 23, indicating the hours of the day.
<i>days_of_the_month</i>	1 to 31, indicating days of the month. For some months only 1 to 28, 1 to 29 or 1 to 30 are valid ranges.
<i>month</i>	1 to 12, indicating months of the year, with 1 corresponding to January
<i>days_of_the_week</i>	0 to 6, indicating days of the week, where 0 corresponds to Sunday.
<i>seconds</i>	0 to 59, indicating seconds past the minute.

The operator can be embedded within an event expression in a manner similar to the `wait` operator. If used outside the scope of an `all` operator it will trigger only once, at the *next* valid time as expressed within its elements. In conjunction with an `all` operator, it will trigger at *every* valid time.

The wildcard symbol (*) can be specified to indicate that all values are valid, that is

```
at(5, *, *, *, *)
```

would trigger at the next “five minutes past the hour”, while

```
all at(5, *, *, *, *)
```

would trigger at five minutes past each hour (that is, every day, every month).

Whereas,

```
all at(5, 9, *, *, *)
```

would trigger at 9:05am every day.

However,

```
all at(5, 9, *, *, 1)
```

would trigger at 9:05am only on Mondays, and never on any other weekday. This is because the effect of the wildcard operator is different when applied to the *days_of_the_week* and the

`days_of_the_month` elements. This is due to the fact that both specify the same entity. The rule is therefore as follows:

- As long as both elements are set to wildcard, then each day is valid.
- If either of the `days_of_the_week` or the `days_of_the_month` elements is not a wildcard, then only the days that match that element will be valid. The wildcard in the other element is effectively ignored.
- If both the `days_of_the_week` and the `days_of_the_month` elements are not a wildcard, then the days valid will be the days which match either. That is, the two criteria are *or*, not *and*.

A range operator (`:`) can be used with each element to define a range of valid values. For example

```
all at(5:15, *, *, *, *)
```

would trigger every minute from 5 minutes past the hour till 15 minutes past the hour.

A divisor operator (`/x`) can be used to specify that every *x*th value is valid. Therefore

```
all at(* / 10, *, *, *, *)
```

would trigger every ten minutes, that is, at 0, 10, 20, 30, 40 and 50 minutes past every hour.

If you wish to specify a combination of the above operators you must enclose the element in square brackets (`[]`), and separate the value definitions with a comma (`,`). For example,

```
all at([* / 10, 30:35, 22], *, *, *, *)
```

indicates as following values for minutes to trigger on: 0, 10, 20, 22, 30, 31, 32, 33, 34, 35, 40 and 50.

A further example,

```
all at(* / 30, 9:17, [* / 2, 1], *, *)
```

would trigger every 30 minutes from 9am to 5pm on even numbered days of the month as well as specifically the first day of the month.

Optimizing event expressions

When a developer creates an event expression, a substantial percentage of the computational overhead goes into parsing the event expression itself.

If you need to create several instances of an event expression where only literal values in event templates vary, this repeated parsing cost can be removed through the use of a *prepared* event expression.

Instead of writing,

```
EventExpression eventExpr1 = new EventExpression(
    "NewItem(\"ACME\",*) -> Tick(\"ACME\",*)");
EventExpression eventExpr2 = new EventExpression(
    "NewItem(\"EMCA\",*) -> Tick(\"EMCA\",*)");
eventExpr1.addMatchListener(matchListener1);
eventExpr2.addMatchListener(matchListener2);
```

you could write,

```
PreparedEventExpressionTemplate et
    = new PreparedEventExpressionTemplate(
        "NewItem(?,*) -> Tick(?,*)");

PreparedEventExpression pex1=et.getInstance();
pex1.setString(0, "ACME");
pex1.setString(1, "ACME");

PreparedEventExpression pex2=et.getInstance();
pex2.setString(0, "EMCA");
pex2.setString(1, "EMCA");

pex1.addMatchListener(matchListener1);
pex2.addMatchListener(matchListener2);
```

The above example shows how instead of creating two very similar event expressions you can create a single prepared event expression template, and then customize multiple instances of it. The main advantage of the latter approach is the fact that the event expression was parsed in Java only once. With an example as simple as the ones above this would in fact hardly make any difference, but in Java code with hundreds of such event expressions the difference in performance can be significant.

As shown in the code snippet above, the procedure for creating listeners with prepared event expressions is slightly different from that of normal event expressions.

You must create a `PreparedEventExpressionTemplate` and define within that the event expression. The syntax for event expression definitions is the same as previously with the exception of the `?` operator. This can be used instead of any literal value. The next step is to get an instance of a `PreparedEventExpression`, and then to set values for any literals replaced by `?` in the prepared event expression template. Finally, you can create listeners on the `PreparedEventExpression` instances just as with normal event expressions.

Validation of event expressions

When an `EventExpression` or `PreparedEventExpressionTemplate` is created or when `addMatchListener()` is called on an event expression within a `JMon` monitor the event expression is not validated immediately. It is queued for processing later when the `JMon` monitor yields control back to the correlator. This means that a badly formed event expression does not cause an exception to be thrown from the constructor. Instead, the correlator logs an error message later when it tries to validate the event expression.

16 Concept of Time in the Correlator

■ Getting the current time	488
■ About timers and their trigger times	489

An understanding of how the correlator handles time is essential to writing Apama applications. See [“Understanding time in the correlator” on page 179](#) for more information. Note that the information provided there applies when developing Apama applications in EPL. Most of this, however, and especially the following topics also apply when developing Apama applications in Java:

- [“Correlator timestamps and real time” on page 179](#)
- [“Event arrival time” on page 180](#)
- [“Disabling the correlator's internal clock” on page 182](#)

The topics in this section are specific to Java. Similar topics are provided for developing Apama applications in EPL.

Getting the current time

In the correlator, the current time is the time indicated by the most recent clock tick. There are two exceptions to this:

- If you specify the `-xclock` option when you start the correlator, the correlator does not generate clock ticks. Instead, you must send time events (`&TIME`) to the correlator. The current time is the time indicated by the most recent externally generated time event. See [“Externally generating events that keep time \(&TIME events\)” on page 183](#).
- When the correlator is firing a timer, the current time is the timer's trigger time. See [“About timers and their trigger times” on page 489](#).

The information in the remainder of this topic assumes that the current time is the time indicated by the most recent clock tick.

Use the static method `double com.apama.jmon.Correlator.getCurrentTime()` to obtain the current time. The value returned by the `getCurrentTime()` method is the current time represented as seconds since the epoch, January 1st, 1970 in UTC.

In the correlator, the current time is never the same as the current system time. In most circumstances it is a few milliseconds behind the system time. This difference increases when public context input queues grow.

When a listener triggers, it causes a call to the listener's `match()` method. The correlator executes the entire method before the correlator starts to process another event. Consequently, while the listener is executing a method, time and the value returned by the `getCurrentTime()` method do not change.

Consider the following code snippet,

```
double a;
void checkTime() {
    a = Correlator.getCurrentTime();
}

// A listener calls the following method some time later
void logTime() {
```



```

System.out.println("a: "+a );
// The time when checkTime() was called

System.out.println("current time: "+Correlator.getCurrentTime() );
// The time now
}

```

In this code, a method sets double variable `a` to the value of `getCurrentTime()`, which is the time indicated by the most recent clock tick. Some time later, a different listener prints the value of `a` and the value of `getCurrentTime()`. The values logged might not be the same. This is because the first use of `getCurrentTime()` might return a value that is different from the second. If the two listeners have processed the same event, the logged values are the same. If the two listeners have processed different events, the logged values are different.

About timers and their trigger times

In an event expression, when you specify the `within`, `wait`, or `at` operator you are specifying a timer. Every timer has a trigger time. The trigger time is when you want the timer to fire.

- When you use the `within` operator, the trigger time is when the specified length of time elapses. If a `within` timer fires, the listener fails. In the following listener, the trigger time is 30 seconds after `A` becomes `true`.

```
A -> B within(30.0)
```

If `B` becomes `true` within 30 seconds, the trigger time for the timer is not reached, the timer does not fire, the listener triggers, and the monitor calls any attached JMon listeners. If `B` does not become `true` within 30 seconds, the trigger time is reached, the timer fires, and the listener fails. The monitor does not call the `MatchListener`.

- When you use the `wait` operator, the trigger time is when the specified pause during processing of the event expression has elapsed. When a `wait` timer fires, processing continues. In the following expression, the trigger time is 20 seconds after `A` becomes `true`. When the trigger time is reached, the timer fires. The listener then starts watching for `B`. When `B` is `true`, the monitor calls any attached listeners.

```
A -> wait(20.0) -> B
```

- When you use the `at` operator, the trigger time is one or more specific times. An `at` timer fires at the specified times. In the following expression, the trigger time is five minutes past each hour every day. This timer fires 24 times each day. When the timer fires, the monitor calls any attached JMon listeners.

```
all at(5, *, *, *, *)
```

At each clock tick, the correlator evaluates each timer to determine whether that timer's trigger time has been reached. If a timer's trigger time has been reached, the correlator fires that timer. When a timer's trigger time is exactly at the same time as a clock tick, the timer fires at its exact trigger time. When a timer's trigger time is not exactly at the same time as a clock tick, the timer fires at the next clock tick. This means that if a timer's trigger time is .01 seconds after a clock tick, that timer does not fire until .09 seconds later.

When a timer fires, the current time is always the trigger time of the timer. This is regardless of whether the timer fired at its trigger time or at the first clock tick after its trigger time.

A single clock tick can make a repeating timer fire multiple times. For example, if you specify `all wait(0.01)`, this timer fires 10 times every tenth of a second.

Because of rounding constraints,

- A timer such as `all wait(0.1)` drifts away from firing every tenth of a second. The drift is of the order of milliseconds per century, but you can notice the drift if you convert the value of the current time to a string.
- Two timers that you might expect to fire at the same instant might fire at different, though very close, times.

The rounding constraint is that you cannot accurately express 0.1 seconds as a float because you cannot represent it in binary notation. For example, the `on wait(0.1)` listener waits for 0.10000000000000000555 seconds.

To specify a timer that fires exactly 10 times per second, calculate the length of time to wait by using a method that does not accumulate rounding errors. For example, calculate a whole part and a fractional part:

```
@Application(author="Tim Berners", company="Apama",
description="Demonstrate tenth of second timers", name="Tenth",
version="1.0")
@MonitorType
public class TenthOfSecond implements Monitor {

    private static final Logger LOGGER =
        Logger.getLogger(TenthOfSecond.class);
    private static final NumberFormat formatter =
        NumberFormat.getInstance();
    static { formatter.setGroupingUsed(false); }

    double startTime;
    double fraction;

    public void onLoad() {
        startTime = Math.ceil( Correlator.getCurrentTime() );
        fraction = Math.ceil(
            (Correlator.getCurrentTime() - startTime) * 10.0);
        setupTimeListener();
    }
    void setupTimeListener() {
        fraction++;
        if (10.0 <= fraction) {
            fraction = 0.0;
            startTime++;
        }
        EventExpression ee = new EventExpression("wait("+ ((startTime +
            (fraction / 10.0))-Correlator.getCurrentTime()) +")");
        ee.addMatchListener(new MatchListener() {
            public void match(MatchEvent evt) {
                LOGGER.info(formatter.format(Correlator.getCurrentTime()));
                // System.out.println(Correlator.getCurrentTime());
                // This would go to STDOUT, and isn't as pretty
            }
        });
    }
}
```

```
        new TestEvent(Correlator.getCurrentTime()).emit();
        setupTimeListener();
    }
    });
}
// TenthOfSecond
```

When a timer fires, the correlator processes items in the following order. The correlator

1. Triggers all listeners that trigger at the same time.
2. Routes any events, and routes any events that those events route, and so on.
3. Fires any timers at the next trigger time.

17 Developing and Deploying JMon Applications

■ Steps for developing JMon applications in Software AG Designer	494
■ Java prerequisites for using Apama's JMon API	495
■ Steps for developing JMon applications manually	496
■ Deploying JMon applications	496
■ Removing JMon applications from the correlator	497
■ Creating deployment descriptor files	497
■ Package names and namespaces in JMon applications	505
■ Sample JMon applications	505

This section describes the steps required to develop and deploy a JMon application. You can develop JMon applications with Apama in Software AG Designer or manually, outside Software AG Designer. When you use Software AG Designer, some development steps are performed automatically for you. This section describes all development steps and notes which steps Software AG Designer automatically performs.

For more information on developing JMon applications in Software AG Designer, see "Working with Projects" in *Using Apama with Software AG Designer*.

See also [“Writing EPL Plug-ins in Java” on page 527](#) which describes how it is possible to call out to code written in Java even when the main application logic is written in EPL rather than JMon.

Steps for developing JMon applications in Software AG Designer

> To develop JMon applications in Software AG Designer

1. Add Java support to a project.

See "Adding the Java nature to an Apama project" in *Using Apama with Software AG Designer*.

2. Create your application's source files.

Select **File > New > Java Event** or select **File > New > Java Monitor**.

Or, in the **Project Explorer**, right-click your project and select **New > Java Event** or select **New > Java Monitor**.

A wizard appears that lets you specify the event or monitor's name, the package, a description, the Java source folder and Java package. Software AG Designer automatically adds an entry for the event or monitor to the `jmon-jar.xml` deployment descriptor file and regenerates the JMon JAR file to include the new event or monitor.

If you want to build your JAR files manually, right-click your project and select **Apama > Build JAR Files**. This is useful if you unselected the **Build jar files automatically** option in the `apama_java.xml` file, which is in the `config` directory of your project. One reason you might not want to build the JAR files automatically is that the build takes too long. When **Build jar files automatically** is selected, Software AG Designer builds the JAR files every time you modify a JMon file.

If there are events that you defined in JMon and you refer to those events, or listen for those events in EPL code, then you must define those events in EPL as well as JMon. If you do not also define the events in EPL, Software AG Designer flags EPL references to those events as errors.

See also "Creating new files for JMon applications" in *Using Apama with Software AG Designer*.

3. Create your application's launch configuration.

Software AG Designer adds all JMon JAR files to the correlator initialization list and all non-JMon JAR files to the correlator class path.

If you want to build your project's files outside Software AG Designer and Eclipse, right-click your project and select **Apama > Generate Ant Buildfiles**. Software AG Designer generates an Ant build file (with the name `build-project_name.xml`), which you can use only to build your project's JMon JAR files outside of the Eclipse environment. Note that this is unrelated to the Software AG Designer feature for exporting an Ant build file that you can use for deployment.

See "Defining custom launch configurations" in *Using Apama with Software AG Designer*.

4. Run and test your application.

See "Launching Projects" in *Using Apama with Software AG Designer*.

5. Debug your application.

See "Debugging JMon Applications" in *Using Apama with Software AG Designer*.

6. Deploy your application.

See ["Deploying JMon applications" on page 496](#).

Software AG Designer generates your application's JMon JAR file in the **jmon_config_name java application files** folder of your project's directory. By default, **jmon_config_name** is the project name.

You can manage the content of the JMon JAR file and `jmon-jar.xml` file by using the editor in Software AG Designer to update the `apama_java.xml` file, which is located in the project's `config` folder. You can use this editor to do the following:

- Set JMon metadata.
- Set the injection order of the events and monitors.
- Add non-JMon Java classes to the JMon JAR files.
- Add JMon classes that were not created by the Apama wizards in Software AG Designer to the JMon JAR file.

Java prerequisites for using Apama's JMon API

When you install Apama, the installation script installs the JMon API as `ap-correlator-extension-api.jar` in the Apama `lib` directory.

Software AG Designer includes the required Java compiler for running your application.

Steps for developing JMon applications manually

➤ To develop JMon applications outside Software AG Designer

1. Ensure that `ap-correlator-extension-api.jar` is in your Java CLASSPATH environment variable.
2. Create a folder in which to develop your application.
3. In this development folder, define one `.java` file for each event type and one `.java` file for each monitor class.
4. Ensure that there is a deployment descriptor file named `jmon-jar.xml`. See [“Creating deployment descriptor files” on page 497](#).
5. In your development folder, compile all your Java source code.

```
javac *.java
```

If `ap-correlator-extension-api.jar` is not already in your CLASSPATH environment variable, you can specify the `-classpath` command-line option to point to `ap-correlator-extension-api.jar`.

6. In your development folder, create a JAR file that contains the deployment descriptor and all class files. The command line format is as follows:

```
jar -cf application_name.jar META-INF/jmon-jar.xml *.class
```

Replace `application_name` with a name you choose for your application. On Windows, use backslashes (`\`) instead of forward slashes (`/`).

If your application uses an event type definition class that is also used by another JMon application, you must include the event type definition class in the JAR file of each application that uses it. If you do not include a shared event type definition class in your application's JAR file, injection fails with an `ApplicationVerificationException`.

You cannot specify the location of a shared event type definition class in your CLASSPATH environment variable. The correlator uses a separate classloader for each application, and it cannot use the system classloader for event type definition classes.

7. If any of your application's `.class` files are in your CLASSPATH environment variable, remove them. If the JRE can resolve a class path by using either your application's JAR file or your CLASSPATH environment variable, Apama fails to load your application.

Deploying JMon applications

➤ To deploy and run your application outside Software AG Designer

1. Start a correlator with Java enabled:

```
correlator -j other_options
```

2. Inject the application JAR file:

```
engine_inject -j application_name.jar
```

Apama creates an object instance of each monitor class defined in the deployment descriptor file and executes its `onLoad` method. If there are multiple monitor classes, they are injected in the order in which they are specified in the `jmon-jar.xml` file.

The classes in the application's JAR file cannot also exist (have the same packaging and name) anywhere else on the classpath. If they do, it causes the application to fail to load.

When you start the correlator, you can pass properties and options to the embedded JVM with the `-J` option. Specify the `-J` option with each property or option you want to specify.

For example, you can use this mechanism to specify a global classpath for the JVM with: `-J-Djava.class.path=path`. Apama prepends its own internal classpath `.jar` files to the path you specify. If you specify both the `CLASSPATH` environment variable and a classpath on the correlator start-up command line, the classpath specified on the command line takes precedence. See also [“Specifying the classpath in deployment descriptor files” on page 499](#) for information about specifying the classpath for each individual application.

Removing JMon applications from the correlator

To stop and delete a running JMon application, execute the `engine_delete` operation:

```
engine_delete [options_to_identify_correlator]application_name
```

If the application you want to delete is not running on the local host on the default correlator port, be sure to specify options that indicate the correlator that is running the application you want to delete.

Replace `application_name` with the name of the application as specified in the deployment descriptor. This is not necessarily the same as the name of the application's JAR file.

Deleting a JMon application does the following:

- Terminates the application's active listeners.
- Deletes the application's monitor classes.
- Leaves the event type definitions loaded in the correlator. To remove the event type definitions, execute `engine_delete` and specify the files that contain the event type definitions.

Creating deployment descriptor files

The JMon application's JAR file must contain a deployment descriptor file. Inside the correlator, the JVM processes the application's deployment descriptor file and uses it as a guide to the event

types and monitor classes to load. The name of the deployment descriptor file must be `jmon-jar.xml`.

When you use the Java support in Software AG Designer to develop your JMon application, the deployment descriptor file is generated for you. If you develop your JMon application outside Software AG Designer, there are two ways to create a deployment descriptor file:

- Manually write the deployment descriptor XML file. Use your favorite editor to create this XML file according to the [“Format for deployment descriptor files” on page 498](#).
- Insert Java annotations in your source files and run a utility to generate the deployment descriptor file. The annotations you can insert are defined in the `java.apama.jmon.annotation` package.

Of course, you can use the utility to generate the deployment descriptor file and then manually edit the result. If you then run the utility again, you would lose any manual changes you had made.

The technique you use is largely a matter of personal preference — hand-coded or machine-generated. If you have a very large application with many event types and monitors, you might prefer to insert the annotations and generate the deployment descriptor file. If you have a small application, you might find it easier to write the deployment descriptor file.

Format for deployment descriptor files

The format of the deployment descriptor file must be compliant with the XML defined by the following XML Document Type Definition (DTD):

```
http://www.apama.com/dtd/jmon-jar_1_2.dtd
```

You should become familiar with this DTD to understand the exact definition of the deployment descriptor file. However, the normal structure of the file is as follows. In the following format, all text inside XML element tags, *which is in italic typeface*, indicates placeholders for which you would supply an actual value.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jmon-jar PUBLIC "-//Apama, Inc.//DTD Java Monitors 1.2//EN"
  "http://www.apama.com/dtd/jmon-jar_1_2.dtd">

<jmon-jar>

  <name>Application Name in the Correlator</name>
  <version>Version Number</version>
  <author>Author</author>
  <company>Company Name</company>
  <description>Description of this application</description>
  <classpath>${sys:MY_THIRD_PARTY_DIR}/lib/foo.jar;
             ${sys:MY_THIRD_PARTY_DIR}/lib/bar.jar</classpath>
  <application-classes>

    <event>
      <event-name>Event Type name in the Correlator</event-name>
      <event-class>Event Type's class location</event-class>
      <description>Description of Event Type</description>
    </event>
```

```

<monitor>
  <monitor-name>Monitor's name in the Correlator</monitor-name>
  <monitor-class>Monitor's class location</monitor-class>
  <description>Description of Monitor class</description>
</monitor>
</application-classes>
</jmon-jar>

```

The most important part of the deployment descriptor file is the `application-classes` element. This element must contain an `event` element for each event type your JMon application defines. It must also contain a `monitor` element for each monitor your JMon application defines.

The application name that you specify in the `name` element is important because it defines the JMon application's name in the correlator. The `engine_inspect` management tool displays this name when it lists data for your application. If you want to delete your application, you specify this name. The application name must be unique across all currently loaded applications. If the application name is not unique, injection fails.

Specifying the classpath in deployment descriptor files

Each JMon or Java plug-in JAR is loaded in its own dedicated Java classloader, which by default has access only to its own classes, and those available globally in the correlator's system classloader.

Note: The correlator's system classloader includes some standard Apama libraries such as the `ap-correlator-extension-api.jar` and `ap-util.jar` JAR files plus any additional JAR files the user chooses to specify on the correlator command line using `-J-Djava.class.path=path`.

It is also possible to specify additional JAR files for use by a specific JMon application or Java plug-in, to provide access to any third-party libraries that the JAR requires. This approach is more self-contained than adding to the correlator's global classloader.

The classpath string for a JMon application or Java plug-in is specified in its deployment descriptor XML file as follows:

- If you are manually writing the deployment descriptor XML, add the optional `classpath` element just after the `description` element, for example:

```

...
<description>Description of this application</description>
<classpath>${sys:MY_THIRD_PARTY_DIR}/lib/foo.jar;
           ${sys:MY_THIRD_PARTY_DIR}/lib/bar.jar</classpath>
...

```

Note that the `classpath` element is only available in the 1.2 (and greater) versions of the JMon XML DTD (`jmon-jar_1_2.dtd`), so it may be necessary to update the `DOCTYPE` of the deployment descriptor to specify this DTD version if it does not already.

- If you are generating the deployment descriptor automatically using Java annotations, then use the optional `classpath` attribute in the `@Application` annotation:

```

@Application(
    name = "Simple",
    author = "My Name",

```

```
version = "1.0",
company = "Apama",
description = "My simple JMon application or Java plug-in",
classpath = "${sys:MY_THIRD_PARTY_DIR}/lib/foo.jar;
            ${sys:MY_THIRD_PARTY_DIR}/lib/bar.jar"
)
```

- If you are using Software AG Designer to generate the .jar and deployment descriptor, use the `@Application` annotation approach to specify the classpath.

In both cases, the classpath string consists of any number of classpath entries, delimited by semicolon characters (;). Note that the semicolon must be used even on platforms that typically use a colon or other character to separate path entries, and also that forward slashes (/) should be used instead of backslashes (\), in order to ensure that the application works in the same way regardless of the platform it is deployed on.

Avoid using absolute paths in the classpath, as this makes it difficult to use the application JAR on different machines. Instead, use `${...}` placeholders to identify the first part of each path, for example, the installation directory of a third party whose libraries you wish to use. Currently two types of placeholder are supported:

- `${sys:MY_SYS_PROP_NAME}` is replaced by a Java system property called `MY_SYS_PROP_NAME`
- `${env:MY_ENV_VAR_NAME}` is replaced by an environment variable called `MY_ENV_VAR_NAME`

The values for system property placeholders can be specified on the correlator command line using: `-J-DMY_SYS_PROP_NAME=path`.

The correlator will log a warning for any path that cannot be found, but will fail to inject the application entirely if the classpath includes any `${...}` placeholders that are not defined.

You can make the location of your project directory available by defining an environment variable in a YAML configuration file as described in "Setting environment variables for Apama components" in *Deploying and Managing Apama Applications*. You can use `${PARENT_DIR}` as the value of the property if the YAML file is located in your project directory root. If you do this, you can specify JAR files from your project directory. For example:

```
<classpath>
  ${env:APAMA_PROJECT_DIR}/lib/foo.jar;${env:APAMA_PROJECT_DIR}/lib/bar.jar
</classpath>
```

Defining event types in deployment descriptor files

The deployment descriptor file must define an event element for each event type class in your JMon application's JAR file. Each event element must contain the following two elements:

- **event-name** — The name by which this event type is to be defined within the correlator. The correlator has a single namespace. Consequently, this name must be unique across *all* applications. For example, `Tick` or `SimpleApp.Tick`. If you specify a package qualified name, it is the qualified name that must be unique.
- **event-class** — The name of the Java class in which this event type is defined. This must correspond to the fully qualified name of the class, for example, `Tick` if the event type class is

defined within the default Java package, or `com.apama.example.types.Tick` if the event type class is defined in the `com.apama.example.types` package. The file, for example, `Tick.java`, is expected to be located within a folder structure that maps to the packaging, as per standard Java convention.

The event element can optionally contain a third element. This is the `description` element. Specify a description of the event type. For example:

```
<event>
  <event-name>Tick</event-name>
  <event-class>Tick</event-class>
  <description>Event that signals a stock trade</description>
</event>
```

JMon and EPL share a single namespace for event types. After an event type is loaded into the correlator, using either JMon or EPL, it is available for use in either environment. However, within a JMon application, you cannot instantiate variables of an event type defined in EPL.

When you try to inject an event type definition that has the same name as a loaded event type, the correlator checks whether the two definitions are duplicates. If they are, the correlator ignores the duplicate you are trying to load. If the definitions are different, the correlator generates an injection error.

Defining monitor classes in deployment descriptor files

The deployment descriptor file must define a `monitor` element for each monitor class in your JMon application's JAR file. Each `monitor` element must contain the following two elements:

- `monitor-name` — The name by which this monitor is to be defined within the correlator. The correlator has a single namespace. Consequently, this name must be unique across *all* applications. For example, `SimpleMon` or `SimpleApp.SimpleMon`. If you specify a package qualified name, it is the qualified name that must be unique.
- `monitor-class` — The name of the Java class in which this monitor is defined. This must correspond to the fully qualified name of the class, for example, `SimpleMon` if the monitor class is defined within the default Java package, or `com.apama.example.monitors.SimpleMon` if the monitor class is defined in the `com.apama.example.monitors` package. The file, for example, `SimpleMon.java`, is expected to be located within a folder structure that maps to the packaging, as per standard Java convention.

The `monitor` element can optionally contain a third element. This is the `description` element. Specify a description of the monitor. For example:

```
<monitor>
  <monitor-name>Simple</monitor-name>
  <monitor-class>Simple</monitor-class>
  <description>A simple JMon monitor, used to show functionality of
    a new installation.</description>
</monitor>
```

Inserting annotations for deployment descriptor files

In your JMon source files, you can specify the following annotations:

- **@Application** — This annotation indicates the name of the application, as well as the author, version, company, and description of the application. Insert this annotation in any one, and only one, of your JMon source files. Each value is required. This annotation must be after any import statements and before the class definition statement. For example:

```
@Application(  
    name = "Simple",  
    author = "Moray Grieve",  
    version = "1.0",  
    company = "Apama",  
    description = "Deployment descriptor for a simple JMon monitor",  
    classpath = "${sys:MY_THIRD_PARTY_DIR}/lib/foo.jar;  
                ${sys:MY_THIRD_PARTY_DIR}/lib/bar.jar"  
)
```

- **@MonitorType** — This annotation indicates the definition of a monitor. In each monitor class, insert this annotation immediately before the monitor class definition statement. You can specify a name and a description for the monitor. The name is the fully qualified EPL name for the monitor. If you do not specify a name, the name defaults to the fully qualified JMon class name of the class you are annotating.

```
@MonitorType(description = "A simple JMon monitor, used to show  
    functionality of a new installation.")
```

- **@EventType** — This annotation indicates the definition of an event type. In each event type definition class, insert this annotation immediately before the definition statement for the event type. You can specify a name and a description for the event. The name is the fully qualified EPL name for the event. If you do not specify a name, the name defaults to the fully qualified JMon class name of the class you are annotating. For example:

```
@EventType(description = "Event that signals a stock trade")
```

- **@Wildcard** — This annotation indicates a wildcard event field. Insert it immediately before the field definition statement. You must have specified the **@EventType** annotation for the event type that defines this field. For example:

```
import com.apama.jmon.*  
import com.apama.jmon.annotation.*  
  
@EventType  
public class EventWithWildcard extends Event {  
    public long indexedField;  
    @Wildcard  
    public long wildcardField;  
    public EventWithWildcard() {  
        this(0, 0);  
    }  
    public EventWithWildcard(long iField, long wField) {  
        this.indexedField = iField;  
        this.wildcardField = wField;  
    }  
}
```

Sample source files with annotations

Following are two sample source files with annotations. These are the source files for the simple sample application provided with Apama. The lines with the annotations are in bold typeface for your convenience.

Here is the Simple.java file with comments removed:

```
import com.apama.jmon.*;
import com.apama.jmon.annotation.*;

@Application(name = "Simple",
    author = "Moray Grieve",
    version = "1.0",
    company = "Apama",
    description = "Deployment descriptor for the Simple JMon monitor",
    classpath = ""
)

@MonitorType(description = "A simple JMon monitor, used to show
    functionality of a new installation.")
public class Simple implements Monitor, MatchListener {

    public Simple() {}

    public void onLoad() {
        EventExpression eventExpr = new EventExpression(
            "all Tick(*, >10.0):t");
        eventExpr.addMatchListener(this);
    }
    public void match(MatchEvent event) {
        Tick tick = (Tick)event.getMatchingEvents().get("t");
        tick.emit();
    }
}
```

Here is the Tick.java file with comments removed:

```
import com.apama.jmon.Event;
import com.apama.jmon.annotation.*;

@EventType(description = "Event which signals a stock trade")
public class Tick extends Event {

    public String name;
    public double price;
    public Tick() {
        this("", 0);
    }

    public Tick(String name, double price){
        this.name = name;
        this.price = price;
    }
}
```


Generating deployment descriptor files from annotations

There are two utilities that you can use to generate the deployment descriptor file from annotations in your source files:

- `com.apama.jmon.annotation.DirectoryProcessor` — This utility processes a directory and generates the deployment descriptor file, which you must add to your application's JAR file.
- `com.apama.jmon.annotation.JarProcessor` — This utility processes an application's JAR file and adds the deployment descriptor file to that JAR file.

You can execute these utilities from the command line or from a Java build file.

The `DirectoryProcessor` utility takes three optional arguments:

- `-r` indicates that you want to recursively process the `.class` files in each directory and subdirectory in the specified directory. The default is that the utility processes only the `.class` files that are in the specified directory.
- `-d` specifies the directory that contains the `.class` files you want to process. The default is that the utility processes any `.class` files in the current working directory.
- `-o` specifies the file in which to store the output. The default is that output goes to `stdout`. In the JMon application JAR file, the name of the deployment descriptor file must always be `jmon-jar.xml`.

After you generate the deployment descriptor file, you must place it in the `META-INF` directory of your development directory. For example, you can execute the `DirectoryProcessor` utility from the command line as follows:

```
cd src
javac -classpath
$APAMA_CORRELATOR_HOME/lib/ap-correlator-extension-api.jar
*.java
java -DAPAMA_LOG_LEVEL=WARN -classpath
$APAMA_CORRELATOR_HOME/lib/ap-correlator-extension-api.jar
com.apama.jmon.annotation.DirectoryProcessor -r -d ./src -o
./src/META-INF/jmon-jar.xml
jar -cf ../simple-jmon.jar META-INF/jmon-jar.xml *.class
```

The `JarProcessor` utility takes one required argument, which is the name of the JAR file to operate on. To execute the `JarProcessor` utility from a Java build file, you can define something like the following:

```
<!--Target to process the annotations in the JMon application classes
to produce jmon-jar.xml -- the deployment descriptor file.
-->
<target name="process-jar" depends="jar">
  <echo message=
    "Process annotations in jar file: ${process-jar-file}" />
  <java jvm="java"
    classname="com.apama.jmon.annotation.JarProcessor" dir="."
    fork="yes">
    <classpath>
      <fileset dir="${lib-dir}">
```



```

        <patternset refid="libs" />
      </fileset>
    </classpath>
    <jvmarg value="-DAPAMA_LOG_LEVEL=WARN" />
    <arg value="${process-jar-file}" />
  </java>
</target>

<target name="process" depends="jar">
  <antcall target="process-jar">
    <param name="process-jar-file" value="${jar-file}" />
  </antcall>
</target>

```

Package names and namespaces in JMon applications

There is no correlation between the correlator namespace defined for a named JMon event or monitor, and the Java package structure of the class file in which that event or monitor is implemented. Event expressions are based on the correlator namespace, not on the Java package of the implementation.

Consider the following example. An event type defined in a Java class `a.b.c.MyEvent` that is given the correlator name `x.MyEvent`. Also a monitor defined in a Java class `a.b.c.MyListener` that is given the correlator name `y.MyListener`. Now, although the two classes are in the same Java package and need not use import statements to see each other, their correlator names are in different namespaces. This means that an event expression in `y.MyMonitor` will need to use the fully qualified name `x.MyEvent` to refer the event.

Sample JMon applications

The Apama distribution includes a number of complete sample applications. These applications are in the `samples/java_monitor` folder, and are called `simple`, `stockwatch`, `vwap`, `dos`, `context` and `complex`.

See the `README.txt` file included with each sample for complete instructions for how to compile and run the sample application.

III Developing EPL Plug-ins

18	Introduction to EPL Plug-ins	509
19	Providing an EPL Event Wrapper for a Plug-in	511
20	Writing EPL Plug-ins in C++	513
21	Writing EPL Plug-ins in Java	527
22	Writing EPL Plug-ins in Python	539

18 Introduction to EPL Plug-ins

Although the correlator's native programming language, the Apama Event Processing Language (EPL), has most of the functionality of modern programming languages, its primary purpose is enabling the detection of, correlation across, and triggering on complex event patterns.

In most cases, existing code could be ported and rewritten in EPL, but in practice this might not be feasible. For example, an application might need to carry out advanced arithmetic operations and a significant programming library of such functions might already be available. Porting such complex code to EPL would be a lengthy, expensive and error prone task, and is unnecessary.

The following topics describe Apama's EPL plug-in APIs and illustrate how to use them.

Note:

After upgrading to a new release of Apama, you always have to recompile your EPL plug-ins.

In order to incorporate existing specialized functionality, you can write EPL plug-ins in the following languages:

- **C++:** An EPL plug-in consists of an appropriately formatted library of C++ functions, which can be called from within EPL code.
- **Java:** The Java classes that are called from an application's EPL code are contained in a `.jar` file.
- **Python:** An EPL plug-in consists of a Python class with methods decorated to indicate they should be exported to EPL with the EPL action signature.

The correlator does not need to be modified to enable or to integrate with a plug-in, as the plug-in loading process is transparent and occurs dynamically when required.

Custom EPL plug-ins can be developed using Apama's EPL plug-in APIs for C++, Java, and Python. Once a plug-in is developed, you can call the functions it contains directly from the EPL code, passing EPL variables and literals as parameters, and getting return values that can be manipulated.

Note:

It is very important that strict plug-in development guidelines are followed when developing a plug-in. The functions provided must be adequately debugged prior to their integration within a plug-in. This is because when the correlator loads a plug-in, it is dynamically linked with the correlator's runtime process. If any code within the plug-in causes a runtime error, the correlator might fail and terminate.

For this reason, Apama customers who experience problems with correlator stability while using plug-ins will be asked by Apama Technical Support to remove the plug-in and reproduce the problem prior to being offered further technical help. Apama Technical Support will only lift this restriction if the plug-ins have had prior certification by Apama.

19 Providing an EPL Event Wrapper for a Plug-in

When creating a plug-in, it is considered best practice to provide an EPL event wrapper to access all methods of the plug-in. This provides type safety at runtime with respect to EPL objects of type chunk, that is, opaque objects whose contents cannot be seen or directly manipulated in EPL.

An example of this is the `TimeFormat` event which is provided as a wrapper for the Time Format plug-in (see also [“Using the TimeFormat Event Library” on page 341](#)). Using the plug-in directly, you can write code such as the following:

```
monitor UsePlugin {
    import "TimeFormatPlugin" as timeMgr;
    chunk pattern;

    action onload() {
        pattern := timeMgr.compilePattern("EEE MMM dd HH:mm:ss yyyy");
        float stateTimestampSec :=
            timeMgr.parseTimeFromPattern(pattern, "1996.07.10 AD at 15:08:56");
    }
}
```

Of course there is nothing to prevent someone passing a chunk from another plug-in as the parameter to the `parseTimeFromPattern` method. You can forestall this possibility and enforce type safety by using an event wrapper instead to hide the chunk type as in the following example:

```
using com.apama.correlator.timeformat.TimeFormat;
using com.apama.correlator.timeformat.CompiledPattern;

monitor UseEventWrapper {
    CompiledPattern pattern;

    action onload() {
        TimeFormat timeFormat := TimeFormat();
        pattern := timeFormat.compilePattern("EEE MMM dd HH:mm:ss yyyy");
        float stateTimestampSec := pattern.parseTime("1996.07.10 AD at 15:08:56");
    }
}
```

The event definitions for the `TimeFormat` and `CompiledPattern` events can be found in the `TimeFormatEvents.mon` file, which is located in the `monitors` directory of your Apama installation. Note how the `CompiledPattern` event wraps a chunk object, and the `parseTime` method on the `CompiledPattern` event uses the chunk in the `CompiledPattern` object and the string parameter passed in to the action.

This approach gives a more object-oriented feel to using the plug-in and can be used to emulate calling methods on C++ or Java objects. The signatures of actions on event definitions are also available to Apama in Software AG Designer, so they can be viewed there and benefit from completion proposals and type checking.

Note:

Software AG Designer does not know about the actions exposed by plug-ins, so it cannot provide type checking for them.

20 Writing EPL Plug-ins in C++

■ Creating a plug-in using C++	514
■ Using plug-ins written in C++	525

Creating a plug-in using C++

The APIs for writing plug-ins in C++ are all documented in the *API Reference for C++ (Doxygen)*. The relevant classes are in the `com::apama::epl::` namespace, provided in the `epl_plugin.hpp` header file.

To create a plug-in for EPL in C++, you have to create a class which inherits from `EPLPlugin`. `EPLPlugin` is a class template with one template parameter, which should be the derived class which implements your plug-in. Your class must provide a zero-argument constructor and a static `initialize` method which takes a `base_plugin_t::method_data_t` & argument. For example:

```
class MyPlugin: public EPLPlugin<MyPlugin>
{
public:
    MyPlugin(): base_plugin_t("MyPlugin")
    {}
    static void initialize(base_plugin_t::method_data_t &md)
    {
    }
};
```

`base_plugin_t` is a convenience typedef to the `EPLPlugin` base class. The base class constructor takes a single argument of descriptive string which is used for logging purposes. The base class provides two members to derived classes:

Member	Description
<code>logger</code>	A Logger object which can be used to write to the correlator log file. See also “Writing to the correlator log file” on page 519 .
<code>static getCorrelator()</code>	Returns a <code>CorrelatorInterface</code> & which can be used to make various callbacks into the correlator.

A single instance of your class is created when the plug-in is loaded. Functions which you want to expose to EPL are member functions on that instance. To export a function to EPL, you need to declare a member function on the class with argument and return types which can be translated into EPL (see [“Method signatures” on page 516](#)). The function is exported into EPL by calling `registerMethod` on the `method_data_t` passed to `initialize`:

```
static void initialize(base_plugin_t::method_data_t &md)
{
    md.registerMethod<decltype(&MyPlugin::doSomething),
        &MyPlugin::doSomething>("doSomething");
}
void doSomething(int64_t arg1, double arg2)
{
    // ...
}
```

`registerMethod` is templated over the following:

- the type of the function to export, which can be shortcut with `decltype()` as shown above, and

- the function to export.

`registerMethod` takes a mandatory argument, which is the name of the function in EPL, and two optional arguments.

- If the method signatures in EPL are not deducible from the C++ arguments, then you must provide the EPL type signature of the method.
- There is a final Boolean parameter, which defaults to `true`. It should be `true` if the plug-in may block (see [“Blocking behavior of plug-ins” on page 523](#)) and `false` otherwise.

```
md.registerMethod<decltype(&MyPlugin::simpleSignature),
    &MyPlugin::simpleSignature>("simpleMethod");
md.registerMethod<decltype(&MyPlugin::simpleSignatureNonBlocking),
    &MyPlugin::simpleSignatureNonBlocking>("nonBlockingMethod", false);
md.registerMethod<decltype(&MyPlugin::complexSignature),
    &MyPlugin::complexSignature>("complexMethod",
    "action<sequence<string>, integer> returns string");
```

Finally, to export the plug-in, you have to use the `APAMA_DECLARE_EPL_PLUGIN` macro. If your class is in a namespace, then you have to put the macro inside the same namespace and give it the class name without qualification as an argument:

```
APAMA_DECLARE_EPL_PLUGIN(MyClass)
```

Plug-ins are loaded into the correlator by using the `import` statement in the monitor or event which wants to use the plug-in. The argument is the name of the library which contains the plug-in.

```
import "MyPlugin" as plugin;
```

Complete simple example

This example implements a plug-in which simply keeps a single global count.

```
#include <epl_plugin.hpp>
using namespace com::apama::epl;

class CountPlugin: public EPLPlugin<MyPlugin>
{
public:
    CountPlugin()
        : base_plugin_t("CountPlugin"),
        count(0)
    {}
    static void initialize(base_plugin_t::method_data_t &md)
    {
        md.registerMethod<decltype(&CountPlugin::incrementCount),
            &CountPlugin::incrementCount>("increment");
        md.registerMethod<decltype(&CountPlugin::getCount),
            &CountPlugin::getCount>("getCount");
    }
    int64_t getCount() { return count; }
    void incrementCount(int64_t n) { count += n; }
    // Note that this is not thread-safe; a real plug-in would
    // want to use some synchronization here
private:
    int64_t count;
```

```
};
```

```
monitor foo
{
    import "PluginLibrary" as counter;
    action onload()
    {
        on all A() {
            counter.increment(1);
            print "Current count: "+counter.getCount().toString();
        }
    }
}
```

Method signatures

Only methods which have arguments and return values which can be converted into EPL types may be exported. For some of the types, the equivalent EPL signature can automatically be deduced. For methods which only take and return those types, you do not need to supply the explicit type. For other types, you must provide the EPL signature when registering the function.

EPL method signatures take the following forms:

```
action<integer>
action<sequence<string>, integer>
action<> returns integer
action<dictionary<string, string>, string> returns string
```

The following tables lists the EPL types and their equivalent C++ types.

Automatically deducible types

EPL type	C++ argument type	C++ return type	Notes
integer	int64_t	int64_t	
string	const char *	const char *	<p>You must <i>only</i> use const char * as a return type if it refers to a static string or a string whose lifetime is controlled outside of the function being returned from, such as one owned by a chunk.</p> <p>If you are constructing the string in the function, you must use std::string as the</p>

EPL type	C++ argument type	C++ return type	Notes
			return type of the function instead.
string	std::string	std::string	Use std::string as the return type for your function unless you are certain that the lifetime of the string will outlast the function returning it.
float	double	double	
boolean	bool	bool	
decimal	decimal_t	decimal_t	
chunk	const custom_t<T> &	custom_t<T>	Works for any class as the template parameter T.

Types requiring explicit signatures

EPL type	C++ argument type	C++ return type	Notes
context	int64_t	int64_t	int64_t will be deduced to integer, can be explicitly changed to context.
sequence<A>	const list_t &	list_t	Works for any sequence contents type.
dictionary<A, B>	const map_t &	map_t	Works for any dictionary contents type.
EventType	const map_t &	map_t	Works for any event type. The event is converted to a map_t where the key is of type string and the value is of type data_t. Keys are the names of the fields in the event.
any	const data_t &	data_t	Events are converted to a map_t (with the name

EPL type	C++ argument type	C++ return type	Notes
			set). See "C++ data types" in <i>Connecting Apama Applications to External Components</i> for further information.

Compiling C++ plug-ins

To compile a C++ plug-in, you need to do the following:

1. Add `#include <epl_plugin.hpp>` to the source file.
2. Put `$APAMA_HOME/include` on your compiler's include path.
3. Link the plug-in with `apclient.dll` (Windows) or `libapclient.so` (Linux).
4. Put `$APAMA_HOME/lib` on your compiler's linker path.
5. Enable C++11 standards mode.
6. Compile your class into a shared library `.dll` (Windows) or `.so` (Linux).

Using GCC on Linux this would look like this:

```
g++ --shared --std=c++0x -I$APAMA_HOME/include -L$APAMA_HOME/lib -lapclient -o
libMyPlugin.so MyPlugin.cpp
```

You can find example makefiles (Linux) and Visual Studio project files (Windows) in the `samples` directory of your Apama installation.

The C++ EPL plug-in API is implemented using a C-only ABI, even though it has a C++ API. This means that there is no requirement to compile your plug-in library with a specific compiler or compiler version. However, the compiler must support the C++ 11 syntax used in the API header files. We recommend use of the compilers specified in the *Supported Platforms* document for the current version, which is available from the following web page: <http://documentation.softwareag.com/apama/index.htm>.

If you are building a shared library to be used by multiple plug-ins and using the plug-in-specific data structures as part of your API between the library and the plug-ins, then you must ensure that the library and all of the plug-ins are compiled using the same version of the Apama header files. This means that if you upgrade Apama and want to recompile one of them, you must recompile all of them. You can choose not to recompile anything and they will still work.

If you compile with headers from multiple service packs of Apama, then you may see errors similar to the following when you try to link them.

■ Linux :

```
undefined reference to `Foo::test(com::softwareag::connectivity10_5_3::data_t const&)'
```

■ Windows:

```
testlib2.obj : error LNK2019: unresolved external symbol "public: void __cdecl
Foo::test(class com::softwareag::connectivity10_5_3::data_t const &)"
(?test@Foo@@QEAAEAEBVdata_t@connectivity10_5_3@softwareag@com@@@Z) referenced in
function "public: void __cdecl Bar::test(class
com::softwareag::connectivity10_5_3::data_t const &)"
(?test@Bar@@QEAAEAEBVdata_t@connectivity10_5_3@softwareag@com@@@Z)

testlib2.dll : fatal error LNK1120: 1 unresolved externals
```

If you encounter a similar error, try recompiling all your components with the same version of the headers.

If you are compiling a single plug-in, or multiple completely independent plug-ins, you can recompile them in any combination at any time.

Exceptions

Plug-ins can throw exceptions from plug-in methods. They must be derived from `std::exception` as normal.

If a plug-in throws an exception, this will be turned into an exception in EPL which may be caught with the EPL `try { } catch` syntax.

If you do not catch the exception, then the calling monitor instance will be terminated with the message in the exception thrown from the plug-in.

Writing to the correlator log file

The `EPLPlugin` base class provides a member called `logger` to plug-ins. This can be used to write messages to the correlator's log file.

Log messages are prefixed with the string `plugins.PluginName`, which is also the category that can be used to control the log level for this plug-in via the `correlatorLogging` section in the YAML configuration file (see "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications*).

```
void logMessage(const char *msg) {
    logger.info("Message is: %s", msg);
}
```

Documentation about the available functions and log levels can be found in the *API Reference for C++ (Doxygen)*.

Storing data using chunks

By themselves plug-ins can only store global state which must be protected by threading primitives from access from multiple EPL contexts simultaneously. It is possible for plug-ins to store state in an opaque (to EPL) object. This can be stored specific to a single monitor instance and then passed back into the plug-in for further processing. In EPL, this is represented using a chunk object. Chunk

objects are specific to a single plug-in, and an attempt to pass them to other plug-ins will throw an exception.

You can store any C++ type from your plug-in in a chunk, provided that it is copyable using the default copy constructor. If you have multiple different types from a single plug-in stored in EPL chunks, then you must have them share a class hierarchy so that you can distinguish them when they are passed back from EPL.

The argument and return type that you must use to pass chunks is `custom_t<T>`. `custom_t` is templated over the type that you are storing in the chunk. Chunks can only be created by your plug-in and returned from a function to EPL. When they are passed back to your plug-in, you can modify the referred to class, but not change which object it points to. Chunk objects are garbage collected by the EPL runtime for you and will be deleted when they go out of scope. You do not retain ownership of them.

If you need lifetime shared between EPL and the plug-in (or another chunk object), then indirectly access the shared object via `std::shared_ptr`. The `custom_t` would then point to a `std::shared_ptr` or a class containing a `std::shared_ptr`, and the plug-in follows the `shared_ptr` to access the shared object. The plug-in can also have a `std::shared_ptr` to the shared object. The object will not be destroyed until both EPL has garbage collected the chunk object and the plug-in has destroyed its `shared_ptr` to the object.

```
class EPLData
{
public:
    EPLData(int64_t value): value(value) {}
    // default copy constructor and destructor
    int64_t value;
};

custom_t<EPLData> createChunk(int64_t value)
{
    return custom_t<EPLData>(new EPLData(value));
}

void logChunkValue(const custom_t<EPLData> &chunk)
{
    logger.info("Chunk value is: %" PRIu64, chunk->value);
}
```

```
monitor m {
    import "DataPlugin" as plugin;
    action onload() {
        chunk c := plugin.createChunk(42);
        plugin.logChunkValue(c);
    }
}
```

Chunk objects can be copied in EPL. This occurs when the explicit `clone()` method is called on a chunk or an event containing the chunk, or if the monitor spawns. When a chunk object is copied, the value in the `custom_t` is copied, using the default copy constructor of the template parameter for `custom_t`. Thus, each instance will only be accessible from a single monitor instance and thus only from one thread at a time (though copies may be accessed concurrently to the original object).

If you have multiple different chunk types, then use the base class as the template parameter for `custom_t` and then use `dynamic_cast` or virtual dispatch to distinguish between them. When returning a `custom_t` value, you must use the derived type as the template parameter, because that is the type used to create a copy of the object, if needed.

```
class ChunkBase
{
    virtual void doSomething() = 0;
};
class DerivedChunk: public ChunkBase
{
};
void modifyChunk(const custom_t<ChunkBase> &chunk)
{
    chunk->doSomething(); // virtual dispatch
    auto derived = dynamic_cast<DerivedChunk*>(chunk.get());
    if (derived) {
        // derived-specific methods
    }
};
```

Sending events

Method calls on plug-ins are synchronous and generally should be written not to take too long or hold up processing in the correlator (see [“Blocking behavior of plug-ins” on page 523](#)). One technique to enable asynchronous behavior in the plug-in is to interact with EPL by sending events which can be handled asynchronously, potentially from a background thread which is processing events as well as from methods themselves.

The `CorrelatorInterface` returned from `getCorrelator()` contains several methods for sending events into the correlator. You can send events either as the string representation of the event in Apama's internal string format (for example, `MyEvent(1.3, true)`) or as a `map_t` type where the keys are strings corresponding to the field names in the event, and the values are the values for those fields, in the appropriate type/format for the type of the field. In the latter case, you also need to supply the name of the event type to parse the map as.

You can select the destination of the event in several ways:

- **Named channel.** The preferred method is to deliver the event to a specific named channel. This will go to everything which has subscribed to that named channel. Subscribers can either be context, external receivers, connectivity chains or other plug-ins which are subscribed to receive events on that channel (see [“Receiving events” on page 522](#)).

Send as string:

```
getCorrelator().sendEventTo("MyEvent(42)", "channelName");
```

Send as object:

```
map_t event;
event.insert(data_t("number"), data_t(int64_t(42)));

getCorrelator().sendEventTo("MyEvent", std::move(event), "channelName");
```

- **Context by ID.** You can deliver the event to a specific context by context ID. Context IDs can either be passed into a plug-in from an EPL `context()` object, or they can be obtained with the `CorrelatorInterface.getCurrentContextId()` method.

Send as string:

```
getCorrelator().sendEventTo("MyEvent(42)", ctxId);
```

Send as object:

```
map_t event;
event.insert(data_t("number"), data_t(int64_t(42)));

getCorrelator().sendEventTo("MyEvent", std::move(event), ctxId);
```

You can send events from within a method called from EPL, from any callback handler method, or from threads spawned by the plug-in itself.

Receiving events

Plug-ins can subscribe to receive events which are delivered to certain named channels. They will receive events sent to those channels from EPL, from external senders, from connectivity chains or from other plug-ins (see [“Sending events” on page 521](#)). To subscribe to events, you need to subclass the `EventHandler` class and implement the `handleEvent()` virtual method. Then you need to call `registerEventHandler` on the `CorrelatorInterface` returned by `getCorrelator()` with an instance of your `EventHandler`. `registerEventHandler` takes ownership of the handler object.

```
class MyHandler: public EventHandler
{
    virtual void handleEvent(const char *type, data_t &&event,
        const char *channel)
    {
        // do something with event
    }
};

MyPlugin(): base_plugin_t("MyPlugin") {
    EventHandler::subscription_t handler = getCorrelator().registerEventHandler(
        MyHandler::ptr_t(new MyHandler()), "channelName",
        /*mode=*/MAP_MODE, /*blocking=*/true);
}
```

registerEventHandler() parameters

`registerEventHandler()` has mandatory and optional parameters:

Parameter	Description
<i>handler</i>	Required. The handler object which will process delivery of the events.
<i>channel</i>	Required. The initial channel to subscribe to.

Parameter	Description
<code>mode</code>	Optional. If set to <code>STRING_MODE</code> , then events are delivered in Apama string form. If set to <code>MAP_MODE</code> (default), events are delivered in <code>map_t</code> form (see “Sending events” on page 521).
<code>blocking</code>	If set to <code>true</code> , then this handler may block the thread. If set to <code>false</code> , the handler will never block (see “Blocking behavior of plug-ins” on page 523).

handleEvent() parameters

`handleEvent()` is called with the following parameters:

Parameter	Description
<code>type</code>	The name of the type of the event.
<code>event</code>	The event itself. The <code>data_t</code> contains a <code>const char *</code> if the handler was subscribed in <code>STRING_MODE</code> or a <code>map_t</code> if it was subscribed in <code>MAP_MODE</code> . You are given ownership of the event so you can move it for further processing without taking a copy.
<code>channel</code>	The channel on which this event was delivered.

EventHandler::subscription_t methods

`registerEventHandler()` returns an object of type `EventHandler::subscription_t`, which can be used to add channels to a subscription, remove channels from a subscription, or remove the handler entirely. You must not call either of the remove methods from inside the handler object.

`EventHandler::subscription_t` has the following methods:

Method	Description
<code>void addChannel(const char *channel)</code>	Subscribe to an additional channel.
<code>bool removeChannel(const char *channel)</code>	Unsubscribe from the channel, if subscribed to it. If this reduces the subscription count to 0, then it destroys the handler. Returns <code>true</code> if the handler was destroyed.
<code>void removeAllChannels()</code>	Remove all subscriptions and destroy the handler.

Blocking behavior of plug-ins

The correlator system assumes that all functions return in a reasonable amount of time, and do not do potentially blocking operations such as file-system operations or remote calls. For code written in EPL, the correlator can enforce this. For code provided in plug-ins, the correlator cannot

enforce this or know whether the method may block. Therefore, by default, the correlator assumes that any plug-in method may block indefinitely. This may cause the correlator to create new operating system threads to service incoming events on other contexts.

If you know that your plug-in does not do anything long-running or potentially blocking, then you may declare the method as “non-blocking” at the point it is registered in the `initialize` function:

```
static void initialize(base_plugin_t::method_data_t &md)
{
    md.registerMethod<decltype(&MyPlugin::nonBlockingMethod),
        &MyPlugin::nonBlockingMethod>("methodName", /*blocking=*/false);
}
```

For event handlers, this is done with the call to `registerEventHandler` (see [“Receiving events” on page 522](#)).

If you do this, then the correlator assumes that the method will return soon and not spawn additional threads. This can avoid extra overhead of starting and stopping operating system threads.

CAUTION:

Declaring a method as non-blocking when it actually blocks can cause poor performance or even deadlock the correlator entirely.

If you have a method which is normally non-blocking, but may sometimes block, you can declare it as non-blocking initially and then, when it encounters a condition which is blocking, notify the correlator that this call is blocked to allow the correlator to spawn additional threads.

```
int64_t get(int64_t key) {
    if (local(key)) {
        return local(key);
    } else {
        getCorrelator().pluginMethodBlocking();
        return remote(key);
    }
}
```

Load-time or unload-time code

If you need to execute code at the time the plug-in is loaded, then you should put it in the plug-in class constructor.

If you need to execute code at the time the plug-in is unloaded, then you should put it in the plug-in class destructor.

Typically, you will register any callbacks at load time in the constructor.

```
class MyPlugin: public EPLPlugin<MyPlugin>
{
    MyPlugin(): base_plugin_t("MyPlugin") {
        // load-time code here
    }
    ~MyPlugin() {
        // unload-time code here
    }
}
```

```
    }
};
```

Handling thread-specific data in plug-ins

The correlator executes plug-ins on a number of internal threads which may change over time, including spawning new threads or destroying old threads. If a plug-in needs to store any state in thread-local variables, then - as well as cleaning up this state when the plug-in unloads - it also needs to clean it up when a thread is destroyed. To enable this, plug-ins can register to receive a callback whenever the correlator destroys a thread. This is done by subclassing `ThreadEndedHandler` with an implementation of `threadEnded()` and then passing an instance of your subclass to the `receiveThreadEndedCallbacks()` method on `CorrelatorInterface`. `receiveThreadEndedCallbacks` takes ownership of the handler and may only be called once per plug-in.

```
class MyThreadHandler: public ThreadEndedHandler {
public:
    virtual void threadEnded() {
        // cleanup code here
    }
};
MyPlugin(): base_plugin_t("MyPlugin")
{
    getCorrelator().receiveThreadEndedCallbacks(
        MyThreadHandler::ptr_t(new MyThreadHandler()));
}
```

Using plug-ins written in C++

The plug-in library you compiled must be available on the correlator's `PATH` (Windows) or `LD_LIBRARY_PATH` (Linux). By default, the `$APAMA_WORK/lib` directory is included on the appropriate path, and we recommend that you use this location for your own plug-ins.

Plug-ins are loaded using the `import` statement in EPL:

```
monitor m {
    import "MyPlugin" as plugin;
}
```

This statement can be made at the top-level of either monitors or events.

The correlator then attempts to load `MyPlugin.dll` (Windows) or `libMyPlugin.so` (Linux) from the path. The correlator reads the list of exported methods from the plug-in. These methods are then available on the name provided in the `as` statement in the rest of that monitor or event:

```
event E {
    import "MyPlugin" as plugin;
    action increment(integer i) {
        plugin.increment(i);
    }
    action get() returns integer {
        return plugin.getCount();
    }
}
```

Note:

When compiling with the GNU compiler collection (g++), it is not possible to completely unload plug-ins when GNU UNIQUE symbols are present. These are created transparently by the compiler. If you try and unload a plug-in via `engine_delete` and you see the warning telling you that the unloading of *plug-in_name* failed and that the DSO probably contains GNU UNIQUE symbols, then you are affected by this issue. If you need to do hot redeployment of your plug-in code (without restarting the correlator), then you will have to rename your plug-in and change the references to it in order to reload it into the same correlator.

21 Writing EPL Plug-ins in Java

■ Creating a plug-in using Java	528
■ Using EPL plug-ins written in Java	530
■ Sample plug-ins in Java	536

EPL plug-ins can be written in Java. Java plug-in classes are automatically analyzed by the correlator and any suitable methods exposed as methods that can be called from EPL.

EPL plug-ins written in Java are packaged and deployed in the same way as JMon applications. See [“Developing and Deploying JMon Applications” on page 493](#) for more information.

Creating a plug-in using Java

➤ To create a Java class to use as an EPL plug-in

1. In the Java class used as a plug-in, you need to have one or more public static methods that match the permitted signatures, which are described in [“Permitted signatures for methods” on page 529](#).

All calls from an Apama application will be made to these static methods from all contexts.

As the plug-in author you are responsible for any concurrency concerns.

2. EPL plug-ins in Java are deployed using a JMon application and are packaged in a .jar file. You need to create a JMon deployment descriptor file in the application's META-INF/jmon-jar.xml file. For the plug-in, you need to add a <plugin> to the <application-classes> element.

For more information on Apama deployment descriptor files, see [“Creating deployment descriptor files” on page 497](#).

An example plug-in stanza looks like this:

```
<plugin>
  <plugin-name>TestPlugin</plugin-name>
  <plugin-class>test.TestPlugin</plugin-class>
  <description>A test plugin</description>
</plugin>
```

- plugin-name defines the name visible to EPL.
- plugin-class indicates the class to load from the jar for this plug-in.
- description is a simple textual description that appears in log messages.

Instead of writing a deployment descriptor file manually, if you are using Software AG Designer to create the plug-in, you can annotate the plug-in class and have Software AG Designer automatically generate the descriptor file. Here is an example annotation:

```
@com.apama.epl.plugin.annotation.EPLPlugin(name="TestPlugin",
                                           description="A test plugin")
class testplugin
{
    ...
}
```


3. Create a `.jar` file for deploying the plug-in and add the Java class file and the deployment descriptor file `META-INF/jmon-jar.xml` to it. In Software AG Designer when you create a JMon application, this is done automatically.

For applications that you plan to inject into a correlator, the recommendation is to create separate `.jar` files for:

- EPL plug-ins written in Java
- JMon applications

Although the mechanism for creating these jars and describing their meta-data is similar, the interactions of these two different uses of injected jars mean that they will often need to be injected into the correlator separately. The creation of separate `.jar` files ensures that you can inject your application components in the correct order, which is typically:

1. EPL plug-ins written in Java
2. EPL monitors and events
3. JMon applications

Permitted signatures for methods

For a method to be exposed to EPL, it must be public, must be static and every argument plus the return type must be one of the following:

Java Entry	EPL Type	Notes
<code>int</code>	<code>integer</code>	Truncated when passed in, for compatibility.
<code>long</code>	<code>integer</code>	
<code>String</code>	<code>string</code>	Copy in / copy out.
<code>boolean</code>	<code>boolean</code>	
<code>double</code>	<code>float</code>	
<code>java.lang.Number</code>	<code>decimal</code>	Is either <code>java.lang.Double</code> type for NaN or infinity, or <code>java.math.BigDecimal</code> for a value.
<code>java.math.BigDecimal</code>	<code>decimal</code>	Passing in either NaN or infinity throws an exception that kills the monitor instance if not caught. Deprecated, use <code>java.lang.Number</code> instead.

Java Entry	EPL Type	Notes
<code>com.apama.epl.plugin.Context</code>	<code>context</code>	New type defined for plug-ins.
<code>com.apama.epl.plugin.Channel</code>	<code>com.apama.Channel</code>	New type defined for plug-ins.
<code>Object</code>	<code>chunk</code>	Any Java object can be held in EPL via a chunk.
<code>TYPE[]</code>	<code>sequence<TYPE></code>	Any above type except <code>int</code> can be passed in as an arbitrary-depth nested array->sequence. The sequence is strictly copy-in, non-modifiable, but can be returned as copy-out.
<code>void</code>	N/A	Permitted as a return type only.

Any method not matching this signature is ignored and logged at `DEBUG`.

Overloaded functions

Any function with multiple overloads is ignored (none of them are exposed) and this is logged once at `WARN` and once per method at `DEBUG`.

Using EPL plug-ins written in Java

After you create an EPL plug-in in Java, it must be injected into a Java-enabled correlator before it is available for use in Apama applications. Applications that will use the plug-in also need to import the plug-in by name.

Injecting

The `.jar` file containing the EPL plug-in must be injected into a correlator that has been started with the `--java` option, which enables support for JMon applications. When using the Apama `engine_inject` utility to inject the `.jar` file, you also need to use the `--java` option.

Importing

Once a Java plug-in has been injected it is available for import using the `plugin-name` defined in the deployment descriptor file. The correlator will automatically introspect the class and make available any suitable, public methods that can be called directly from EPL. For example, the following code imports a plug-in named `TestPlugin` and calls its `dosomething` method:

```
monitor m {
```

```
import "TestPlugin" as test;
action onload()
{
    test.dosomething();
}
}
```

Note, if the plug-in .jar has been incorrectly injected, the correlator will try to load the plug-in as a C++ plug-in and may give an error such as `Error opening plug-in library libfoo.so: libfoo.so: cannot open shared object file: No such file or directory`. If this happens and you were trying to load a plug-in written in Java, then check that the .jar file was created and injected correctly before your EPL file was injected.

Classpath

Each JMon or Java plug-in application is loaded into its own separate classloader. This means that they have no access to any classes loaded in other .jar files. If your plug-in requires any other Java libraries they must be listed in the `classpath` element of the deployment descriptor, included in the correlator's global classpath, or injected in the same application .jar as the plug-in. See [“Specifying the classpath in deployment descriptor files” on page 499](#) for more details.

Deleting

An EPL plug-in can be explicitly deleted by calling `engine_delete` with the application name defined in the deployment descriptor, as with JMon applications. Monitors using the plug-in depend on the plug-in type in the normal fashion. The plug-in will not be deleted until the application and all dependent monitors are deleted.

As each plug-in is loaded in its own classloader, once the application has been deleted, the plug-in can be re-injected and it will be loaded into a new classloader.

Interacting with contexts

EPL plug-ins can be passed context objects using the `com.apama.epl.plugin.Context` type. The `Context` object is defined as:

```
package com.apama.epl.plugin;
public class Context
{
    public String toString();
    public Context();
    public String getName();
    public int hashCode();
    public boolean isPublic();
    public boolean equals(Context other);
    public static native Context getCurrent();
}
```

The `getCurrent` method returns the context that this method was called from.

Interacting with the correlator

EPL plug-ins can use the `com.apama.epl.plugin.Correlator` class to send an event, subscribe to a channel, or to specify blocking behavior. The `Correlator` class is defined as:

```
package com.apama.epl.plugin;
public class Correlator
{
    public static native void sendTo(String evt, String chan);
    public static native void sendTo(String evt, Context ctx);
    public static native void sendTo(String evt, Context[] ctxs);
    public static native void sendTo(String evt, Channel c);

    public static native void subscribe(EventHandler handler, String[] channels);
    public static native void unsubscribe(EventHandler handler, String[] channels);
    public static native void unsubscribe(EventHandler handler);

    public static native void pluginMethodBlocking();
}
```

The `Correlator` methods are:

- `sendTo(String, String)` – Sends the event represented in the first `String` to the channel specified in the second `String`. Any contexts and external receivers that are subscribed to the specified channel receive the event. If there are no subscribers the event is discarded.
- `sendTo(String, Context)` – Sends the event represented in `String` to the context referred to by the `com.apama.epl.plugin.Context` argument. An exception is thrown if the context reference is invalid.
- `sendTo(String, Context[])` – Sends the event represented in `String` to the array of contexts referred to by the `com.apama.epl.plugin.Context[]` argument. If one context reference is invalid an exception is thrown and the event is not sent to any context.
- `sendTo(String, Channel)` – Sends the event represented in `String`. If the specified `com.apama.epl.plugin.Channel` object contains a string then the event is sent to the channel that has that name. If `Channel` contains a context then the event is sent to that context.
- `subscribe(EventHandler, String[])` – Subscribes the handler object to the channels listed in the string array. If the handler is already subscribed to some channels then the channels listed in the array are added to the list of existing subscriptions. Subscribing to the same channel multiple times results in a single subscription. However, to completely remove a channel subscription that has been added multiple times you must unsubscribe from that channel the same number of times that it was subscribed to.
- `unsubscribe(EventHandler, String[])` – For the channels specified in the string array, this method removes the subscriptions from the specified handler. It is possible for the result of this method to be that the handler is not subscribed to any channels. Unsubscription from a channel that the handler is not subscribed is ignored.
- `unsubscribe(EventHandler)` – Removes all subscriptions from the specified handler. If this handler is not subscribed to any channels the method is ignored.

- `pluginMethodBlocking()` – Informs the correlator that the plug-in is potentially blocking for the rest of this call and the correlator is free to spin up additional threads on which to run other contexts.

For more information on `com.apama.epl.plugin.Context` and `com.apama.epl.plugin.Correlator`, see the *API Reference for Java (Javadoc)*.

Receiving events from named channels

A Java plug-in can register callbacks to receive events that are sent to named channels. This is similar to the `monitor.subscribe()` method in EPL. Events are delivered in string form by means of a method on a known interface.

To register a callback, the plug-in must define a class that implements the `com.apama.epl.plugin.EventHandler` interface:

```
public interface EventHandler
{
    void handleEvent(String event, String channel);
}
```

The `handleEvent()` method is called once for each event sent to a channel that this handler is subscribed to, with the channel on which it was received. To manage `EventHandler` object channel subscriptions, use the `subscribe()` and `unsubscribe()` methods on `com.apama.epl.plugin.Correlator`. When a handler is unsubscribed from all channels any in-progress callbacks will complete, but no further callbacks will be made to that handler.

Working with Channel objects

Similar to context objects, you can pass EPL `com.apama.Channel` objects into a Java plug-in. The equivalent Java class is `com.apama.epl.plugin.Channel` and you can use objects of this class to send events to channels. Like the EPL `Channel` type, the Java `Channel` class has three constructors:

```
Channel (String name)
Channel (com.apama.epl.plugin.Context c)
Channel ()
```

A `Channel` object can contain a string that is the name of a channel or it can contain a context. The no-argument constructor creates a `Channel` object that contains an empty context. If you try to send an event to an empty context the `sendTo()` method throws an exception.

You can call the `empty()` method on a Java `Channel` object. It returns `true` only if the object contains an empty context.

Exceptions

If a method throws an exception, that exception is passed up to the calling EPL and can be caught by the calling monitor. If an exception is not caught it will terminate the monitor instance. Details on catching exceptions in EPL can be found in [“Exception handling” on page 276](#).

If a Java plug-in throws a `java.lang.RuntimeException`, or subclass, which is in the `java.` namespace (for example, `java.lang.NullPointerException`) then it will be logged at `ERROR` with a stacktrace

before being rethrown. Unchecked exceptions from other sources (for example client exception types) will not be logged.

Load, unload, and shutdown hooks

If a plug-in needs to run anything when it is loaded, you can do this in a static initializer:

```
public class Plugin
{
    static {
        ... // initialization code here
    }
}
```

It is not natively possible for a plug-in to run anything when it is unloaded. If you need this functionality you can declare a method to be called when the plug-in is unloaded using annotations:

```
public class Plugin
{
    @com.apama.epl.plugin.annotation.Callback(
        type=com.apama.epl.plugin.annotation.Callback.CBType.SHUTDOWN)
    public static void shutdown()
    {
        ... // shutdown code here
    }
}
```

The method must be a public static function which takes no arguments and returns void. Currently, Apama does not support callbacks other than SHUTDOWN.

Non-blocking plug-ins and methods

In a correlator some threads have the potential to block and others do not. If a thread might block, the correlator starts new threads if it has additional runnable contexts. By default the correlator assumes that a plug-in call may block and will start additional threads on which to run other contexts. In situations where the plug-in call can never block, the additional overhead of starting new threads when all CPUs are busy is unnecessary. If you know that a plug-in or an individual method is non-blocking, you can improve efficiency by annotating either entire plug-ins or individual methods as non-blocking.

Note, however, if a method declared as non-blocking does block, the correlator can block all threads waiting for them to finish, resulting in a deadlocked correlator. For methods that are normally non-blocking, but may block in predictable situations, see "Sometimes-blocking functions" below.

- **Annotations.** You can apply the annotation `com.apama.epl.plugin.annotation.NoBlock` with no arguments to either a plug-in class, or to a method on a class:

```
@com.apama.epl.plugin.annotation.NoBlock()
public class Plugin
{
    ...
}
```

When applied to a class, the annotation indicates that no method on the plug-in can ever block.

```
public class Plugin
{
    @com.apama.epl.plugin.annotation.NoBlock()
    public static String getValue() { ... }
}
```

When applied to a method, the annotation indicates that this method will never block, but other methods may block.

- **Sometimes-blocking functions.** If you have a function that usually will not block, but under some known conditions may block, then the method can be declared as `NoBlock` as long as it then uses a callback to indicate when it is starting the potentially-blocking behavior. The callback is a static method on `com.apama.epl.plugin.Correlator` called `pluginMethodBlocking`. This function takes no arguments, returns no value and is idempotent. When it is called, the correlator will then assume that the plug-in is potentially blocking for the rest of this call and is free to spin up additional threads on which to run other contexts.

```
public class Plugin
{
    @com.apama.epl.plugin.annotation.NoBlock()
    public static String getValue()
    {
        if (null != localValue) return localValue;
        else {
            com.apama.epl.plugin.Correlator.pluginMethodBlocking();
            localValue = getRemoteValue();
            return localValue;
        }
    }
}
```

Logging

EPL plug-ins written in Java can log to the correlator's log file. This is done via the `com.apama.util.Logger` class, or alternatively using the open-source SLF4J API. The SLF4J API is provided in the `/common/lib/ext` directory of your Software AG installation.

Each plug-in must create a static instance of `com.apama.util.Logger` using the static `getLogger` method. This instance provides `debug(...)`, `info(...)`, `warn(...)` and `error(...)` methods, which log a string at that log level in the correlator log file. See also "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications*.

For more information on using the `Logger` class, including how to override the default log level, see the *API Reference for Java (Javadoc)*.

The following is an example of logging in an EPL plug-in:

```
package test;
import com.apama.util.Logger;
public class MyPlugin
{
    private static final Logger logger = Logger.getLogger("plugins.test.MyPlugin");
    public static void foo()
    {
        logger.info("A string that's logged at INFO");
    }
}
```

```
}
```

This will produce entries in the correlator log file like this:

```
2019-06-24 14:22:21.974 INFO [1167792448:processing] - <test.MyPlugin> A string that's
logged at INFO
```

It is recommended to put “plugins.” at the beginning of the category name (as shown in the above example).

If your plug-in uses a third-party library that logs with SLF4J or Log4j 2, then the log output goes to the main correlator log file automatically. The root classloader also contains the Log4j 1.x bridge (log4j-1.2-api.jar) which redirects calls to most of the Log4j 1 API through to Log4j 2. So provided you do not use internal features such as programmatic configuration, Log4j 1 libraries should work fine. However, we recommend that you transition away from Log4j 1 since it has reached end-of-life status.

When using a library which uses some other logging implementation, such as the JDK logger, or Apache Java commons logging (JCL), then add a bridging jar to convert it to SLF4J where possible. Several bridges are available in the `common/lib/ext` and `common/lib/ext/log4j` directories of your Software AG installation.

Sample plug-ins in Java

Apama provides sample EPL plug-ins written in Java, located in the `samples\correlator_plugin\java` directory of your Apama installation. The samples are:

- `SimplePlugin` – a basic plug-in with one method that takes a string, and returns another string.
- `ComplexPlugin` – a plug-in that has several methods and handles more complex types.
- `SendPlugin` – a plug-in that demonstrates passing contexts around and sending events.
- `SubscribePlugin` – a plug-in that shows how to subscribe to receive events sent on a particular channel.

The `samples\correlator_plugin\java` directory contains the Java code for the samples, the EPL code for the Apama applications that call each of the plug-ins, the deployment descriptor files, and an Ant `build.xml` file for building all of the samples. The directory also contains a `README.txt` that describes how to build and run the samples as well as text files that depict what the output of the samples should be like.

A simple plug-in in Java

The simple plug-in sample can be found in the `samples\correlator_plugin\java` directory of your Apama installation.

The Java code for the `SimplePlugin` class contains the public static `test` method. (Methods that will be called from EPL code need to be public and static.)

```
public class SimplePlugin
{
```



```

public static final String TEST_STRING = "Hello, World";
public static String test(String arg)
{
    System.out.println("SimplePlugin function test called");
    System.out.println("arg = "+arg);
    System.out.println("return value = "+TEST_STRING);
    return TEST_STRING;
}
}

```

The `SimplePlugin.xml` file is the deployment descriptor and contains the following `<plugin>` stanza that illustrates how to specify the plug-in.

```

<application-classes>
  <plugin>
    <plugin-name>SimplePlugin</plugin-name>
    <plugin-class>SimplePlugin</plugin-class>
    <description>A test plugin</description>
  </plugin>
</application-classes>

```

The `SimplePlugin.mon` file contains the EPL code. It imports the plug-in and calls the test method.

```

monitor SimplePluginTest {
    // Load the plugin
    import "SimplePlugin" as simple;
    // To hold the return value
    string ret;
    string arg;
    action onload() {
        // Call plugin function
        arg := "Hello, Simple Plugin";
        ret := simple.test(arg);
        // Print out return value
        log "simple.test = " + ret at INFO;
        log "arg = " + arg at INFO;
    }
}

```

A more complex plug-in in Java

The complex plug-in sample can be found in the `samples\correlator_plugin\java` directory of your Apama installation.

The Java code for the `ComplexPlugin` class contains the public static methods: `test1`, `test2`, `test3`, and `test4`. It also contains an object, `ComplexChunk` that represents a complex type.

The `complex_plugin.xml` file is the plug-in's deployment descriptor and contains the `<plugin>` stanza that specifies the name, class, and description for the plug-in.

The sample's `ComplexPlugin.mon` file contains the EPL code for the Apama application. It imports the plug-in and calls the various `testx` methods.

A plug-in in Java that sends events

The `SendPlugin.java` file in the `samples\correlator_plugin\java` directory of your Apama installation is a sample plug-in that shows how to pass contexts around and how to send events to specific contexts.

The Java class for the plug-in imports `com.apama.epl.plugin.Context` and `com.apama.epl.plugin.Correlator` and it declares a public method that sends an event to a channel and another public method that sends an event to a particular context.

The `SendPlugin.xml` deployment descriptor file contains the name, class, and description of the plug-in in the `<plugin>` stanza.

The Apama application `SendPlugin.mon` first imports the plug-in and then calls the plug-in's `sendEventToChannel()` method as well as its `sendEventTo()` method with a variety of contexts.

A plug-in in Java that subscribes to receive events

The `SubscribePlugin.java` file in the `samples\correlator_plugin\java` directory of your Apama installation is a sample that shows how a plug-in subscribes to receive events sent on a particular channel.

The Java code for the `SubscribePlugin` class contains the public static `createHandler` method. (Methods that will be called from EPL code need to be public and static.)

The deployment descriptor file `SubscribePlugin.xml` contains the `<plugin>` stanza that illustrates how to specify the plug-in.

The EPL code in the file `SubscribePlugin.mon` imports the plug-in and calls the `createHandler()` method.

22 Writing EPL Plug-ins in Python

■ Creating a plug-in using Python	540
■ Using Python plug-ins	544
■ Installing Python modules	546
■ Sample plug-ins written in Python	547

Creating a plug-in using Python

The APIs for writing plug-ins in Python are documented in the *API Reference for Python*. The relevant classes are in the `apama.eplplugin` module.

Note:

EPL plug-ins written in Python support Python 3, which is shipped with Apama. They do not support Python 2.

To create a plug-in for EPL in Python, you have to create a class which inherits from `apama.eplplugin.EPLPluginBase`. Your class must provide a one-argument constructor and pass the argument verbatim to the `EPLPluginBase` constructor. For example:

```
import apama.eplplugin

class MyPluginClass(apama.eplplugin.EPLPluginBase):
    def __init__(self, init):
        super(MyPluginClass, self).__init__(init)
    ...
```

The base class provides two member functions to derived classes:

Member	Description
<code>getLogger()</code>	Returns a logger object which can be used to write to the correlator log file. See also “Writing to the correlator log file” on page 543 .
<code>getConfig()</code>	Returns a dictionary of the configuration from the correlator configuration file. See also “Using Python plug-ins” on page 544 .

A single instance of your class is created for each time it is listed in the configuration file. Functions which you want to expose to EPL are member functions on that instance. To export a function to EPL, you need to declare a member function on the class and decorate it to indicate the name and signature of the function in EPL using the `EPLAction` decorator:

```
@EPLAction("getCounter", "action<string> returns integer")
def lookupCounter(name):
    return counters[name].value()
```

You configure the plug-ins in the YAML configuration file for the correlator, in the `eplPlugins` stanza:

```
eplPlugins:
  counterPlugin:
    pythonFile: counters.py
    class: MyPluginClass
```

Note:

In Software AG Designer, you can easily do this by adding the above configuration to the `config/CorrelatorConfig.yaml` file. For further information, see "YAML configuration file for the correlator" in *Deploying and Managing Apama Applications*.

You load the plug-ins into EPL by using the `import` statement in the monitor or event which wants to use the plug-in:

```
import "counterPlugin" as plugin;
```

Note:

If your plug-in starts any Python background threads, you must ensure all such threads are stopped before unloading the plug-in. Failure to do so can cause the correlator to terminate with a `Py_EndInterpreter` message, indicating that this is not the last thread.

Complete simple example

This example implements a plug-in which simply keeps a single global count.

```
from apama.eplplugin import EPLPluginBase, EPLAction

class CountPlugin(EPLPluginBase):
    def __init__(self, init):
        super(MyPluginClass, self).__init__(init)
        self.count = 0
    @EPLAction("action<integer>", "increment") # override name
    def incrementCount(self, number):
        self.count = self.count + number
    @EPLAction("action<> returns integer")
    def getCount(self):
        return self.count
```

```
eplPlugins:
    countPlugin:
        pythonFile: ${PARENT_DIR}/countplugin.py
        class: CountPlugin
```

```
monitor foo
{
    import "countPlugin" as counter;
    action onload()
    {
        on all A() {
            counter.increment(1);
            print "Current count: "+counter.getCount().toString();
        }
    }
}
```

Method signatures and types

Only methods which take arguments and return values which can be converted into EPL types can be used in an EPL plug-in written in Python. For each action, you must provide the EPL signature for the method to the `EPLAction` decorator. Typing is not strictly enforced by Python, but is enforced by EPL when being used as a plug-in. EPL method signatures take the following forms:

```

action<integer>
action<sequence<string>, integer>
action<> returns integer
action<dictionary<string, string>, string> returns string

```

EPL to Python type conversion

EPL type	Python type	Notes
integer	integer	
string	unicode	
float	float	
boolean	bool	
decimal	decimal.Decimal	
chunk	<i>any type</i>	Maps to an arbitrary Python object.
dictionary<T>	dict	All keys must be the same type. All values must be the same type.
sequence<T>	list	All elements must be the same type.
context	apama.eplplugin.Context	
Channel	unicode or apama.eplplugin.Context	
optional<T>	T	May be None.
location	apama.eplplugin.Location	
EventType	apama.eplplugin.Event	All EPL event types are mapped to a single Python Event type. It has two fields. Type is the event name and fields is a dict of <i>fieldname</i> : <i>fieldvalue</i> .
any	apama.eplplugin.Any	

Methods using args/kwargs

You can either explicitly specify all the arguments that your plug-in methods take or you can rely on **args*-type argument handling.

You can pass any number of EPL arguments into a Python function expecting **args*. This functions in the same way as passing the arguments from within Python.

Expanding a sequence as function parameters is not supported. In this case, `*args` would contain a single parameter, of type `list`. It is possible to achieve this using a wrapper function from within Python. For example:

```
def funcThatUsesArgs(*args):
    ...

@EPLAction("action<sequence<any>>")
def foo(d):
    funcThatUsesArgs(*d)
```

Apama does not invoke plug-in methods with argument names, so `**kwargs` patterns will not work. However, it is possible to use Python functions expecting `**kwargs` by using a wrapper function in the same way as with `*args`. For example:

```
def funcThatUsesKwargs(**kwargs):
    ...

@EPLAction("action<dictionary<string, any>>")
def foo(d):
    funcThatUsesKwargs(**d)
```

Exceptions

Plug-ins can raise exceptions from plug-in methods. They must be derived from `exceptions.Exception` as normal.

If a plug-in throws an exception, this will be turned into an exception in EPL which may be caught with the EPL `try ... catch` statement. See also [“The try ... catch statement” on page 655](#).

If you do not catch the exception, then the calling monitor instance will be terminated with the message in the exception thrown from the plug-in.

Writing to the correlator log file

The `EPLPluginBase` class provides a member function `getLogger` to plug-ins. This can be used to write messages to the correlator's log file.

Log messages are prefixed with the string `plugins.PluginName`, which is also the category that can be used to control the log level for this plug-in via the `correlatorLogging` section in the YAML configuration file (see "Setting correlator and plug-in log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications*).

```
def logMessage(msg):
    getLogger().info("Message is: %s", msg);
```

Documentation about the available functions and log levels can be found in the *API Reference for Python*.

Sending events

Method calls on plug-ins are synchronous. Generally, they should be written not to take too long or hold up processing in the correlator. One technique to enable asynchronous behavior in the plug-in is to interact with EPL by sending events which can be handled asynchronously, potentially from a background thread which is processing events as well as from methods themselves.

The `apama.eplplugin.Correlator` class contains several methods for sending events into the correlator. You can send events either as the string representation of the event in Apama's internal string format (for example, `MyEvent(1.3, true)`) or as a dictionary either with the event type specified explicitly or as an `apama.eplplugin.Event` type. In the dictionary case, the keys are strings corresponding to field names in the event, and the values are the value for those fields in the appropriate type/format for the type of the field.

You can select the destination of the event in two ways:

- **Named channel.** The preferred method is to deliver the event to a specific named channel. This will go to everything which has subscribed to that named channel. Subscribers can either be contexts, external receivers, connectivity chains or other plug-ins which are subscribed to receive events on that channel.

Send as string:

```
Correlator.sendTo("channelName", "MyEvent(42)")
```

Send as object:

```
event = {}
event["number"] = 42
Correlator.sendTo("channelName", event, type="MyEvent")
```

- **Context.** You can deliver the event to a specific context using an `apama.eplplugin.Context` object. Contexts can either be passed into a plug-in from an EPL context object, or they can be obtained with the `apama.eplplugin.Correlator.getCurrentContext()` method.

Send as string:

```
Correlator.sendTo(Correlator.getCurrentContext(), "MyEvent(42)")
```

Send as object:

```
apama.eplplugin.Event event;
event.fields["number"] = 42
event.type = "MyEvent"
Correlator.sendTo(contextObject, event)
```

Using Python plug-ins

After you have created an EPL plug-in in Python, you must configure it in a Python-enabled correlator so that it is available for use in your Apama applications. Applications that use the plug-in also need to import the plug-in by name.

Enabling Python support in the correlator

To enable Python support in the correlator, you must use the `--python` command-line option of the correlator executable. See also "Starting the correlator" in *Deploying and Managing Apama Applications*.

You can also enable Python support using the YAML configuration file for the correlator:

```
correlator:
  pythonSupport: true
```

If you are using a standard Apama installation, a copy of Python is provided in your installation. This Python will be used by default. If you are using the core installer, or wish to use a different version of Python, then you will need to override the location of your Python installation. You can do this by setting the `AP_PYTHONHOME` environment variable.

Adding a Python plug-in to the correlator

EPL plug-ins written in Python are made available to EPL via the YAML configuration file for the correlator (see also "Configuring the correlator" in *Deploying and Managing Apama Applications*).

To configure a specific Python plug-in once you have enabled Python support, you need to add an `eplPlugins` section to the configuration file:

```
eplPlugins:
  myPluginName:
    pythonFile: ${PARENT_DIR}/plugin.py
    class: PluginClass
    pythonPath:
      - ${PARENT_DIR}/dependencies
    config:
      key: value
```

The plug-in name is an arbitrary string which will be used to refer to the plug-in from EPL. The following configuration options are available for each plug-in:

Configuration option	Description
<code>pythonFile</code>	Required. The path to the Python file which contains the plug-in.
<code>class</code>	Required. The name of the class in the file which exposes methods decorated with <code>EPLAction</code> .
<code>pythonPath</code>	Optional. A single string or list of strings containing locations to add to the Python path.
<code>config</code>	Optional. An arbitrary dictionary which will be available to the plug-in via the <code>self.getConfig()</code> method.

You can create multiple instance of the same plug-in with different names.

Importing a Python plug-in to EPL

Once a Python plug-in has been configured, it is available for import using the plug-in name defined in the configuration file. The correlator will make available all methods decorated with the `EPLAction` decorator to be called directly from EPL. For example, the following code imports a plug-in named `TestPlugin` and calls its `dosomething` method:

```
monitor m {  
    import "TestPlugin" as test;  
    action onload()  
    {  
        test.dosomething();  
    }  
}
```

If the plug-in has been incorrectly configured, the correlator will try to load the plug-in as a C++ plug-in and may give an error such as **Error opening plug-in library libTestPlugin.so:**

libTestPlugin.so: cannot open shared object file: No such file or directory. If this happens and you were trying to load a plug-in written in Python, then check the name in your configuration file and make sure that it matches the name you are trying to import.

Python plug-ins and correlator persistence

Since Python plug-ins provide no way to persist data stored inside the plug-in, or in chunks from a Python plug-in, it is not permitted to import a Python plug-in from a persistent monitor or to use an event which imports a Python plug-in from a persistent monitor. You can use Python plug-ins from non-persistent monitors in a persistent correlator.

Installing Python modules

The standard (full) installation of Apama includes a copy of Python which is used by default in the Apama environment. It provides a `pip` (`pip3` on Linux) wrapper to the Python interpreter that is shipped with Apama.

To install a Python module, run the following command in the Apama Command Prompt or `apama_env` wrapper (see also "Setting up the environment using the Apama Command Prompt" in *Deploying and Managing Apama Applications*).

Windows:

```
pip install module
```

Linux:

```
pip3 install module
```

Note:

If you are facing any build issues after reinstalling (`python3 -m pip install --upgrade --force-reinstall pip`) or upgrading (`python3 -m pip install --upgrade pip`) the `pip` module, prefer to use `pip3` shipped with Apama Python.

Sample plug-ins written in Python

Apama provides sample EPL plug-ins written in Python, located in the `samples\correlator_plugin\python` directory of your Apama installation.

IV Protecting Personal Data in Apama Applications

23	Introduction	551
24	Where personal data is held within the Apama platform	553
25	Documenting personal data flows within an Apama application	557
26	Handling personal data in the "in-memory" state of the correlator	559
27	Handling personal data "at rest" in the correlator persistence and JMS datastores	563
28	Handling personal data "in motion" from dashboards	565
29	Handling personal data "at rest" in log files	567
30	Handling personal data "at rest" in the correlator input log file	573
31	Handling personal data "at rest" in containerization environments	575

23 Introduction

Legislation in various parts of the world – such as the General Data Protection Regulation (GDPR) of the European Union (EU) - specifies that personal data cannot be collected and processed without a person's consent or other legitimate basis, and that organizations are responsible for protecting personal data that is entrusted to them. The concept of “personal data” typically covers details that can be used to identify a person, such as the person's name, email address or IP address.

Note:

In the different countries of the EU, the GDPR may be known under another, language-specific name. For example, it known as the Datenschutz-Grundverordnung (DSGVO) in Germany and as Règlement Général à la Protection des Données (RGPD) in France.

Apama is a general-purpose event processing platform on which customers build their own applications. Most of the data handled by Apama is arbitrary customer-defined data whose meaning is defined by the customer who developed the application. Some of that customer-defined data may qualify as “personal data”, so if you are developing applications on the Apama platform, you should be careful to ensure compliance with laws related to that data.

See the following topics for some suggestions to help with identifying how personal data can be protected when building applications on the Apama platform.

24 Where personal data is held within the Apama platform

Most deployments of Apama deal with personal data only in customer-defined data fields, which are largely under the control and responsibility of the customer who writes and deploys the Apama application.

In Apama applications, customer-defined data is usually held “in memory” in EPL event fields and monitor variables, by connectivity plug-ins, or by EPL plug-ins such as the `MemoryStore`.

Customer-defined application data may also be stored “at rest” in the following places:

- Correlator, IAF and dashboard server log files (the main log file, and for the correlator also any additional files defined by `eplLogging` and `correlatorLogging` configuration). These files include logging performed by the customer's application and by standard Apama connectivity and EPL plug-ins and the correlator itself. For example, the contents of Apama events are often logged (either in full, or truncated) if an error occurs during processing or sending of the event, and data from events or other EPL data structures may be logged as part of correlator error messages.
- Correlator input log. If enabled, it contains the contents of all events sent into the correlator.
- Correlator persistence database (and if using JMS, the reliable receive database). If enabled, it contains an on-disk representation of the state of the Apama application.

Apama also provides the ability to store customer-defined data in external systems such as a Terracotta distributed cache (using our `MemoryStore` API) or a database. You should consult the documentation of systems such as these for information about how to ensure personal data written there by your application is properly handled and protected, and you should also check that your Apama application logic includes mechanisms to rectify or erase personal data stored there, if required.

We strongly advise against allowing any personal data to exist in the application logic itself (the EPL source files), and this documentation assumes that this principle is being followed.

In addition to the customer-defined data mentioned above, there are a small number of situations where the Apama platform could potentially be considered to directly handle personal data. You should establish whether in your own environment any of the “users” listed below represent the “personal data” of an identifiable human protected by legislation, and which merely represent machine-to-machine communication, or system administrators who have accepted the logging of their user name and IP address as part of their terms of employment.

Product area	Potential "personal data"	Where data could be stored
Correlator, IAF, dashboard servers	User identifiers and IP addresses for direct connections to/from Apama server processes (typically only for machine-to-machine communication between server processes, or monitoring and management by system administrator accounts). These are logged to provide an audit trail in case of an attack or accidental mistake by a system administrator.	<ul style="list-style-type: none"> ■ correlator main log file ■ correlator input log ■ correlator persistence database ■ IAF and dashboard server log files
Scenario Service API, queries, DataViews, dashboard servers, custom clients and dashboards	<p>The Scenario Service event protocol contains a "username" field, identifying users who created instances of scenarios (for example, DataViews or queries). There are various places where this username could show up.</p> <p>See also "Scenario Service API" in <i>Connecting Apama Applications to External Components</i>.</p>	<ul style="list-style-type: none"> ■ correlator and dashboard server log files ■ correlator input log ■ correlator in-memory state ■ correlator persistence database
Dashboard servers	<p>User identifiers and IP addresses of dashboard clients that connect, who may be end-users. These are logged to provide an audit trail.</p> <p>See also "Managing the dashboard data server and display server" in <i>Building and Using Apama Dashboards</i>.</p>	<ul style="list-style-type: none"> ■ dashboard server log files
Dashboard servers, if using JAAS	<p>User identifiers of dashboard clients for authentication purposes. These are logged to provide an audit trail.</p> <p>See also "Managing the dashboard data server and display server" in <i>Building and Using Apama Dashboards</i>.</p>	<p>Under the control of the JAAS plug-in used. For example, the <code>UserFileLoginModule</code> provided by Apama stores usernames in plaintext in an XML file, whereas other plug-ins are available that hold usernames on a remote server such as an LDAP server. See "Administering Dashboard Security" in <i>Building and Using Apama Dashboards</i>.</p> <p>You can choose an appropriate JAAS plug-in which complies with the way you need to protect the user data if required.</p>

Product area	Potential "personal data"	Where data could be stored
HTTP server connectivity plug-in	<p>User identifiers and IP addresses of clients that connect to the HTTP server, as specified in HTTP header. These are written to the log file. Along with other HTTP headers they are also present in the message metadata. Thus they can optionally be mapped to fields in an Apama event, using a connectivity codec such as the mapper codec.</p> <p>See also "The HTTP Server Transport Connectivity Plug-in" in <i>Connecting Apama Applications to External Components</i>.</p>	<ul style="list-style-type: none"> ■ correlator main log file ■ correlator input log file ■ correlator in-memory state ■ correlator persistence database
HTTP server connectivity plug-in, only if authentication is enabled	<p>User identifiers of clients who are permitted to connect to the HTTP server (with a secure hash of the passwords).</p> <p>See also "The HTTP Server Transport Connectivity Plug-in" in <i>Connecting Apama Applications to External Components</i>.</p>	<ul style="list-style-type: none"> ■ HTTP server authentication password file, which is stored on disk in plaintext and contains un-encrypted usernames and hashed salted passwords, see "Authentication" in <i>Connecting Apama Applications to External Components</i>. As the file is completely under the user's control, you can use standard tools included with your operating system to set access control for protecting this data as needed. Users can be deleted from the file using a text editor or the <code>httpserver_passman</code> provided by Apama as described in the documentation.

25 Documenting personal data flows within an Apama application

You are strongly encouraged to write and maintain a personal data register, a document describing the places where personal data is handled within your application, how long it is stored for, and how that data is protected through access controls and policies. This should also describe the flows of personal data, for example, being explicit about when personal data is passed to an EPL plug-in such as the MemoryStore, or sent outside the correlator to another system such as a Universal Messaging topic or queue, a database or a distributed store such as Terracotta. This register will be useful for demonstrating compliance with regulations, and also for enabling an effective response should any data breach occur.

In addition to documenting occurrences of standard “personally identifiable information” (PII) such as usernames and IP addresses, any “sensitive personal information” (SPI) such as information about medical conditions should be explicitly called out, since additional regulations and a higher level of caution may apply for such data.

At a minimum, we recommend writing ApamaDoc comments on all event definitions and monitors that handle personal data. See [“Generating Documentation for Your EPL Code” on page 429](#) for more information about writing ApamaDoc. Using ApamaDoc as a starting point for describing personal data flows has several advantages:

- It ensures that developers working on the application are aware of the regulatory implications of changes they make to code that involves personal data.
- It helps developers to be more mindful of the need to write the application code in a way that minimizes the amount of personal data held and transferred between parts of the system.
- It reduces the chance of personal data documentation getting out of sync with the codebase, as the application evolves.
- The HTML that can be generated from the ApamaDoc comments provides a great starting point for writing and maintaining other documents regarding personal data that your organization's compliance policies may require.

The conventions and guidelines for ApamaDoc commenting should be defined by the customer developing the Apama application, and checked through a code review process.

See below for an example illustrating a possible use of ApamaDoc to describe personal data storage and flows:

```
/** Represents an incoming order to be handled by Apama.
```

```
Contains "personal data" in the "username" and "ip" fields.
This personal data is stored in the OrderManager monitor, is
stored in the distributed MemoryStore table "MyOrders",
and is also sent out to the Universal Messaging "OrderAlert" topic.
@see OrderManager
*/
event MyOrder
{
    string orderId;
    float timestamp;

    /** This field contains "personal data". */
    string username;

    /** This field contains "personal data". */
    string ip;
}

/** Manages orders.

    Holds "personal data" identifying customers who placed orders, and
    stores this data in the distributed MemoryStore table "MyOrders".

    @listens MyOrder Receives incoming orders containing "personal data".
    @sends OrderAlert Sends "personal data" to the UM "OrderAlert" topic.
*/
monitor OrderManager
{
    /** Contains "personal data". */
    dictionary<string, MyOrder> orders;
    ...
}
```

If you intend to use the HTML ApamaDoc to summarize personal data uses, make sure that the `@private` tag is not applied to any fields containing personal data, which would suppress them from the HTML documentation.

26 Handling personal data in the "in-memory" state of the correlator

In Apama applications, customer-defined data - potentially including personal data - is held "in memory", in various places:

- event fields
- monitor variables
- variables in the scope of an event listener
- MemoryStore
- state held in EPL plug-ins
- state held in connectivity plug-ins

As for languages such as C++ or Java, for Apama it is the responsibility of both the author of the application and those responsible for deploying and using it to ensure that this personal data is handled appropriately, by writing suitable logic into the application, and ensuring that the right policies and access controls are in place.

Some key areas to consider are listed below:

- **Data minimization** should be practiced by ensuring that no personal data is sent into Apama other than that absolutely required to implement the required functionality. For example, consider using the ApamaDoc to review all the event definitions used for data entering or leaving the correlator, and any monitors that store personal data, and use them to check for data that could be safely removed. If there is some personal data that Apama does not need to make direct use of, but must be opaquely passed through to another system that Apama is sending messages to, consider whether it is possible to avoid that data existing as plaintext inside Apama by having the system that generated the messages pass it into Apama as a Base64-encoded encrypted string, with the key shared only between the originating upstream system and the downstream system that needs to use the data.
- **Rectification** (correction) or **erasure** of data can be implemented by identifying which monitor instances and data structures hold personal data, and ensuring that EPL logic is in place to change or delete it for a given person, in response to an Apama event requesting this. If the application has a monitor instance dedicated to each user, this could be as simple as making that monitor instance "die" in response to an event requesting removal. Otherwise, it may be a case of removing that user from dictionaries and other data structures. If the application uses

plug-ins such as the MemoryStore to hold personal data, the EPL application must also remove the keys holding personal data from the MemoryStore. It is important that any rectification or erasure capabilities are built into the application from the beginning and carefully tested, since it would often not be possible to add them once the application is deployed without losing the correlator's state.

- On-demand **access** to a user's personal data can be implemented by identifying where personal data is held and ensuring EPL logic is in place to return it, perhaps by sending an event containing the data in response to an event requesting the data. Personal data can be exported in an open and portable format by using the JSON EPL plug-in to serialize it to a JSON string (see also [“Using the JSON plug-in” on page 395](#)).
- **Pseudonymization** should be practiced where possible. This means keeping the identity of a person and data about that person separate as much as possible. For example, rather than sending a message into Apama containing a user's real name and information such as the user's medical history, the user's name could be replaced by a unique identifier (for example, a GUID) assigned and protected by the upstream system, so that the user's identity and the information about the user are not held together inside the correlator. By separating the user's name from the other information, and ensuring that the mapping between the real name and the assigned unique identifier is kept secure, the risk of the data about that user being leaked and linked back to the person it relates to is significantly reduced. Pseudonymization techniques should be applied as early as possible in the processing of messages, ideally before the message enters the Apama correlator. This will help to minimize the number of systems where both the data about the persons and their identity exist together. If that is not possible, it should be performed as early as possible in the correlator's handling of the data, for example, in a connectivity plug-in before it is passed to any EPL, or in the initial EPL listener but before it is stored in a data structure inside the application. This reduces the chance of the personal data leaking out in a log message.
- A **security audit** trail should be created, where practical, whenever personal data is created, modified or deleted, in order to protect its accuracy and allow errors to be tracked down, whether introduced accidentally or as part of an attack. For example, this could be achieved by using EPL `log` statements. It is possible to configure the file that log entries are written to; so if necessary, audit logging could be written to a dedicated log file. It is usually best to perform logging regarding personal data at the application level in EPL, rather than relying on logging of input events or connections from connectivity plug-ins. However, it is important to ensure that user identifiers in the incoming events can be relied upon, which means using a connectivity plug-in such as the HTTP server that supports per-user authentication, or if using a message bus such as Universal Messaging, ensuring that channel permissions are set appropriately and that the system that is publishing messages can be relied upon to set usernames accurately. See [“Specifying log statements” on page 279](#) for more information about logging from EPL.
- **Security** must be architected into the design and deployment of the Apama application to ensure the information security of personal data handled by Apama, and protect the confidentiality and integrity of data. Key points are:
 - See "Security Requirements for Apama" in *Deploying and Managing Apama Applications* for detailed information about how to ensure your Apama deployment is secure.

- Remember to also fully configure all connected systems to perform adequate authentication and authorization, for example, setting appropriate permissions on all channels if using Universal Messaging (see "Configuring the connection to Universal Messaging (dynamicChainManagers)" in *Connecting Apama Applications to External Components*).
- Note that it is possible for a user with direct access to the correlator's port to receive events passing through the correlator - which may contain customer-defined personal data - or to inject code that may change or access that data. Similar considerations apply to the IAF port and the dashboard server management port. There is no encryption or authentication for these ports, but in a properly configured deployment they would always be locked down using standard operating system configuration tools and firewalls, so that you can be confident that only trusted server processes and system administrators are able to connect to them. For monitoring purposes, products such as webMethods API Gateway can expose required correlator features such as the read-only monitoring REST APIs in a secure way without exposing other features that are security-critical. Alternatively, Command Central can be used for management and monitoring, in which case there is no need for the correlator port to be exposed beyond the machine it is running on.
- Apama provides many ways to get data in and out of the correlator that support encryption (for example, SSL/TLS) and authentication to protect the confidentiality and integrity of data: for example, the dashboard servers, and correlator connectivity plug-ins such as the HTTP server (when configured to use TLS/HTTPS). Be sure to check the documentation for your chosen means of connectivity carefully to ensure all the required security options are enabled. If using a message bus such as Universal Messaging or JMS, ensure that the permissions on the channels/topics/queues are set securely, and that if message publishers are providing information to identify users that will be relied upon for authorization or audit logging, that they are implementing authentication securely to ensure the information is accurate.
- Apama application developers can use EPL to implement any user-specific authorization checks needed to protect access to personal data. However it is essential to check that authentication is being securely performed by the connectivity plug-ins or upstream message publisher that is setting the username field that the EPL will be relying on, to ensure an adequate security audit trail.
- Ensure that you regularly install the latest Apama fixes and keep your operating system fully patched, to ensure the latest security fixes are present.

Apama also provides the ability to store customer-defined data in external systems such as a Terracotta distributed cache (using our MemoryStore API) or a database. You should consult the documentation of systems such as these for information about how to ensure personal data written there by your application is properly handled and protected, and you should also make sure that your Apama application logic includes mechanisms to rectify or erase personal data stored there by the application.

27 Handling personal data "at rest" in the correlator persistence and JMS datastores

You may be using the correlator persistence feature, in which the correlator periodically takes a snapshot of its current in-memory state and saves it on disk. Most of the correlator's in-memory application state that contains personal data will therefore be included in the persistence datastore. There are a few exceptions: state in non-persistent monitors and in-memory stores will not be included. See also "Using Correlator Persistence" in *Developing Apama Applications*.

It is important to protect this personal data "at rest" by setting appropriate permissions on directories where datastore files will be written to ensure only the correlator process (and authorized system administrators and backup processes) have access to it. On Windows, this would mean setting an inheritable Access Control List (ACL) limiting read access to the contained files. On UNIX systems, this would involve restricting read and execute permissions to only the owning user (that is, "700") and if possible also setting a umask of 0077 on the correlator process to ensure files created by the correlator also have locked down permissions.

There is no need to implement rectification or erasure, since the persistence store is intended for access only by the correlator, and any rectification or erasure operations performed on the main correlator state will be automatically replicated into the persistence datastore within a very short time.

There is an additional datastore used to hold recently received JMS messages when reliable correlator-integrated messaging for JMS is used. See "XML configuration bean reference" in *Connecting Apama Applications to External Components* for details about how to configure the `reliableReceiveDatabase` location. The same considerations apply as for the persistence datastore - rectification and erasure are not necessary as only recently received messages are retained, and the datastore should not be read by any process other than the correlator. In the case of `AT_LEAST_ONCE` receivers, messages are retained only until they have been acknowledged by the correlator. In the case of `EXACTLY_ONCE` receivers, unique message identifiers (`uniqueMessageId`) may be retained for longer in order to avoid duplicates (see "Duplicate detection when using JMS" in *Connecting Apama Applications to External Components* for details) but this should not have any privacy implications, provided user identifiers are not used in the message identifiers.

28 Handling personal data "in motion" from dashboards

See "Security Requirements for Apama" in *Deploying and Managing Apama Applications* for detailed information about how to secure dashboard servers in your Apama deployment.

Data will be automatically removed from the dashboard servers shortly after it has been removed from the correlator state.

29 Handling personal data "at rest" in log files

■ Example log messages containing personal data	568
■ Protecting and erasing data from Apama log files	569
■ Recommended log levels	570
■ Recommendations for logging by Apama application code	570

Personal data can appear in the log files of any Apama server process, such as the correlator, IAF, or dashboard servers.

Example log messages containing personal data

These files include logging performed by the customer's application and by standard Apama connectivity, EPL or IAF plug-ins and the correlator, IAF and dashboard servers themselves.

For example, customer application log statements may contain personal data. It is also important to note that the contents of Apama events are often logged (either in full, or truncated so that only the beginning of the event's fields are displayed) if an error occurs during processing or sending of the event, and data from events or other EPL data structures may be logged as part of error messages.

We provide a small set of indicative log messages as examples to give an idea of the kind of data that may be present. Note that this is for illustrative purposes only. It is not possible to provide an exhaustive list of all possible log messages, and the format of the messages shown below is subject to change at any time and should not be relied upon.

When a client connects directly to the correlator port, the IP addresses and username are logged (note that the username is not authenticated, so should not be relied upon for security purposes; this would be either a system administrator or another machine, not an end user):

```
2018-05-30 14:23:43.950 INFO [30188] - Sender engine_inject
(MY_USERNAME) (00000000000435490) (component ID
6561364086281508704/6561645561258219360) connected from 127.0.0.1:6714
```

When the HTTP server connectivity plug-in receives a new incoming connection, the IP address and username is logged (authentication and TLS may optionally be enabled, in which case the username can be relied upon):

```
2018-05-30 14:23:45.257 INFO [29300] -
<connectivity.httpServer.httpServer-instance> Started receiving messages
from host 127.0.0.1
2018-05-30 14:23:45.260 WARN [29300] -
<connectivity.httpServer.httpServer-instance> Authentication failed for
user 'uniqueusername' from host 127.0.0.1.
```

When an event is sent directly to the correlator and cannot be parsed (perhaps due to an application bug, or an error in the format used by the sender), a message like this will be logged (in log messages like this, all the personal data is in customer-defined fields):

```
2018-05-30 16:36:58.609 WARN [29308] - Failed to parse the event
"com.acme.MyEvent("private political opinions go here", "1/2/1990",
"My full name", "MY_USERID")" from My Sender Client due to the error:
Unable to find event type com.acme.MyEvent
```

When a connectivity plug-in fails to transform an incoming message into the form the application is expecting, the message will be logged. The order in which the fields appear is undefined, and it is possible the message will be truncated:

```
2018-05-30 16:44:47.811 WARN [17644:processing] -
<connectivity.myCodec.myFirstChain> Codec plug-in MyCodec failed to
transform message Message<metadata={ "sag.channel"="MyFirstChainChannel",
```



```
"sag.type"="MyMessage", "username":"J_BLOGGS", payload={"medical
info": "Embarrassing medical info here", "name":"Joe Bloggs",
"username":"J_BLO...}>: java.lang.Exception: Something bad happened
```

When correlator-integrated JMS receives an invalid incoming message, it may log some or all of the body (which may contain customer-defined personal data), potentially truncated if it is long:

```
2018-05-30 16:25:37.114 WARN
[11708:JMSReceiver:myConnection-receiver-apama-queue-01] -
1 mapping rule warning(s) while mapping to target event:
test.TestMessage("DOB=1/2/1990","Embarrassing medical info here",
"Joe Bloggs","J_BLO...:
- MappingRule<source="{jms.body.textmessage}",
  target="{apamaEvent['str']}", action="xpath", actionResource="/xxx">
- Exception evaluating the xpath "/xxx": org.xml.sax.SAXParseException;
  lineNumber: 3; columnNumber: 70; The element type "xxx" must be
  terminated by the matching end-tag "</xxx>".
with source JMS message:
Property.USERNAME=J_BLOGGS
---> JMSDestination=Queue<apama-queue-01>
---> Body=<mydata>
---> <val key="date_of_birth">1/1/2018</val>
---> <val key="name">Joe Bloggs</val>
---> <val key="medicalinfo">Embarrassing medical info here</val>
---> <val key="username">J_BLO...
```

Protecting and erasing data from Apama log files

To protect the security of personal data in log files, it is important that operating system file permissions are set on the log files and directory containing them to ensure that only the correlator process and authorized system administrators have access to the files. On Windows, this would mean setting an inheritable Access Control List (ACL) limiting read access to the contained files. On UNIX systems, this would involve restricting read and execute permissions to only the owning user (that is, "700") and if possible also setting a umask of 0077 on the correlator process to ensure files created by the correlator also have locked down permissions.

As there are many situations in which usernames, IP addresses or events containing personal data may be logged, including by customer-provided plug-ins and third-party libraries, it is not practical to enumerate all of the log messages that may contain such data, or the set of categories they may be logged under.

Log files are by nature immutable and formatted for reading by human system administrators (not machines), so rectification of data contained within them does not make sense, and erasure of data for individual persons is not practical. The retention of complete information in log files also serves an important and legitimate purpose, in providing a security audit trail, and the ability to diagnose and fix accidental or unlawful events compromising the availability, integrity or confidentiality of the application and personal data it contains.

For these reasons, the recommended approach to protecting personal data in Apama log files is to regularly rotate the logs, and archive the old log files to a secured location protected by encryption.

Optionally, old log files may be deleted after a set time period, though this should be done only when necessary as it will destroy information that might be important for diagnosing bugs or

attacks that compromise the integrity or availability of the application. Software AG may not be able to provide assistance with support requests if the relevant log files have been deleted.

Apama provides a variety of mechanisms for rotating its various log files. These can be combined with operating system features such as Linux's periodic cron jobs, Windows Scheduled Tasks, or common utilities such as `logrotate` and `gnupg`, to implement whatever log handling scheme best fits with your organization's data protection policies. For full information about how to rotate logs, see the following topics:

- "Rotating correlator log files" in *Deploying and Managing Apama Applications*.
- "IAF log file rotation" in *Connecting Apama Applications to External Components*.
- "Rotating the log files of the data server and display server" in *Building and Using Apama Dashboards*.

You may wish to inform your employees or end-users - or in some cases request from them - regarding the fact personal data may be stored in server log files, along with details of the steps your organization takes to protect the data they contain.

Recommended log levels

We recommend the main correlator, IAF and dashboard server log level to always be set to `INFO`. If it is set to `WARN` or higher, then security-relevant events will not be recorded and diagnosing failures can be difficult. But if it is set to `DEBUG` or lower, then there will be a significant performance impact and security-sensitive information will likely be written to the logs.

For similar reasons (as well as to avoid performance problems), it is important not to go into production with diagnostic logging of input/output messages enabled. For example, do not enable the Diagnostic codec connectivity plug-in or set `logJmsMessages` to `true`, except for non-production testing when there is no real personal data present in the messages.

Recommendations for logging by Apama application code

If you are developing EPL applications, connectivity plug-ins or EPL plug-ins, you will need to make your own choices about what information to log and at what level from the code you write. To comply with the principle of data minimization, it is best to avoid unnecessarily including personal data in log output. So where possible, avoid logging details such as username, IP address or the contents of messages, unless needed for security auditing or for legitimate interests such as diagnosing and resolving application errors. In some cases, "pseudonymization" will be possible. That is, when logging personal data, use an application-generated globally unique identifier (GUID) instead of a username, or an IP address that could be used to link the data to an individual person, and protect the mapping between GUIDs and usernames.

It is important to select an appropriate log level for application-generated log messages. It is possible to select different log levels for individual packages within your EPL application, and to direct the output to different log files. See "Setting EPL log files and log levels dynamically" in

Deploying and Managing Apama Applications for more details. If using multiple log files, ensure that the same file system permissions and secure rotation policies are applied to all of them.

You may wish to gather together all the security audit logging from your EPL application into a single file, or perhaps all of the logging that may include personal data. As EPL log statements are written to a category based on the event definition where they exist, these use cases can be addressed by defining a dedicated Apama event definition to perform the logging. For example:

```
package com.mycompany;
event SecurityAuditLogging
{
    action logModification(string username, string resource)
    {
        log "Security event: user '"+username+"' modified resource: "
          +resource at INFO;
    }
}
...
SecurityAuditLogging.logModification("myuserid", "resource");
```

The log level and log file for this could then be configured in the correlator's YAML configuration file as described in "Setting EPL log files and log levels in a YAML configuration file" in *Deploying and Managing Apama Applications*. For example:

```
eplLogging:
  com.mycompany.SecurityAuditLogging:
    file: apama-security-auditing.log
    level: INFO
```


30 Handling personal data "at rest" in the correlator input log file

The correlator has an optional input log, which when enabled records all incoming Apama events to a text file on disk (see also "Replaying an input log to diagnose problems" in *Deploying and Managing Apama Applications*). This can be very useful for diagnosing and reproducing problems experienced in a production environment, for use when you are debugging your application, or by Software AG support.

Many of the same considerations apply to the input log as to other correlator log files: it is essential to protect the contents by setting appropriate file system permissions on the input log files.

As with the other log files, it is possible to periodically rotate the input log as described in "Rotating an input log file" in *Deploying and Managing Apama Applications*. Note that an incomplete input log is useless. So if rotation is in use, it is important that all previous input logs from a given invocation of the correlator are able to be retrieved if necessary from secure backups or archives (that is, they have not been deleted). It is necessary to manually concatenate the input log files before they can be used with the `extract_replay_log.py` script.

Input logs are read-only logs (not databases), and the file format is not intended for modification by users, so rectification of personal data is not relevant.

In some cases, you may wish to implement erasure of personal data in input logs, particularly if they are being kept for a long time. Input logs contain every Apama event sent into the correlator. The input log file format is not intended for consumption or editing by customers. However, in practice it is usually safe to remove individual lines that start with "EVNT" (indicating an incoming event), and this could be used to strip out lines containing the personal data of a particular user on request. It is possible that removing lines from the input log will prevent it from accurately replaying the original behavior, but in most cases where the processing of different users is fairly independent, it is likely to work adequately.

If correlator persistence is enabled, then the input log contains a copy of the persistence datastore at the point when the correlator was started. It is not possible to provide any erasure of data in the persistence datastore, so this approach is only possible when persistence is disabled.

31 Handling personal data "at rest" in containerization environments

As described in the previous topics, there are a number of places where Apama will store data “at rest” on disk. If you are using a containerization environment such as Docker, we recommend putting log files and persistence data stores on storage external to the container, in order to make security permissions and any rotation/expiry policies easier to enforce.

Clients should select appropriate storage requirements from their cloud provider to meet the privacy and security requirements their data has.

V EPL Reference

32	Introduction	579
33	Types	583
34	Events and Event Listeners	601
35	Monitors	619
36	Queries	627
37	Aggregate Functions	641
38	Statements	645
39	Expressions	659
40	Variables	677
41	Lexical Elements	683
42	Limits	695

32 Introduction

■ Hello World example	580
-----------------------------	-----

The Apama Event Processing Language (EPL) is the native language of the Apama correlator. You use EPL to write programs that process events in the correlator. This EPL reference is a companion to the Apama EPL tutorials in Software AG Designer and , which you can use to learn how to write programs in EPL. Use this EPL reference to answer questions and obtain complete details about a particular construct.

EPL is a flexible and powerful curly-brace, domain-specific, language designed for writing programs that process events.

In EPL, an event is a data object that contains a notification of something that has happened, such as a customer order was shipped, a shipment was delivered, a sensor state change occurred, a stock trade took place, or myriad other things. Each kind of event has an event type name and one or more data elements (called event fields) associated with it. External events are received by one or more adapters, which receive events from an event source and translate them from a source-specific format into Apama's internal canonical format. Derived events can be created as needed by EPL programs.

Note:

MonitorScript is the old name for EPL. You might still see the old name in the product documentation.

Hello World example

Though it contains many of the familiar constructs and features found in general-purpose programming languages like Python or Java, EPL also has special features to make it easy to aggregate, filter, correlate, transform, act on, and create events in a concise manner. Here is the canonical "hello world" example written in EPL:

```
monitor HelloWorld
{
    action onload()
    {
        print "Hello world!";
    }
}
```

The Apama event processor, called the correlator, receives events of various types from external sources and routes them to one or more active EPL programs, called monitors or queries.

- Monitors have registered event handlers, called listeners, for events of particular types with specific combinations of data values or ranges of values. When the correlator detects an event of interest, it calls the appropriate event handlers. If there are no handlers for an event, the correlator discards it or passes it to an event handler specifically for events that have no handler.

Event handlers in EPL are conceptually similar to methods or functions used for handling user-interface events in other languages, such as Java Swing or SWT applications. In EPL, code is executed only in response to events. Except, that is, for the special EPL `onload()`, `ondie()`, and `onunload()` actions. See [“Monitor lifecycle” on page 620](#) for information about these actions.

- Queries define particular event types as input and then partition incoming events of those types according to a specified key. For example, a query might partition bank transactions

according to their account numbers. Like a monitor, a query watches for an event pattern of interest, but it does this in each partition independently of every other partition.

When the correlator finds a match, it executes the procedural code specified in the query.

33 Types

■ Primitive and string types	584
■ Reference types	584
■ Default values for types	586
■ Type properties summary	587
■ Timestamps, dates, and times	590
■ Type methods and instance methods	591
■ Type conversion	592
■ Comparable types	593
■ Cloneable types	594
■ Potentially cyclic types	595
■ Support for IEEE 754 special values	598

EPL has primitive types and reference types. Data in the primitive types are simple scalar values. Reference types (also called complex types or object types) have values that are more complicated and some, like the `dictionary` type, have multiple values and have definitions that involve more than one type.

When values are passed as parameters in action and method invocations, primitive types are passed by value, and reference types are passed by reference. When a parameter is passed by value, the called action or method receives a copy of the value and has no direct way to change the variable that the value may have been derived from. When a parameter is passed by reference, the called action or method receives a reference instead of a copy and if the called action changes the value, the caller also sees the change.

In addition to the primitive types and reference types, there is also the `monitor` pseudo-type which provides static methods to configure the monitor instance or context they are called from.

Note that there is no type equivalent to a memory address or pointer.

See the *API Reference for EPL (ApamaDoc)* for detailed descriptions of all of these types. You can find them under **All Types**, and under the headings **Built-in types** and **Aggregates**. Alternatively, you can find them in the following packages:

- `<Default Package>` (most of the built-in types can be found here)
- `com.apama` (includes the `Channel` type)
- `com.apama.aggregates` (includes all of the built-in aggregate functions such as `avg`)
- `com.apama.exceptions` (includes the `Exception` and `StackTraceElement` types)

Primitive and string types

Apama supports the following primitive types:

- `boolean`
- `decimal`
- `float`
- `integer`

In addition, there is `string` which is technically a reference type. However, strings are immutable. Therefore, `string` behaves more like a primitive type than a reference type.

Reference types

In addition to the primitive types, EPL provides for a number of object types. These types are manipulated by reference as opposed to by value (in the same way as complex types are handled in Java). These are the following reference types:

- `action`

- any
- Channel
- chunk
- context
- dictionary
- event
- Exception
- listener
- location
- optional
- sequence
- StackTraceElement
- stream

When a variable of reference type is assigned to another one of the same type, the latter will reference the same object as the former, and should one be changed, the other one would reflect the change as well.

If you require a variable of reference type to contain a copy of another one of the same type, that is a completely distinct but identical copy, then you should use the `clone()` method as described below. This returns a deep copy of the variable, that is, it copies it and all its contents (and their contents in turn) recursively.

The `string` type is technically a reference type, but unlike all other reference types, the `string` type is immutable; its value cannot change. The `clone()` method has no effect on strings, as they cannot be changed. Therefore, `string` behaves more like a primitive type.

Note that you cannot use an object type for matching in an event template. For example, suppose you have the following event types:

```
InnerEvent
{
    float f;
}

WrapperEvent
{
    string s;
    InnerEvent anInnerEvent;
}
```

The following statement is correct:

```
on all WrapperEvent(s = "some_string")
```

However, the following statement is not allowed:

```
on all WrapperEvent(anInnerEvent.f = 5.5)
```

More than one variable can have a reference to the same underlying data value. For example, consider the following code:

```
sequence <integer> s1;
sequence <integer> s2;
s1 := [12, 55, 42];
s2 := s1;
print s1[1].toString; // print second element of s1
s2[1] := 99; // change the second element
print s1[1].toString; // print second element of s1 again
```

Both `s1` and `s2` refer to the same array, so whichever variable you use, there is only one copy of the data values. So the program's output is:

```
55
99
```

Default values for types

The following table lists the default values for the primitive types and reference types.

Type	Description	String form
action	An empty value that throws an exception if you try to execute the action.	
any	Empty value.	<code>any()</code>
boolean		<code>false</code>
chunk	Contains no state. Each plug-in must define what to do upon receiving a default-initialized chunk as an argument.	
context	An empty context that cannot be used in any meaningful way. To use this variable, you must explicitly assign a context that was created with a name.	
decimal		<code>0.0d</code>
dictionary	Empty dictionary.	<code>{}</code>
event	Instance of the event where each of its fields has the standard default values as per this table.	<code>event_name (default fields)</code>
float		<code>0.0</code>
integer		<code>0</code>

Type	Description	String form
listener	An empty listener that cannot be used in any meaningful way. To use this variable, you must assign a listener to it from within an on statement, from another listener variable, or from a stream listener in a from statement.	
location		(0.0,0.0,0.0,0.0)
optional	Empty.	optional()
sequence	Empty sequence.	[]
stream	An empty stream that cannot be used in any meaningful way. To use the variable you must assign a non-empty stream to it.	
string	Empty string.	""

Type properties summary



















Apama type properties include the following:

- **Indexable** — An indexable type can be referred to by a qualifier in an event template.
- **Parseable** — A parseable type can be parsed and has `canParse()` and `parse()` methods. The type can be received by the correlator.
- **Routable** — A routable type can be a field in an event that is
 - Sent by the `route` statement
 - Sent by the `send...to` or `enqueue...to` statement
 - Sent outside the correlator with the `emit` statement
- **Comparable** — A comparable type can be used as follows:
 - Dictionary key
 - Item in a sequence on which you can call `sort()` or `indexOf()`
 - Stream query partition key
 - Stream query group key
 - Stream query window `with-unique` key
 - Stream query `equi`join key

- **Potentially cyclic** — A potentially cyclic type uses the `@n` notation when it is parsed or converted to a string. When a potentially cyclic type is cloned, the correlator uses an algorithm that preserves aliases. See [“Potentially cyclic types” on page 595](#)
- **Acyclic** — An acyclic type is a type that is not potentially cyclic.
- **E-free** — E-free types cannot contain references to instances of a particular event type E. This property is used only to determine whether E is acyclic.

The following table shows the properties of each Apama type.

Type	Indexable	Parseable	Routable	Comparable	Acyclic	E-free
boolean	✓	✓	✓	✓	✓	✓
decimal	✓	✓	✓	✓ ¹	✓	✓
float	✓	✓	✓	✓ ¹	✓	✓
integer	✓	✓	✓	✓	✓	✓
string	✓	✓	✓	✓	✓	✓
location	✓	✓	✓	✓	✓	✓
Channel	✗	✓ ²	✓	✓	✓	✓
Exception	✗	✓	✓	✓	✓	✓
context	✗	✗	✓	✓	✓	✓
any	✗	✓ ³	✓ ³	✓ ³	✗	✗
listener	✗	✗	✗	✗	✓	✓
chunk	✗	✗	✗	✗	✓	✓
stream	✗	✗	✗	✗	✓	✓
action	✗	✗	✗	✗	✗	✗
sequence	✗	⚠	⚠	★	⚠	⚠

Type	Indexable	Parseable	Routable	Comparable	Acyclic	E-free
dictionary						
optional						
event E		 ⁴	 ⁴			

Legend:

Symbol Description



Yes. This type has the corresponding property.

¹ Attempts to use a NaN in a key terminates the monitor instance.

² A Channel object is parseable only when it contains a string.

³ The any type is treated as parseable, routable and comparable, but it is comparable only if the contained type does not contain aliases. If the type of the value contained within it does not meet these requirements, such operations will throw an exception at runtime. An empty any value is parseable and comparable, and can be routed as a field of an event, but not as an argument to route.



No. This type does not have the corresponding property.



This type inherits the corresponding property from its constituent types, that is, the item type in a sequence, the key and item types in a dictionary, the types of fields in an event. The type has the corresponding property only when all its constituent types have that property.

⁴ An event defined inside a monitor cannot be received from an external source nor emitted from that correlator. An event defined inside a monitor can be sent or enqueued only within the same correlator.



The type is comparable only when all its constituent types are both comparable and acyclic.



An event E is acyclic only when all its constituent types are both acyclic and E-free.

Examples

The following code provides examples of event type definitions and their properties.

```
// You can do everything with "Tick", including index both its fields.

event Tick {
    string symbol;
```

```
    float price;
}

// You can do everything with "Order", except refer to its target or
// properties fields in an event template.

event Order {
    string customer;
    Tick target;
    string symbol;
    float quantity;
    dictionary<string,string> properties;
}

// The correlator cannot receive the next event as an external event,
// but you can send it, route it, or enqueue it to a context.

event SubscriptionRequest {
    string channel;
    context recipient;
}

// You can do very little with this event except access its members and
// methods. It cannot be routed, you cannot sort sequence<TimeParse>,
// trying to group a stream query by TimeParse is illegal, and so on.

event TimeParse {
    import "TimeFormatPlugin" as TF;
    string pattern;
    chunk compiledPattern;
}

// This has all the same restrictions as TimeParse, but is also
// potentially cyclic, so will use the @n format when parsed or
// converted to a string.

event Room {
    string roomName;
    float squareFeet;
    sequence<Room> adjacentRooms;
    sequence<Employee> occupants;
}
```

Timestamps, dates, and times

Although EPL does not have time, date, or datetime types, timestamp (a date and time) values can still be represented and manipulated because EPL uses the `float` type for storing timestamps. See [“currentTime” on page 680](#).

Timestamp values are encoded as the number of seconds and fractional seconds (to a resolution of milliseconds) elapsed since midnight, January 1, 1970 UTC and do not have a time zone associated with them. Although the resolution is to milliseconds, the accuracy can be plus or minus 10 milliseconds, or some other value depending on the operating system.

If you have two float variables that both contain timestamp values, subtracting one from the other gives you the difference in seconds.

You can add or subtract a time interval from a timestamp by adding or subtracting the appropriate number of seconds (60.0 for 1 minute, 3600.0 for 1 hour, 86,400.0 for 1 day, and so forth).

See also:

- `event.getTime()` for information about when the correlator assigns timestamps to events (see the *API Reference for EPL (ApamaDoc)*).
- [“Using the TimeFormat Event Library” on page 341](#) for information about formatting timestamps.

Type methods and instance methods

There are two kinds of inbuilt methods: type methods and instance methods. Type methods are associated with types. Instance methods are associated with values.

Type methods

To call a type method, you specify the name of the type followed by a period, followed by the method name with its parameters enclosed in parentheses. Some methods do not have parameters and for them you must supply an empty parameter list.

Examples:

```
event someEvent;  
{  
    integer n;  
}  
integer i;  
i:=integer.getUnique();  
print someEvent.getName();
```

Instance methods

Each type (except `action`), whether primitive or reference, has a number of instance methods that provide a number of useful functions and operations on instance variables of that type. These methods are quite similar to actions except that they are predefined and associated with variables, not monitors or events.

To call an instance method, you specify an expression followed by a period and the name of the method, followed by a parenthesized list of actual parameters or arguments to be passed to the method when it is called. Some methods do not have parameters and for them you must supply an empty parameter list.

Examples:

```
integer i := 642;  
float f;  
f := i.toFloat ();  
print f.formatFixed (5);
```

See also

See the descriptions of the built-in types in the *API Reference for EPL (ApamaDoc)* for the methods you can call on types and instances.

Type conversion

EPL requires strict type conformance in expressions, assignment and other statements, parameters in action and method calls, and most other constructs. The only implicit conversion in EPL is to convert concrete types to the any type. This means that:

- The left and right operands of most binary operators must be of the same type.
- An actual parameter passed in a method or action invocation must be of the same type as the type of the corresponding formal parameter in the action or method definition, or the parameter's type must be the any type.
- The expression on the right side of an assignment statement must be the same type as that of the target variable, or the target variable is of the any type, or an optional of the value type.
- The expression in a variable initializer must be the same type as that of the target variable, or the target variable is of the any type, or an optional of the value type.
- The expression in a subscript expression (to locate a sequence entry) must be integer. The expression used as an index for a dictionary must be the same type as the dictionary's key type (or the dictionary's key type is any).
- The expression in a return statement must be the same type as that of the action's returns clause, or the return type is the any type.

For conversions between concrete types, the inbuilt methods on each type include a set of methods which perform type conversion. For example:

```
string number;  
integer value;  
number := "10";  
value := number.toInteger();
```

This illustrates how to map a string to an integer. The string must start with some numeric characters, and only these are considered. So if the string's value was 10h, the integer value obtained from it would have been 10. Had the conversion not been possible because the string did not start with a valid numeric value, then `value` would have been set to 0.

These method calls can also be made inside event expressions as long as the type of the value returned is of the same type as the parameter where it is used. Therefore one can write:

```
on all StockTick("ACME", number.toFloat());
```

Method calls can be chained. For example one can write:

```
print ((2 + 3).toString().toFloat() + 4.0).toString();
```

Note that as shown in this example, method calls can also be made on literals.

The following table indicates the source and target type-pairs for which type conversion methods are provided.

Source Type	Target Type									
	any	boolean	decimal	dictionary	event	integer	float	optional	sequence	string
any	assign and clone	cast	cast	cast	cast	cast	cast	cast	cast	cast valueToString() toString()
boolean	assign	assign						assign		toString()
decimal	assign		assign			round() ceil() floor()	toFloat()	assign		toString()
dictionary	assign			assign and clone				assign		toString()
event	assign				assign and clone			assign		toString()
integer	assign		toDecimal()			assign	toFloat()	assign		toString()
float	assign		toDecimal()			round() ceil() floor()	assign	assign		toString()
optional	assign	getOrThrow()	getOrThrow()	getOrThrow()	getOrThrow()	getOrThrow()	getOrThrow()	assign	getOrThrow()	getOrThrow()
sequence	assign							assign	assign and clone	toString()
string	assign	toBoolean() parse()	toDecimal() parse()	parse()	parse()	toInteger() parse()	toFloat() parse()	assign	parse()	assign and parse()

In the table above, “assign” means values of the type can be directly assigned to another variable of the same type, without calling a type conversion method. “clone” means a value of the type can be copied by calling the `clone()` method. “cast” means that any type values can be cast (see [“Handling the any type” on page 269](#) for more information).

Comparable types

The following types are comparable, and the operators `<`, `>`, `<=`, `>=`, `=`, or `!=` can be used to compare two values of one of these types if both are the same type:

- `boolean`
- `decimal`
- `float`
- `integer`
- `string`
- `context`
- `dictionary` if it contains items that are a comparable type
- `event` if it contains only comparable types
- `location`
- `sequence` if it contains items that are a comparable type
- `optional` if it contains a comparable type
- `any` if it contains items that are a comparable type
- `com.apama.exceptions.Exception`

- `com.apama.exceptions.StackTraceElement`

The correlator cannot compare the following types of items:

- `action`
- `chunk`
- `dictionary` if it contains items that are an incomparable type
- `event` if it contains at least one incomparable type
- `listener`
- `sequence` if it contains items that are an incomparable type
- `stream`
- `any` if it contains items that are an incomparable type
- Potentially cyclic types

For details about how the correlator compares items of a particular type, see the topic about that type.

In EPL code, you must use a comparable type in the following places:

- As the key for a dictionary. The type of the items in the dictionary does not need to be comparable.
- In a sequence if you want to call the `indexOf()` or `sort()` method on that sequence.
- As a key in the following stream query clauses:
 - `Equi-join`
 - `group by`
 - `partition by`
 - `with unique`

Cloneable types

Since variables of reference types are bound to the runtime location of the value rather than the value itself, direct assignment of a variable of reference type copies the reference (that is, the value's location) and not the value. To make a copy of the value, you must use the `clone` instance method instead of assignment. The types that have this property are called cloneable types.

The cloneable types are `string`, `dictionary`, `event`, `location`, `optional`, `any` and `sequence`.

For `dictionary`, `event`, `any` and `sequence` types, the behavior of the `clone()` method varies according to whether or not the instance is potentially cyclic.

- When the instance is potentially cyclic, the correlator preserves multiple references, if they exist, to the same object. That is, the correlator does not create a copy of the object to correspond to each reference. See also [“Potentially cyclic types” on page 595](#).
- When the instance is not potentially cyclic, and there are multiple references to the same object, the correlator makes a copy of that object to correspond to each reference.

While you can call the `clone()` method on a `stream` value, or a value that indirectly contains a `stream` or `listener` value, cloning returns another reference to the original `stream` or `listener` and does not clone it.

Potentially cyclic types

A cyclic object is an object that refers directly or indirectly to itself. For example:

```
event E {
    sequence<E> seq;
}
E e := new E;
e.seq.append(e);
```

When an object is cyclic or contains a reference to a cyclic object, it can be referred to as containing cycles. If it is possible to create an object that contains cycles, the type of that object is referred to as potentially cyclic.

When a type has the potential to contain cycles, and you call `parse()` on that type, or `toString()` or `clone()` on an object of that type, the result is different from when those methods are called on a type, or object of a type that is not potentially cyclic. Consequently, it is sometimes important to understand which types are potentially cyclic and what the string form of these objects looks like.

Which types are potentially cyclic?

A type is potentially cyclic if it contains one or more of the following:

- A dictionary, sequence or optional type that has a parameter that is of the enclosing type. For example:

```
event E {
    dictionary<integer,E> dict;
}
event E {
    sequence<E> seq;
}
event E {
    optional<E> opt;
}
```

- An action variable member. For example:

```
event E {
    action<E> a;
}
```

- An any variable member. For example:

```
event E {  
    any a;  
}
```

- A potentially cyclic type. For example:

```
event E {  
    sequence <E> seq;  
}  
  
event F {  
    E e;  
}
```

F does not have any members that refer back to F, nor does it contain any action variables. However, it does contain E, which is a potentially-cyclic type. Therefore, an instance of F might contain cycles.

Likewise, a dictionary or sequence is potentially cyclic if it has a parameter that is a potentially cyclic type. Consider the following event type:

```
event E {  
    sequence <E> seq;  
}
```

Given this event type, `dictionary<string, E>` is potentially cyclic because its parameter is potentially-cyclic. Similarly, `sequence<E>` is potentially cyclic.

A cyclic object can indirectly contain itself. Consider the following, using the same definition of E as above.

```
E e1 := new E;  
E e2 := new E;  
e1.seq.append(e2);  
e2.seq.append(e1);
```

In this example, both e1 and e2 are cyclic:

- e1 is `e1.seq[0].seq[0]`
- e2 is `e2.seq[0].seq[0]`

Following is another example of an object that indirectly contains a cycle:

```
E e3 := new E;  
E e4 := new E;  
e3.seq.append(e4);  
e4.seq.append(e4);
```

In this example, e3 is cyclic, even though it does not refer back to itself. Instead, e3 refers to e4 and e4 refers back to itself.

You can pass objects that contain cycles between EPL and Java. Remember that JMon programs do not support action type variables, and so any cyclic types you pass cannot contain them.

String form of potentially cyclic types

A potentially cyclic object might have more than one reference to the same object. When you need the string form of a potentially cyclic object, the correlator uses a special syntax to ensure that you can distinguish multiple references to the same object from references to separate objects that merely have the same content.

When the correlator converts a potentially cyclic object to a string, the correlator labels that object @0. If the correlator encounters a second object during execution of the same method, it labels that object as @1, and so on. Whenever the correlator encounters an object that it has already converted, it outputs that object's @*index* label rather than converting it again. For example:

```
event E { sequence<E> seq; }
E e := new E;
e.seq.append(e);
print e.toString(); // "E([@0])"
```

Following is a more complicated example:

```
event Test {
  string str;
  sequence<Test> seq;
  string str2;
}

monitor m {
  action onload() {
    Test t:=new Test;
    t.str:="hello";
    t.str2:=t.str;
    t.seq.append(t);
    Test t2:=new Test;
    t.seq.append(t2);
    t.seq.append(t2);
    t2.seq.append(t);
    print t.toString();
  }
}
```

This prints the following:

```
Test("hello",[@0,Test("",[@0],""),@2],"hello")
```

The objects @0, @1, @2, and @3 correspond to the following:

@0	Test("hello",[@0,Test("",[@0],""),@2],"hello")	t in the above example
@1	[@0,Test("",[@0],""),@2]	t.seq in the above example
@2	Test("",[@0], "")	t2 in the above example
@3	[@0]	t2.seq in the above example

The following example uses the `clone()` method and contains `action` references. The result uses the new string syntax for aliases to the same object.

```
event E {
    action<> act;
    sequence<string> x;
    sequence<string> y;
}

monitor m {
    action onload() {
        E a:=new E;
        a.x.append("alpha");
        a.y:=a.x;
        E b:=a.clone();
        b.x[0]:="beta";
        print b.y.toString();
        print a.toString();
    }
}
```

The output is as follows:

```
["beta"]
E(new action<>,"alpha"),@1)
```

Note that dictionary keys can never contain aliases so they do not receive *@n* labels for referenced objects in `toString()` and `parse()` methods.

Whether you need to do anything to handle this string syntax depends on why you want a string representation of your object:

- If you are using the string for diagnostic messages, you just need to understand the syntax.
- If you plan to feed the string into the `parse()` method, the `parse()` method will handle it correctly.
- If you plan to feed the string into some other program, you should either avoid repeated references in an object or make sure the other program can handle the *@index* syntax.

Support for IEEE 754 special values

EPL supports the following IEEE 754 special `float` and `decimal` values:

- `NaN` — In EPL, these are quiet NaNs. The string representation is `"NaN"`.
- `+Infinity` — The string representation is `"Infinity"`.
- `-Infinity` — The string representation is `"-Infinity"`.

The correlator returns one of these values as the result of an invalid computation. For example, dividing zero by zero or calculating the square root of a negative number. The correlator returns infinities as the result of computations that overflow, for example, taking a very large number and dividing it by a very small number.

The correlator can receive external events that contain these special values. You can send, route, emit, and enqueue events that contain these values. If the correlator receives an event that contains

a floating point value that is too large to be represented as a 64-bit floating point number, the behavior is as if the value had overflowed and the correlator represents the value as infinity.

See the descriptions of `decimal` and `float` in the *API Reference for EPL (ApamaDoc)* for more information.

The following operations return NaN:

- `0.0/0.0`
- `x.sqrt()` (if `x < 0`)
- `x.ln()` (if `x < 0`)
- `x.log10()` (if `x < 0`)
- `Infinity - Infinity`
- `0.0 * Infinity`

In addition, most operations that accept NaN as a parameter return NaN. For example:

- `NaN.exp() = NaN`
- `NaN + 3.0 = NaN`

The NaN value behaves differently when compared to other floating point numbers. NaN does not compare equal to any other number, including itself. It is unordered with respect to all other floating point numbers, so `NaN < x` and `NaN > x` are both `false`.

The following operations return positive infinity (note that IEEE 754 has signed zeroes):

- `x/0.0` (if `x > 0`)
- `x/-0.0` (if `x < 0`)
- `Infinity.sqrt()`

The following operations return negative infinity:

- `x/0.0` (if `x < 0`)
- `x/-0.0` (if `x > 0`)
- `(0.0).ln()`

The constants that are supported by EPL ensure consistent values, and a few have been provided for convenience. For a list of all available constants, see the descriptions of `decimal`, `float` and `integer` in the *API Reference for EPL (ApamaDoc)*.

Operations where rounding is required use the IEEE standard approach of “round half to even” (also known as “banker's rounding”).

34 Events and Event Listeners

■ Event definitions	602
■ Event templates	604
■ Event listener definitions	609
■ Event lifecycle	609
■ Event listener lifecycle	610
■ Event processing order for monitors	611
■ Event processing order for queries	612
■ Event expressions	613
■ Event channels	618

In EPL, an event is a data object that is a notification of something happening, such as arrival of a customer order, shipment delivery, sensor state change, stock trade, or myriad other things. Each kind of event has an event type name, zero or more data elements or fields, and zero or more event actions associated with it.

Event objects can also be used simply as complex data structures to hold multiple related data values. They can also be used as a container for actions that can be shared by multiple monitors.

Event objects are hierarchical structures that can contain simple values, other events, and arrays.

When the correlator executes an `on` statement, it creates an event listener. An event listener watches for an event, or a sequence of events, that matches the event or event sequence specified in the `on` statement. Conceptually, event listeners sift the events that come in to the correlator and watch for matching events.

Event definitions

An event definition specifies the event type, and any event fields and/or event action fields.

Example:

```
event MyEvent {  
    string s;  
    MyOtherEvent e;  
    location l;  
    wildcard integer i;  
}
```

For detailed information, see [“Defining event types” on page 26](#).

Event fields

An event field definition specifies the type and name of the field.

Event fields and variables are similar, but unlike variables, event fields cannot be initialized with a value.

Event fields that do not have the `wildcard` attribute are indexed by the correlator when you listen for them. There can be at most 32 indexes on an event type. Event fields of the type `location` use two indexes for each field.

An event that contains an `action`, `chunk`, `listener`, and/or `stream` field is valid only within the monitor that creates it. You cannot send, enqueue or route an event that contains, directly or indirectly, a field of such types.

Event actions

An event action is a subprogram or function that is associated with the event definition. It can be invoked or called from any monitor or from another action in the same event. Like monitor actions, the caller must supply actual parameters of the same type and number as the event action's formal parameters and if the action returns a value, then the return value must be consumed by the caller.

Like monitor actions, event actions can optionally be prefixed with annotations. See [“Annotations” on page 694](#).

Unlike monitor actions (see [“Monitor actions” on page 623](#)), events do not have the special actions `onload()`, `onunload()`, and `ondie()`.

Event action example:

```
action myEventAction(string s, location l) returns float {
    ...
    return 10.0;
}
```

See [“Defining actions” on page 250](#) for further information.

It is also possible to define static actions which apply only to the event type (and not to specific instances of an event). For example:

```
event E {
    static action someStaticAction(){
        print "I am a static action on event type E";
    }
}
```

See [“Defining static actions” on page 262](#) for further information.

Event action formal parameters

The formal parameters are a comma-separated list of parameter definitions, enclosed in parentheses. A parameter definition consists of a type name and an identifier. The identifier is the name of a parameter variable which will be bound to a copy of the value of an expression specified by the caller (that is, the value passed by the caller) when the action is invoked. The number and type of actual parameter values passed by a caller must match those listed in the action's formal parameters.

The scope of a parameter variable is the statement or block that forms the action body. Parameter variables are very similar to an action's local variables.

Event action return value

An event action return value specifies the return value type.

If the event action definition includes a `returns` clause, then the action returns a value of the specified type. All control paths within the action body must lead to a `return` statement before the end of the action body.

Event action body

The block construct forms the event action body. All variable references within an event action body must be one of the following:

- A field of the event
- A formal parameter of the action

- A local variable defined in the action body

Event field and action scope

The scope of an event's fields and actions is the same as the scope of the event itself except that the event fields are always referenceable within the event's actions.

Event templates

An event template is a construct that allows you to specify qualifying or matching criteria based on values of one or more of an event's fields. In event templates, you can qualify only on those event fields whose type is a primitive type. Event templates are used with `on` statements. See [“The `on` statement” on page 654](#).

An event template begins with the name of an event type that is to be matched. It can also start with `any`, in which case it will match against all events regardless of their type; see [“Listening for events of all types” on page 152](#).

Event templates can be either positional (see [“By-position qualifiers” on page 604](#)) or named (see [“By-name qualifiers” on page 605](#)) or a combination of both. Further, the criteria can be omitted entirely, in which case any event of the same event type will match. When both positional and named qualifiers are present in an event template qualifier expression list, the positional matches must come first.

Optionally, a colon and an identifier can follow the event expressions. This is called an event coassignment and specifies a variable whose value will become (that is, will be assigned) a reference to the matched event structure when the correlator detects a matching event and listener, and invokes the actions defined in the listener.

See also [“Stream source templates” on page 675](#).

By-position qualifiers

The correlator evaluates a positional event template against the event field that is at the same position in the event definition as the qualifier's position in the qualifier list.

For example, suppose an event has the fields shown below:

```
event sample1
{
    string  itemName;
    float   price;
    integer quantity;
}
```

An example of a by-position qualifier list for this event is as follows:

```
sample1 ("eggs", 0.50, 3)
```

This template matches `sample1` events that have an `itemName` value of `"eggs"`, a `price` value of `0.50`, and a `quantity` value of `3`.

In a by-position qualifier, an asterisk (*) matches any value of an event field in the corresponding position.

A range expression matches the event field values in the corresponding position to a low and high boundary value of the range. A match occurs when the field value is within the range. See [“Range expressions” on page 605](#).

The comparison operators < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), and = (equal to) specify a comparison of the event field value with the expression value that follows. A match occurs when the relation result is true. The expression to the right of the comparison operator cannot contain any references to the event's fields and must have a result type that is the same as the event field's type and must be one of `decimal`, `float`, `integer` or `string`.

By-name qualifiers

A by-name qualifier names an event field whose value is to be matched, instead of matching by position.

The identifier must be the name of one of the event's fields. The field's type must be `integer`, `decimal`, `float`, or `string`. Each event field is allowed to appear only once on the left side of a by-name qualifier, and the same field is not allowed in both a by-position qualifier and a by-name qualifier in the same event template.

An example of a by-name qualifier list is as follows (see the example in [“By-position qualifiers” on page 604](#) for the event fields that are also used for this example):

```
sample1 (itemName="eggs", price=0.50, quantity=3)
```

If the qualifier uses = *, then the qualifier matches all possible values of the specified event field.

If the qualifier uses one of the relational operators < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), and = (equal to), then the event field value is compared with the event template's value, and a match occurs when the result of the comparison is true.

If the qualifier uses `in` followed by a range expression, then the field is compared against the boundary values of the range.

The expression or range expression on the right side is not allowed to refer to any of the event's fields.

The expression or range expression is evaluated once, when the `on` statement containing the template is executed and its event expressions evaluated, not each time an event of the same type is processed by the correlator.

Range expressions

A range expression is a part of a qualifier expression that describes a range of consecutive `decimal`, `float`, `integer`, or `string` values between a low boundary and a high boundary. The correlator tests an event's field value against this range to determine whether or not it falls within the specified range.

The values for the low boundary and the high boundary are the expression values. Both expression values must be of the same type and one of decimal, float, integer, or string. Both expression types must be of the same type as the event field being tested. Neither expression can contain any references to the event's fields.

If the low boundary value is greater than the high boundary value, the EPL runtime automatically reverses them.

Example

In the following EPL, the three `on` statements specify event listeners that are all listening for the same range of events:

```
event test
{
    string s;
    float f;
}
monitor RangeExample
{
    test t;
    action onload()
    {
        on test (f > 9.0 ) and test (f <= 10.0)
        {
        }
        on test ("", (9.0 : 10.0])
        {
        }
        on test (f in (9.0 : 10.0])
        {
        }
    }
}
```

Depending on which of the starting operators, `[` or `(`, and ending operators, `]` or `)`, you use, the boundary values will either be included in the range or excluded from it.

- If the starting operator is `[`, then the low boundary value is included and candidate values greater than or equal to the low boundary value are in the range.
- If the starting operator is `(`, then the low boundary value is excluded and candidate values larger than the low boundary value are in the range.
- If the ending operator is `]`, then the high boundary value is included and candidate values less than or equal to the high boundary value are in the range.
- If the ending operator is `)`, then the high boundary value is excluded and candidate values lower than the high boundary value are in the range.

Note that you can have one kind of starting operator at the beginning and the other kind at the end; they do not need to match.

Field operators

Field operators can appear within event templates to define a field value.

The `on` keyword creates an event listener that watches the series of events processed by the correlator for individual events or patterns of particular events. You define the sequence of interest in an event expression made up of one or more event templates. The first part of an event template defines the event type of the event the event listener is to match against, while the section in brackets describes further filtering criteria that must be satisfied by the contents of events of that type for there to be a match. Event template field operators define what values, or range of values, are acceptable for a successful event match.

The value that a field operator applies to can be the result of an expression. Therefore, it is possible to have `>`, `<`, `>=`, `<=`, and/or `=` present in both their roles, as expression operators and as field operators, within an event template. This is not a problem, since the latter are unary while the former are binary and the semantics are quite different.

The following table describes the field operators:

Operator	Description
<code>[value1:value2]</code>	<p>Specifies a range of values that can match. The values themselves are included in the range to match against. For example:</p> <pre>on stockPrice(*, [0 : 10]) doSomething();</pre> <p>This example will invoke the <code>doSomething()</code> action if a <code>stockPrice</code> event is received where the price is between 0 and 10 inclusive. You can apply this range operator to decimal, float, integer and string types.</p>
<code>[value1:value2)</code>	<p>Specifies a range of values that can match. The first value itself is included in the range to match against while the second value is excluded from the range to match against. For example:</p> <pre>on stockPrice(*, [0 : 10)) doSomething();</pre> <p>This example will invoke the <code>doSomething()</code> action if a <code>stockPrice</code> event is received where the price is between 0 and 9 inclusive (assuming the field was of integer type). You can apply this range operator to decimal, float, integer and string types.</p>
<code>(value1:value2]</code>	<p>Specifies a range of values that can match. The first value is excluded from the range to match against while the second value is included. For example:</p> <pre>on stockPrice(*, (0 : 10]) doSomething();</pre> <p>This example invokes the <code>doSomething()</code> action if a <code>stockPrice</code> event is received where the price is between 1 and 10 inclusive (assuming the field was an integer). This operator can apply to decimal, float, integer and string types.</p>

Operator	Description
<code>(value1:value2)</code>	<p>Specifies a range of values that can match. The values themselves are excluded from the range to match against. For example:</p> <pre>on stockPrice(*, (0 : 10)) doSomething();</pre> <p>This example will invoke the <code>doSomething()</code> action if a <code>stockPrice</code> event is received where the price is between 1 and 9 inclusive (assuming the field was of <code>integer</code> type). You can apply this range operator to <code>decimal</code>, <code>float</code>, <code>integer</code> and <code>string</code> types.</p>
<code>> value</code>	<p>All values greater than the value supplied will satisfy the condition for a match. You can apply this operator to <code>decimal</code>, <code>float</code>, <code>integer</code>, and <code>string</code> types. When used with a <code>string</code>, the operator assumes lexical ordering.</p>
<code>< value</code>	<p>All values less than the value supplied will satisfy the condition for a match. You can apply this operator to <code>decimal</code>, <code>float</code>, <code>integer</code>, and <code>string</code> types. When used with a <code>string</code>, the operator assumes lexical ordering.</p>
<code>>= value</code>	<p>All values greater than or equal to the value supplied will satisfy the condition for a match. You can apply this operator to <code>decimal</code>, <code>float</code>, <code>integer</code>, and <code>string</code> types. When used with a <code>string</code>, the operator assumes lexical ordering.</p>
<code><= value</code>	<p>All values less than or equal to the value supplied will satisfy the condition for a match. You can apply this operator to <code>decimal</code>, <code>float</code>, <code>integer</code>, and <code>string</code> types. When used with a <code>string</code>, the operator assumes lexical ordering.</p>
<code>= value</code>	<p>All values equal to the value supplied will satisfy the condition for a match. You can apply this operator to <code>decimal</code>, <code>float</code>, <code>integer</code>, and <code>string</code> types. When used with a <code>string</code>, the operator assumes lexical ordering.</p>
<code>value</code>	<p>With one exception, only a value equivalent to the value supplied will satisfy the condition for a match. The exception is a <code>location</code> type field. A <code>location</code> value consists of a structure with four <code>floats</code> representing the coordinates of the corners of the rectangular space being represented. A listener that is watching for a particular value for a <code>location</code> field matches when it finds a <code>location</code> field that <i>intersects</i> with the <code>location</code> value specified in the listener's event expression. In the following example, the listener matches each <code>A</code> event whose <code>loc</code> field specifies a location that intersects with the square defined by <code>(0.0, 0.0, 1.0, 1.0)</code>.</p> <pre>location l := location(0.0, 0.0, 1.0, 1.0); on all A(loc = l) ...</pre>
<code>*</code>	<p>Any value for this field satisfies the condition for a match.</p>

Event listener definitions

You define an event listener in an `on` statement. See [“The `on` statement” on page 654](#).

Event lifecycle

An event enters the correlator in one of the following ways:

- An event is received from another component, such as the `engine_send` utility, an adapter, another correlator, or a process that is using the Apama client API. The correlator places the event on the input queue of each context that is subscribed to the channel on which the event is sent. If an event is not sent on a named channel then the correlator places the event on the input queue of each public context and each context that is processing a query.

Events sent on the `com.apama.queries` channel are put on the input queue of each context that is processing a query. These contexts automatically receive events sent on the `com.apama.queries` channel.
- A correlator pulls an event from a JMS message queue that is set up to distribute events to a cluster of correlators that is processing queries. The correlator adds the event to the input queue of each context that is processing queries.
- An EPL program creates an event instance and executes a `send...to` statement. If the target is a channel then the correlator places the event on the input queue of each context that is subscribed to that channel. If the target is a context (or a sequence of contexts) then the correlator places the event on the input queue of that context (or on the input queue of each context in the sequence).
- An EPL program creates an event instance and executes an `enqueue...to` statement. The correlator places the event on the input queue of the specified context or on the input queue of each context in the specified sequence of contexts.
- An EPL program creates an event instance and executes a `route` statement. The correlator places the event on the input queue of only the context that contains the monitor instance that routed the event.

Monitors

When the event gets to the front of the context's input queue, the correlator evaluates the event to determine if it is a match for any active event listeners in that context. That is, the correlator checks whether there are any event listeners in that context that are watching for that particular event. If there is a match, the match triggers the event listener. This means that the correlator executes the actions defined in the matching event listener.

It is possible for the actions defined in the event listener to route one or more events back to the context's input queue. A routed event goes right to the front of the context input queue. When the correlator is finished processing the event that triggered the event listener action, the correlator evaluates any routed events before it moves on to the event that was on the input queue after the matching event.

Queries

When the event gets to the front of the context's input queue, the correlator extracts the key of the event according to the definitions of running queries that use that event. The window of events for that key value is retrieved from the distributed cache. The correlator adds the event to the retrieved window, which it writes back to the cache. The event pattern of interest is evaluated against the stored window to determine whether the addition of the event causes a match set.

The event remains in its window until the correlator ejects it to make room for a new event or until the query instance or parameterization terminates.

Event listener lifecycle

When you inject a monitor into the correlator, the correlator instantiates the monitor in the main context and executes the monitor's `onload()` action. The `onload()` action typically specifies at least one `on` statement. An `on` statement includes an event expression that identifies the event or sequence of events that you are interested in. This is what you want to listen for. An `onload()` statement is not required to specify an `on` statement. If there is no `on` statement, the correlator immediately unloads the monitor.

When the correlator executes an `on` statement, it sets up an event listener for the specified event or sequence of events. After the correlator sets up the event listener, the event listener watches for an event that matches its event expression. When the event listener detects a matching event, the event listener triggers and the correlator executes the action specified in the `on` statement.

For an event listener that is looking for a single instance of an event, this is straightforward. However, the event expression that defines what you are looking for can specify all instances of an event, all instances of a sequence of events, and it can have temporal and logical constraints. This makes the lifecycle of an event listener less straightforward.

For example, consider the following event listener:

```
on all A() success();
```

When the correlator sets up this event listener, it sets up an event template to look for an A event. When an A event arrives, the correlator does the following:

- Executes the `success()` event listener action.
- Sets up a new event template to look for the next A event.

Now consider this event listener:

```
on all A() -> all B() success();
```

Again, suppose that the correlator sets up this event listener and an A event arrives. This time the correlator does the following:

1. Sets up an event template to listen for the next B event.
2. Sets up an event template to listen for the next A event.

This event listener will be active until it is explicitly killed because there will always be an event listener that is looking for the next A event.

Additional information about event listener lifecycles is in [“How the correlator executes event listeners” on page 167](#).

Event processing order for monitors

As mentioned earlier, contexts allow EPL applications to organize work into threads that the correlator can execute concurrently. When you start a correlator it has a main context. You can create additional contexts to enable the correlator to concurrently process events. Each context, including the main context, has its own input queue. The correlator can process, concurrently, events in each context.

Concurrently, in each context, the correlator

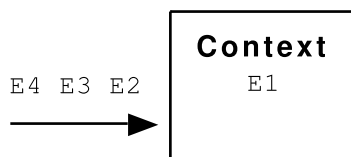
- Processes events in the order in which they arrive on the context's input queue
- Completely processes one event before it moves on to process the next event

When the correlator processes an event within a given context, it is possible for that processing to:

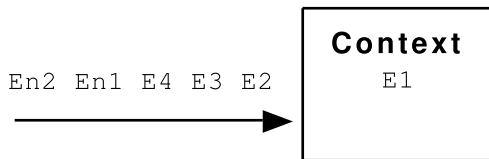
- **Send or enqueue an event to a particular channel.** The correlator places the event on the input queue of each context that is subscribed to the specified channel.
- **Send or enqueue an event to a particular context or to a sequence of contexts.** The correlator places the event on the input queue of the specified context or on the input queue of each context in the specified sequence.
- **Route an event.** The correlator places the routed event at the front of that context's input queue. The correlator processes the routed event before it processes the other events in that input queue.

If the processing of a routed event routes one or more additional events, those additional routed events go to the front of that context's input queue. The correlator processes them before it processes any events that are already on that context's input queue.

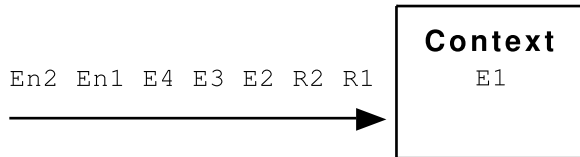
For example, suppose the correlator is processing the E1 event and events E2, E3, and E4 are on the input queue in that order.



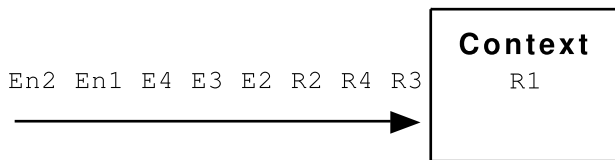
While processing E1, suppose that events E_{n1} and E_{n2} are created in that order and enqueued or sent to the context. Assuming that there is room on the input queue of that context, those events go to the end of the input queue of that context:



While still processing E1, suppose that events R1 and R2 are created in that order and routed. These events go to the front of the queue:

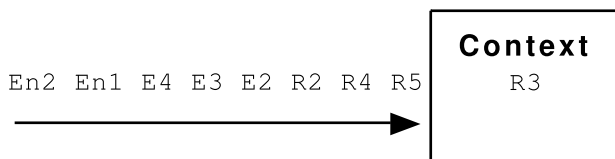


When the correlator finishes processing E1, it processes R1. While processing R1, suppose that two event listeners trigger and each event listener action routes an event. This puts event R3 and event R4 at the front of that context's input queue. The input queue now looks like this:



It is important to note that R3 and R4 are on the input queue in front of R2. The correlator processes all routed events, and any events routed from those events, and so on, before it processes the next routed or non-routed event already on the queue.

Now suppose that the correlator is done processing R1 and it begins processing R3. This processing causes R5 to be routed to the front of that context's input queue. The context's queue now looks like the following:



See also [“Understanding time in the correlator”](#) on page 179.

Event processing order for queries

Unlike EPL monitors, the order in which queries process events is not necessarily the order in which they were sent into the correlator. In particular, if two events that will be processed by the same query with the same key value are sent very close together in time (both events received less than about .1 seconds of each other) then they may be processed as if they had been sent in a different order. For example, consider a query that is looking for an A event followed by an A event. If two A events with the same key arrive 1 millisecond apart then the events might not be processed in the order in which they were sent.

Queries use multiple threads to process events and to scale across multiple correlators on multiple machines. To do this efficiently, there is no enforcement that the events are processed in order. However, when events that have the same key arrive roughly about .5 seconds apart or more then out-of-order processing is typically avoided provided the system can keep up with the load. Therefore, you want to specify a query so that it operates on partitions in which the arrival of consecutive events is spaced far enough apart. For example, consider a query that operates on credit card transaction events, which could mean thousands of events per second. You want to partition this query on the credit card number so that there is one event or less per partition per second. By following this recommendation, it becomes possible to process events that are generated at rates of up to 10,000 events per second.

When creating an evt file for testing purposes, the recommendation is to begin the file with a `&FLUSHING(1)` line to cause more predictable and reliable event-processing behavior. See "Event timing" in the "Correlator Utilities Reference" section of *Deploying and Managing Apama Applications*.

Event expressions

An event expression is a special type of expression that is used with the `on` statement to define the rules for detecting events of interest and invoking an action when a matching event is detected. In an event expression, you can specify filtering rules based on an event's field values, sequencing rules for events followed by other events, times and time ranges during which an event is of interest, and other rules. See also ["The on statement" on page 654](#).

Event expressions should not be confused with ordinary EPL expressions of type `event`. Ordinary EPL expressions of all types are described in ["Expressions" on page 659](#).

Event primaries

The event primary is the simplest form of an event expression clause and can be combined with other event primaries and event operators to form more complex event expressions.

An event primary can be an event template (see ["Event templates" on page 604](#)) optionally prefixed with `completed` or `unmatched`, or it can be a timer (see ["Timers" on page 615](#)).

Event templates are constructs that allow you to specify filtering or matching criteria based on values of one or more of an event's fields.

The completed operator

A `completed` event template matches only after all other work is completed. When an event that matches a `completed` template comes into the correlator, the correlator:

1. Runs all of the event's normal or `unmatched` event listeners. Normal event templates do not specify the `completed` or `unmatched` keyword.
2. Processes all routed events that result from those event listeners.
3. Triggers the `completed` event listeners.

For example:

```
on all completed A(f < 10.0) {}
```

The unmatched operator

An unmatched event template matches against events for which both of the following are true:

- Except for `completed` and `unmatched` event templates, the event is not a match with any other event template currently loaded in the context.
- The event matches the unmatched event template.

The correlator processes events as follows:

1. The correlator tests the event against all normal event templates in the context. Normal event templates do not specify the `completed` or `unmatched` keyword. If there are any matches, those event listeners trigger and the correlator executes those event listener actions. If execution of the event listener actions routes any events, the correlator then processes those events.
2. If the correlator does not find a match, the correlator tests the event against all event templates in the context that specify the `unmatched` keyword. If the correlator finds one or more matches, it triggers an event listener for each match found. In other words, if multiple unmatched event templates match a given event, they all trigger. The correlator executes the event listener actions defined by the event listeners that trigger. If any events are routed during execution of those actions, the correlator processes the routed events.
3. The correlators tests the event against all event templates in the context that specify the `completed` keyword. If the correlator finds one or more matches, it triggers an event listener for each match found.

Example

For example, suppose you have the following code:

```
on all A("foo", < 10) as a {  
    print "Match: " + a.toString();  
    a.count := a.count+1; // count is second field of A  
    route a;  
}  
on all unmatched A(*,*) as a {  
    print "Unmatched: " + a.toString();  
}  
on all completed A("foo", *) as a {  
    print "Completed: " + a.toString();  
}
```

The incoming events are as follows:

```
A("foo", 8);  
A("bar", 7);
```

The output is as follows.

```
Match: A("foo", 8)  
Match: A("foo", 9)  
Unmatched: A("foo", 10)
```

```
Completed: A("foo", 10)
Completed: A("foo", 9)
Completed: A("foo", 8)
Unmatched: A("bar", 7)
```

Specify the unmatched keyword with care. Be sure to communicate with any others who write event templates. If you are relying on an unmatched event template, and someone else injects a monitor that happens to match some events that you expected to match your unmatched event template, you will not get the results you expect. The firing of an event-specific unmatched listener is suppressed if an `on any()` listener matches the event (not just a type-specific listener).

Parenthesized event expressions

Just as with primary and bitwise expressions, event expressions can be enclosed in parentheses to control expression evaluation order or to improve readability.

Timers

Specify a timer with the `wait`, `at`, or `within` keyword. For more detailed information, see [“Defining event listeners with temporal constraints” on page 174](#).

The wait event operator

The `wait` operator can be used to limit the amount of time that an event listener can match an event. The `wait` operator's expression specifies the time in seconds. The result of evaluating the `wait` expression must be of type `float`.

See also [“Waiting within an event listener” on page 176](#).

The at event operator

The `at` operator allows triggering of an event listener at a specific time or repeatedly at multiple times, depending on how the series of expressions that follow the `at` operator are constructed.

The time specification of the `at` operator consists of either five or six expressions, corresponding to the number of minutes of the hour (0 to 59), hour of the day (0 to 23), day of the month (1 to 31), month of the year (1 to 12), day of the week (0 to 6, 0=Sunday), and seconds respectively.

If the optional number of seconds is omitted, 0 is used.

The `*` operator means that all times (minute, hour, etc.) for the corresponding part of the time specification will match.

You can specify one or more time values separated by commas.

See also [“Triggering event listeners at specific times” on page 177](#).

The within operator

The `within` operator takes one operand, which is an expression of type `float`, whose value is the number of elapsed seconds from an event primary's activation time that the event primary can be

matched. The `within` operator's result type is `boolean`. If the event is matched before the specified time has elapsed, the `within` operator's result is `true`. When the time has elapsed and the event has not been matched, the `within` operator's result is `false`.

See also [“Listening for event patterns within a set time” on page 175](#).

The not operator

The `not` operator specifies logical negation.

Example:

```
on A() and not B() executeAction();
```

The all operator

When the `all` operator appears before an event template, when that event template finds a match, it continues to watch for subsequent events that also match the template.

Consider the following event expression:

```
all A -> B
```

This event listener would match on every `A` and the first `B` that follows it. The way this works is that upon encountering an `A`, the correlator creates a second event listener to seek the next `A`. Both event listeners would be active concurrently; one looking for a `B` to successfully match the sequence specified, the other initially looking for an `A`. If more `As` are encountered the procedure is repeated; this behavior continues until either the monitor or the event listener are explicitly killed.

Consider the following sequence of incoming events:

```
C1 A1 F1 A2 C2 B1 D1 E1 B2 A3 G1 B3
```

With these input events, `on all A() -> B()` would return the following:

{`A1, B1`}, {`A2, B1`} and {`A3, B3`}.

Note that `all` is a unary operator and has higher precedence than `->`, `or` and `and`.

The and, xor, and or logical event operators

The logical operators `and`, `xor`, and `or` are binary operators, operating on event expressions that are either side of them. They are similar to the corresponding operators in primary and bitwise expressions, but do not have quite the same precedence. See also [“Event expression operator precedence” on page 617](#).

Operator	Description
<code>and</code>	Logical intersection
<code>xor</code>	Logical exclusive or

Operator	Description
or	Logical or

The followed-by event operator

The followed-by operator `->` takes left and right operands, both event expressions. The followed-by operator waits for the left operand to become true and then waits for the right operand to become true. When both are true, then the result value is `true`. If either becomes false, then the result value is `false`.

Event expression operator precedence

The following table lists the event expression operators in order by their precedence, from lowest to highest. See [“Expression operator precedence” on page 669](#) for a corresponding table of primary and bitwise expression operator precedence.

Operation	Operator	Precedence
Followed-by	<code>-></code>	1
Logical union	<code>or</code>	2
Logical exclusive or	<code>xor</code>	3
Logical intersection	<code>and</code>	4
All	<code>all</code>	5
Logical negation	<code>not</code>	6

For clarity, it is strongly recommended to use brackets in event expressions. This makes it very easy to understand what an expression means. Do not use un-bracketed expressions, except where trivial.

For example, the following expression:

```
on all A() or B() and not C() -> D()
```

is equivalent to this expression, which is much easier to understand:

```
on (
  (all A() )
  or
  (B() and (not C() ))
) -> D()
```

Event channels

Adapter and client configurations can specify the channel to deliver events to. A channel is a string name that contexts and receivers can subscribe to in order to receive particular events. In EPL, you can send an event to a specified channel. Sending an event to a channel delivers it to any contexts that are subscribed to that channel, and to any clients or adapters that are listening on that channel.

You can use the `com.apama.Channel` type to send an event to a channel or context. The `Channel` type holds a string or a context. When it holds a string an event is sent to the channel that has that name. When it holds a context an event is sent to that context.

The default channel is the empty string. Events sent to the default channel and events sent without a channel specification are added to the input queue of each public context as well as each context that is processing queries.

You can use the `com.apama.queries` channel to send events to all contexts that process queries.

35 Monitors

■ Monitor lifecycle	620
■ Monitor files	621
■ Packages	621
■ The using declaration	622
■ Monitor declarations	622
■ The import declaration	622
■ Monitor actions	623
■ Contexts	624
■ Plug-ins	625
■ Garbage collection	625

A `.mon` file is a file that contains the source text for an optional package specification and one or more event declarations and/or monitor definitions. A file can consist entirely of event declarations without any monitors.

Note:

Monitors and queries are the two main EPL programming units. A monitor cannot contain a query. A query cannot contain a monitor. Each unit offers a different approach to event processing. See "Architectural comparison of queries and monitors" in *Introduction to Apama*.

A monitor is a group of related variable declarations and actions. An action is a group of related variable declarations and statements. An action can either be part of a monitor or part of an event declaration.

The executable statements (except for global variable initializers) are always inside an action. An action can be either a subprogram or a function. The difference is that a function has a return value and a subprogram does not.

Each file is injected whole or not at all; if some parts compile validly but others do not, nothing is injected and an error is returned. Injecting can also return warnings about the code injected. For example, use of keywords that may be reserved in the future.

Monitor lifecycle

Monitors are compiled and run (executed) by the Apama correlator. The correlator starts executing in the monitor's `onload()` action. To execute a monitor, you load (inject) it into the correlator. The correlator then does the following:

1. Compiles the monitor's source text
2. If no errors are detected, creates the main monitor instance along with its global variables
3. Invokes the monitor instance's `onload()` action

When the `onload()` action has executed to completion (that is, the control path reaches the closing curly brace of the `onload()` action), if the monitor instance has event listeners or streaming networks, then it remains active but in a suspended state.

The correlator calls the monitor instance's event listeners whenever it detects events that match the event listeners' event expressions.

A monitor instance terminates when one of the following events occurs:

- The monitor instance executes a `die` statement in one of its actions.
- A runtime error condition is raised.
- The monitor is terminated externally (for example, with the `engine_delete` utility).
- The monitor instance has executed all its code and there are no remaining listeners or streaming networks. This will occur rapidly if the `onload()` action does not create any.

When a monitor instance terminates, the correlator does the following:

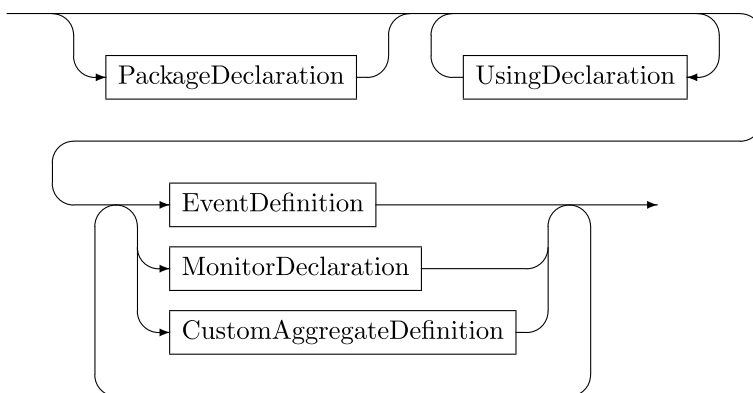
1. Invokes the monitor instance's `ondie()` action, if it is defined.
2. If the monitor instance that is terminating is the last active instance of that monitor, the correlator also does the following:
 - Invokes the monitor's `onunload()` action if it is defined.
 - Removes the monitor's code from the correlator.
 - Frees all the monitor's resources.

To summarize, consider that when a monitor spawns monitor instances, there is a set of monitors that includes the original monitor instance and any spawned monitor instances. As the monitor instances in this set terminate, the correlator calls the `ondie()` action, if it is defined, for each monitor instance that terminates. When the last monitor instance in the set terminates, the correlator also calls the `onunload()` action. Thus, the correlator calls `ondie()` once for each monitor instance in the set, and calls `onunload()` only once for the entire set.

See [“About executing `ondie\(\)` actions” on page 44](#) for information about how `ondie()` can optionally receive exception information if an instance dies due to an uncaught exception.

Monitor files

An EPL monitor file contains an optional package declaration, optional using declaration, event declarations and/or monitor declarations and/or custom aggregate definitions.



Packages

A package declaration provides a scope for events and/or monitors, and/or queries.

Example:

```
package com.myCorporation.myproject;
```

See [“Names” on page 693](#) for further information.

The using declaration

The `using` declaration lets you use a type in a package other than the package the type was defined in without having to specify the fully qualified name of the type.

Insert a `using` declaration (after the optional package declaration and before any other declarations) that specifies the fully qualified name of the type. For example:

```
using com.myCorporation.custom.myCustomAggregate;
```

You can specify multiple `using` declarations in a file.

In a file, you cannot specify two `using` declarations that bring in types that have the same base name. See also [“Name Precedence” on page 693](#).

You cannot specify a `using` declaration for named objects such as monitors, JMon monitors, and namespaces.

A `using` declaration can be in a monitor or in a query.

Monitor declarations

Specify `persistent` when you want a persistence-enabled correlator to save the state of the monitor in a recovery datastore on disk. In a monitor, `import` declarations, event declarations, variable declarations, and action definitions can be freely mixed in any order. For detailed information, see [“Defining Monitors” on page 33](#).

A monitor can be optionally prefixed (before the `persistent` keyword) with annotations. See also [“Annotations” on page 694](#).

The import declaration

The `import` declaration loads a plug-in library and makes it available to an EPL program. Plug-in libraries are shared libraries on Linux and UNIX systems and Dynamic Link Libraries on Windows systems.

On Linux and UNIX systems, the library is loaded from a `libPlugInName.so` file located in one of the directories listed in the environment variable `LD_LIBRARY_PATH`. On Windows, the library is loaded from a `PlugInName.dll` file located in the `bin` folder.

You can name a plug-in. The plug-in name is a library filename, not a full filepath, and is not allowed to contain any of the characters used as directory or device separators (forward slash, colon, or backslash).

You can also give the plug-in an identifier (an alias name) for use in the EPL program when you call the library's actions.

For example, to call a plug-in action `foo()` in the plug-in library `wffftl.so` or `wffftl.dll`, you would write the following:

```

monitor m {
    import "wffftl" as fft;
    action onload()
    {
        sequence <float> data := [];
        fft.foo (data);
    }
}

```

For detailed information, see [“Using EPL Plug-ins” on page 339](#).

Monitor actions

Monitors can have two forms of actions: simple actions and actions with parameters and/or return values. These types of actions are discussed in the topics below.

Monitor actions can optionally be prefixed with annotations. See [“Annotations” on page 694](#).

Simple actions

A simple action has a name and a body consisting of a block. The body contains the executable code of the action. There are no parameters.

The action names given in the table below have special meaning in a monitor. These actions are invoked automatically when certain events in a monitor's life cycle occur. Apama recommends that you do not use these names in queries.

A block must follow the action name. Note that there are no formal parameters in this form of action definition and the action cannot return a value.

Action	Description
<code>onload()</code>	This action is invoked immediately after a monitor has been loaded. This action must be present in every monitor.
<code>ondie()</code>	If present, this action is invoked by the correlator when a monitor instance terminates.
<code>onunload()</code>	If present, this action is invoked by the correlator after all instances of a monitor have terminated, just before the last monitor instance is unloaded.
<code>onBeginRecovery()</code>	If present, this action is invoked by the correlator during recovery of a persistence-enabled correlator. The correlator executes <code>onBeginRecovery()</code> on monitors and any live events after it reinjects source code and restores state in persistent monitors.
<code>onConcludeRecovery()</code>	If present, this action is invoked by the correlator during recovery of a persistence-enabled correlator. The correlator

Action	Description
	executes <code>onConcludeRecovery()</code> on monitors and any live events before it begins to send clock ticks.

Actions with parameters

An action can take an optional list of parameters.

Formal parameters

The formal parameters are a comma-separated list of type name and identifier pairs.

The identifier is the name of a parameter variable that will be bound to a copy of the value of an expression specified by the caller (that is, the value passed by the caller) when the action is invoked. The number and type of actual parameters passed by a caller must match those listed in the action's formal parameters.

The scope of a parameter variable is the statement or block that forms the action body. Parameter variables are very similar to an action's local variables.

Action return value

If you specify a `returns` clause, then the action must return a value whose type matches that specified in the `returns` clause. You specify the return value by using a `return` statement and result expression within the action. Every control path (see [“Transfer of control statements” on page 656](#)) within the action body must lead to a `return` statement with a result expression of the correct type.

Action body

After the `returns` clause (or after the formal parameters if there is no `returns` clause), a statement forms the action body. The action body can be a single statement or a block.

Within the action body, you use the parameter variable names to obtain the values that are passed to the action by its caller.

Contexts

Contexts allow EPL applications to organize work into threads that the correlator can concurrently execute. For detailed information, see [“Implementing Parallel Processing” on page 285](#). This also provides information on the properties of a context (see [“About context properties” on page 287](#)).

Note:

In monitors, you must implement the use of contexts. In queries, the use of contexts is automatically done for you.

You can create any number of contexts. Creating a context just allocates an ID and creates a small object. See also [“Creating contexts” on page 288](#).

For information on how to obtain a reference to a context, see [“Obtaining context references” on page 290](#).

Plug-ins

EPL can be extended through the use of plug-ins, which are modules written either in C++, C, or Java and loaded dynamically into the EPL runtime with the `import` statement. Plug-in modules are invoked in exactly the same way as actions in an EPL event.

See [“Using EPL Plug-ins” on page 339](#).

Garbage collection

EPL, like languages such as Java or C#, relies on garbage collection. Intermittently, the correlator analyses the events that have been allocated, including dictionaries, sequences, closures and streaming networks, and allows memory used by events that can no longer be referenced to be re-used. Thus, the actual memory usage of the correlator might be temporarily above the size of all live objects. While running EPL, the correlator might wait until a listener or `onload()` action completes before performing garbage collection. Therefore, any garbage generated within a single action or listener invocation might not be disposed of before the action/ listener has completed. It is thus advisable to limit individual actions/listeners to performing small pieces of work. This also aids in reducing system latency.

The cost of garbage collection increases as the number of events a monitor instance creates and references increases. If latency is a concern, it is recommended to keep this number low, dividing the working set by spawning new monitor instances if possible and appropriate. Reducing the number of event creations, including string operations that result in a new string being created, also helps to reduce the cost of garbage collection. The exact cost of garbage collection could change in future releases as product improvements are made.

36 Queries

■ Query lifetime	628
■ Query definition	630
■ Metadata section	631
■ Parameters section	632
■ Inputs section	632
■ Query input definition	632
■ Find statement	634

An Apama query is a self-contained processing element that communicates with other queries, and with its environment, by sending and receiving events. Queries are designed to be multithreaded and to scale across machines.

Note:

Queries and monitors are the two main EPL programming units. A query cannot contain a monitor. A monitor cannot contain a query. Each unit offers a different approach to event processing. See "Architectural comparison of queries and monitors" in *Introduction to Apama*.

You use Apama queries to find patterns within, or perform aggregations over, defined sets of events. For each pattern that is found, an associated block of procedural code is executed. Typically this results in one or more events being transmitted to other parts of the system.

A query is defined in a `.qry` file. A query finds specified event patterns or aggregates event values.

Apama queries are useful when you want to monitor incoming events that provide information updates about a very large set of real-world entities such as credit cards, bank accounts, or cell phones. Typically, you want to independently examine the set of events associated with each entity, that is, all events related to a particular credit card account, bank account, or cell phone. A query application operates on a huge number of independent sets with a relatively small number of events in each set.

The following topics provide reference information for the parts of a query definition. For user guide type information, see ["Defining Queries" on page 59](#).

Query lifetime

You inject queries into a running correlator with the Apama macros for Ant, (`install_dir\etc\apama-macros.xml`) or with Software AG Designer. You can delete queries from a running correlator by performing a delete operation and specifying a query name. You can use the same tools that you use to delete monitors: `engine_delete` utility, Software AG Designer, Apama macros for Ant (`apama-macros.xml`), or `deleteName()` method on the engine client API.

If you are using a cluster of correlators, it is your responsibility to inject each query into each correlator in the cluster, and to delete a query from each correlator in a cluster. This keeps deployed queries in sync across the cluster. In other words, injecting or deleting a query on one host in a cluster does not automatically inject or delete the query on the other cluster members.

Unlike monitors, the lifetime of query instances is either automatic (for non-parameterized queries) or controlled by the Scenario Service (for parameterized queries, see "Scenario Service API" in *Connecting Apama Applications to External Components*). There are no `spawn` or `die` equivalents in queries, and you cannot use these EPL statements in queries.

When a non-parameterized query is injected, a single instance of the query is automatically created at injection time and it begins processing events. You cannot use the Scenario Service API to edit or delete this single instance or to create new instances. For parameterized queries, after injection, only the query definition is created automatically. The query does not start processing events specified in its `inputs` section until at least one parameterization is created by means of the Scenario Service. You can control this by using a dashboard or scenario browser. The Scenario Service has methods to create new query instances, edit instances and delete instances.

When using a cluster of correlators, the parameterizations are kept in sync across all members of the cluster. Creating a query instance while connected to one cluster member will create it on all members. The instance can be edited or deleted by any client connected to any member. There may be short delays in replicating parameterization data on each cluster member because this happens asynchronously. However, the recommendation is to edit or delete a particular parameterization from Scenario Service clients that are all connected to the same correlator. This ensures that edit and delete operations are performed in the same order on every cluster member. If you try to edit or delete the same parameterization from different cluster members the results are unpredictable.

If a query executes code in a `where` clause, aggregate or other expression that results in an exception due to the current values in the window, the query ignores the exception and continues running. For example, an attempt to divide an integer by zero causes an `ArithmeticException`. If a query experiences an exception that means it cannot continue (such as repeated exceptions while trying to retrieve or store window data), then the query instance will enter the failed state, which will be reported by the Scenario Service. In this case, the query does not process additional events. The correlator log file should contain information that explains why the query failed. The problem that caused the failed state needs to be corrected. After correcting the problem, if the query is a parameterized query, you should delete the failed parameterization and then re-create it. For a non-parameterized query, you must delete and then re-inject the query.

When a query is deleted with the `engine_delete` utility or equivalent, all instances of the query are terminated and the Scenario Service will reflect that the query definition has been unloaded. The query can be re-injected, if needed. Remember that deletions and injections must be performed on every member in a cluster.

Lifetime of find statements

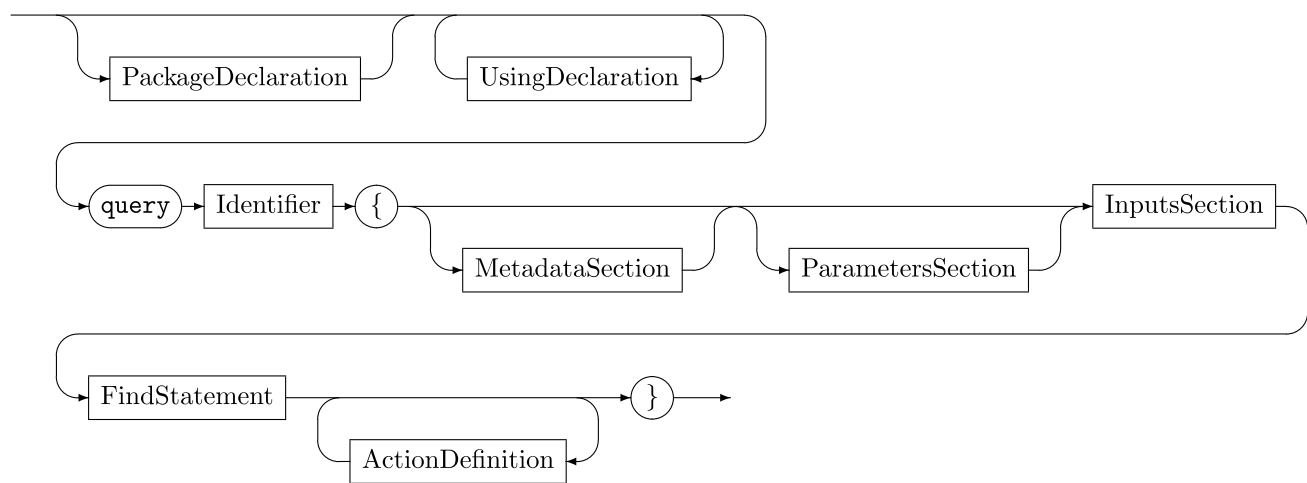
As long as a query is active, the `find` statement in a query is active for each value of the key that is specified in the query's `inputs` section. Thus, `find A as a` in a query is similar to `on all A() as a` in a monitor. The `find` statement generates a match set each time the latest event causes a match. If the `find` statement specifies any aggregates and the `every` modifier, which can only be used with aggregates, then each new match set causes the `find` statement to add to the aggregate.

In monitors, listeners can match either the first set of matching events, or specify the `all` operator to fire for every set of matching events. For example, `on A() as a -> B() as b` fires on the first A and B events, while `on all A() as a -> all B() as b` fires for every combination of an A event with a later B event. In a query, `find A as a -> B as b` fires on every B event after an A event if an A event is still in the window defined in the `inputs` section. The match set contains the most recent A event and the most recent B event. The following table provides examples. The assumption is that all input events remain in the query's window.

Input events	Query match sets for: find A as a -> B as b	Query match sets for: find every A as a -> B as b select... - inputs to aggregates	Monitor match sets for: on A() as a -> B() as b	Monitor match sets for: on all A() as a -> all B() as b
A(1)				
B(1)	A(1), B(1)	A(1), B(1)	A(1), B(1)	A(1), B(1)
A(2)				
B(2)	A(2), B(2)	A(1), B(1) A(1), B(2) A(2), B(2)		A(1), B(2) A(2), B(2)
B(3)	A(2), B(3)	A(1), B(1) A(1), B(2) A(2), B(2) A(1), B(3) A(2), B(3)		A(1), B(3) A(2), B(3)

Query definition

A query searches for an event pattern that you specify. You define a query in a file with the extension “.qry”. Each .qry file contains the definition of only one query.



If specified, any package or using statements must be before the query declaration. See “[Packages](#)” on page 621 and “[The using declaration](#)” on page 622.

You must specify an identifier for the query name. See [“Identifiers” on page 687](#). The convention for specifying the name of a query is to use UpperCamelCase, as shown in the example below.

Specification of metadata is optional. See [“Metadata section” on page 631](#). The convention for specifying the key in the key-value pair of the metadata is to use lowerCamelCase as shown in the example below.

Specification of query parameters is optional. See “Parameters section” on page 632.

An `inputs` section is required. It specifies at least one event type. These are the event types that the query operates on. See [“Inputs section” on page 632](#).

The `find` statement is required. It specifies the event pattern of interest and a block that contains procedural code. See “[Find statement](#)” on page 634.

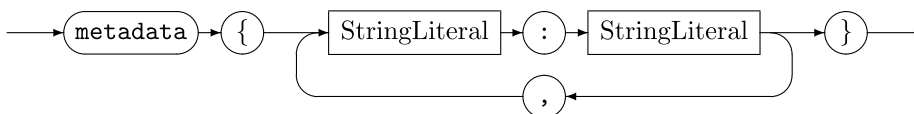
Action definitions, in the same form as actions in events, are optional. See “Event actions” on page 602.

Example:

```
query ImprobableWithdrawalLocations {
  metadata {
    "author": "Apama",
    "version": "1"
  }
  parameters {
    float period;
  }
  inputs {
    Withdrawal() key cardNumber within (period);
  }
  find
    Withdrawal as w1 -> Withdrawal as w2
    where w2.country != w1.country {
      log "Suspicious withdrawal: " + w2.toString() at INFO;
    }
}
```

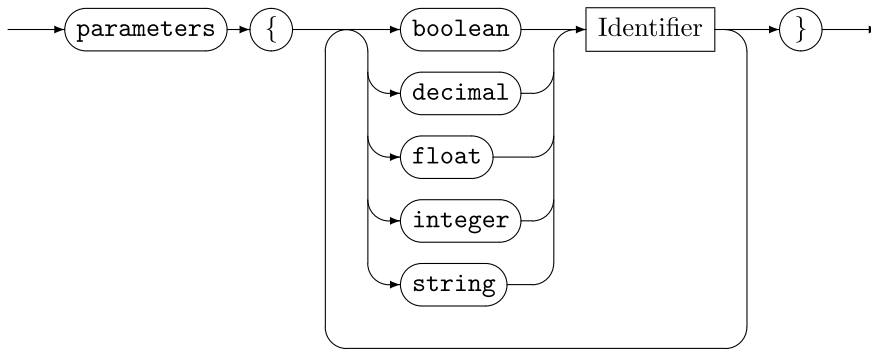
Metadata section

In a query, the optional metadata section specifies a list of key-value pairs. If there is a metadata section, it must be the first section in the query. See [“Defining metadata in a query”](#) on page 70 for further information.



Parameters section

In a query, the optional parameters section specifies any parameters used by the query. If there is a parameters section, it must follow the metadata section, if defined, and it must precede the inputs section. Parameter values are available throughout a query. See [“Implementing parameterized queries” on page 129](#) for further information.



Inputs section

In a query, the required inputs section specifies the events that the query operates on.

At least one input definition is required. Typically, no more than four input definitions are specified.

If there is a parameters section, then the inputs section follows it. The inputs section must be before the find statement.

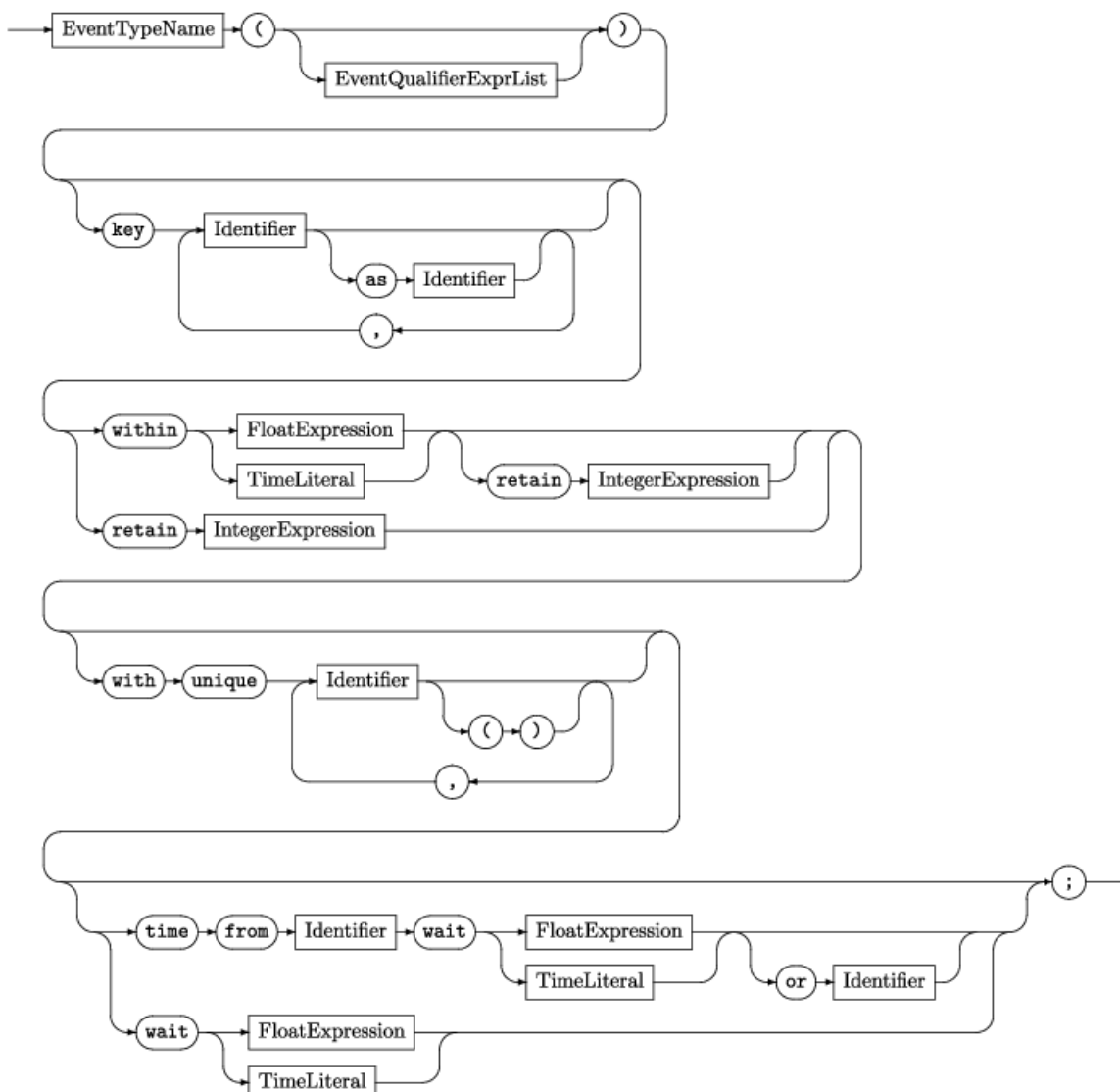
Example:

```
inputs {
  A() key k retain 20;
  B() key k retain 10;
}
```

For more information, see [“Defining query input” on page 77](#).

Query input definition

In a query, the required inputs section must contain at least one input definition.



An event type you specify must be parseable. See [“Type properties summary” on page 587](#). Event type names can come from the root namespace, a `using` declaration, or a local package as specified in a package declaration.

Event filters are optional. Specifying a filter here determines which events are added to a query window. The rules for what you can specify for the event filter are the same as for what you can specify in an event template in EPL. See [“Event templates” on page 604](#).

Specification of a key is optional, but rarely omitted. If there is no key specification, all events are in one partition. The correlator uses the key to partition events. Each partition is identified by a unique key value. Specify one or more fields that are in the input event type. One or two fields in a key is typical. Three fields in a key is unusual and rarely needed. More than three fields is discouraged. If you define more than one input in a query

- The number, type, and order of the key fields in each input definition must be the same.

- If the names of the key fields are not the same in each input definition, you must insert the `as` keyword to specify aliases so that the names match. For details, see [“About keys that have more than one field” on page 76](#).

A `retain` clause or a `within` clause is required. Alternatively, you can specify both.

A `retain` clause indicates how many events to hold in the window. Follow the `retain` keyword with a positive integer. If you specify a negative integer or zero, it is a runtime error that terminates the query.

A `within` clause indicates the length of time that an event stays in the window. Follow the `within` keyword with a positive float expression or a time literal. If you specify a negative float value or zero it is a runtime error that terminates the query.

For information on other clauses, see [“Format of input definitions” on page 80](#).

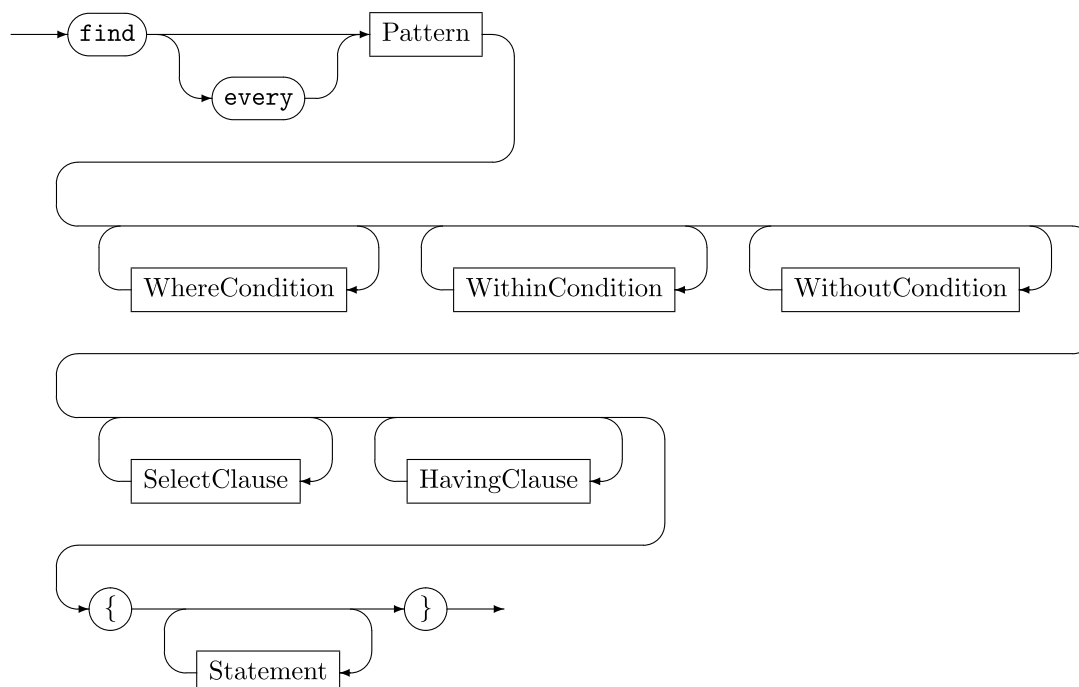
Examples:

```
inputs {
  Withdrawal(amount > 500) key userId within 1 hour;
}
```

```
inputs {
  APNR() key road within(150.0);
  Accident() key roadName as road within(10.0);
}
```

Find statement

A query `find` statement tries to find a match for the event pattern that the `find` statement specifies. When the query finds a match it executes the EPL in the `find` statement block.



When a `find` statement specifies a `select` or `having` clause, the `every` modifier is required. Conversely, you cannot specify the `every` modifier if you do not specify a `select` or `having` clause.

When a `find` statement specifies the `every` identifier, the identifiers in the `select` clause are available in the `having` clause and in the `find` block, but the coassignments in the pattern are not available.

Pattern coassignments are available in a `where` clause that applies to the pattern.

When you do not specify the `every` modifier, all pattern coassignments, except a `without` clause coassignment, are available in the `find` block.

In a `where` clause that is part of a `without` clause, pattern coassignments as well as the coassignment in the `without` clause are available.

Example:

```
find Withdrawal as w1 -> Withdrawal as w2
  where w1.country = "UK" and w2.country = "Narnia" {
    // Recent card fraud in Narnia against UK customers
    emit SuspiciousWithdrawal(w2);
  }
```

Pattern

In a query definition, the `find` statement specifies the event pattern of interest followed by a procedural block that specifies what you want to happen when a pattern match is found.

A coassignment variable specified in an event pattern is within the scope of the `find` block and it is a private copy in that block. Changes to the content that the variable points to do not affect any values outside the query. Unlike EPL event expressions, you need not declare this identifier before you coassign a value to it.

In an event pattern in a `find` statement, each coassignment variable identifier must be unique. You must ensure that an identifier in an event pattern does not conflict with an identifier in the `parameters` section or `inputs` section.

If a pattern specifies a `wait` operator, then it must be at the beginning of a pattern, at the end of a pattern, or both. It cannot be in the middle of a pattern. The followed-by operator (`->`) must be after or before each instance of the `wait` operator. For example:

```
wait(1) as w -> (A as a and B as b)           // Allowed
{(A as a and B as b) -> wait(1) as w }        // Allowed
wait(1) as w1 -> (A as a and B as b) -> wait(1) as w2 // Allowed
wait(1) as w and A as a                       // Not allowed
A as a -> wait(1) as w -> B as b              // Not allowed
```

A `wait` operator must specify a positive `float` value or a time literal. A `float` value always indicates a number of seconds.

Optionally, specify `and`, `or` or `->` and then specify an *event_type* and coassignment variable. Parentheses are allowed in the pattern specification and you can specify multiple operators, each followed by an *event_type* and coassignment variable. For example, the following is a valid `find` statement:

```
find (A as a1 -> ((A as a2)) -> (A as a3) ->
      (A as a4 -> A as a5 -> A as a6) ->
      ((A as a7) -> A as a8) -> A as a9) -> A as a10 {
    print "query with 10: "+a1.toString()+ " - "+a10.toString();
}
```

Where condition

A find statement can specify a where clause that filters which events match the specified event pattern.

Note:

You can specify a find where clause that applies to the event pattern, and you can also specify a without where clause that is part of a without clause. Any where clauses that you want to apply to the event pattern must precede any within or without clauses.

Specify the where keyword followed by a Boolean expression that refers to the events you are interested in. The Boolean expression must evaluate to true for the events to match.

The where clause is optional. You can specify zero, one or more where clauses.

Coassignment variables specified in the find or select statements are in scope in a find where clause. Also available in a find where clause are any parameter values and key values.

Example:

```
find Withdrawal as w1 -> Withdrawal as w2
  where w2.country != w1.country {
    log "Suspicious withdrawal: " + w2.toString() at INFO;
}
```

Within condition

In a find statement, a within clause sets the time period during which all events in the match set or some events in the match set must have been added to their windows.

A pattern can specify zero, one, or more within clauses. These must appear after any find where clauses and before any without clauses.

Specify the within keyword followed by a float expression or a time literal, which indicates the time period during which the events in the match set must be received.

Optionally, specify a between clause to indicate that the time constraint applies to only some of the events in the match set. See [“Between clause” on page 638](#).

Example:

```
find LoggedIn as lc -> OneTimePass as otp
  where lc.user = otp.user
  within 30.0 {
    emit AccessGranted(lc.user);
}
```

Without condition

In a `find` statement, a `without` clause specifies an event type whose presence prevents a match.

Specify the `without` keyword followed by an event type coassigned to an identifier.

An event type that you specify in a `without` clause must be specified in the `inputs` block of the query. A pattern can specify zero, one, or more `without` clauses.

Optionally, after each `without` clause, you can specify one `where` clause, which is referred to as a `without where` clause to distinguish it from a `find where` clause. When a `where` clause is part of a `without` clause:

- The Boolean expression must evaluate to `true` for the presence of the specified event to prevent a match. In other words, when the Boolean expression evaluates to `false`, then there can be a match even when the specified event is in the window.
- The `where` clause applies to the event specified in its `without` clause.
- The Boolean expression can refer to parameters, coassignment identifiers in the event pattern, and the coassignment identifier in the `without` clause.

A `without` clause cannot use the `->` or `and` pattern operators. However, you can specify multiple `without` clauses. If there are multiple `without` clauses each one can refer to only its own coassignment and not coassignments in other `without` clauses. However, all `without` clauses can make use of the pattern's standard coassignments.

If there are multiple `without` clauses, a matching event for any one of them prevents a pattern match. Multiple `without` clauses can use the same type and the same coassignment, which is useful only when their `where` conditions are different.

Typically, a `without where` clause references the event in its `without` clause, but this is not a requirement.

Optionally, after each `without` clause, you can specify a `between` clause, which lists two or more coassigned events or `wait` operators. For an event to cause a match, the type specified in the `without` clause cannot be added to the window between the points specified in the `between` clause. See [“Between clause” on page 638](#).

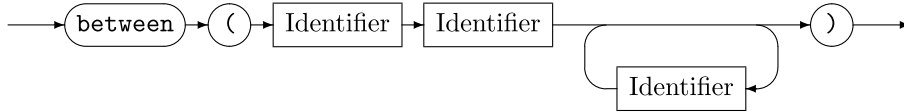
Any `without` clauses must be after any `find where` clauses and `within` clauses. If you specify both optional clauses, the `without where` clause must be before the `between` clause.

Example:

```
find OuterDoorOpened as od -> InnerDoorOpened as id
  where od.user = id.user
  without SecurityCodeEntered as sce where od.user = sce.user {
    emit Alert("Intruder "+id.user);
  }
```

Between clause

In a `within` clause and in a `without` clause, an optional `between` clause restricts which part of the pattern the `within` or `without` clause applies to.



Specify the `between` keyword followed by two or more identifiers that are specified in the event pattern. Enclose the identifiers in parentheses.

The identifiers set a period of time that starts when one of the specified events is received and ends when one of the other specified events is received. The range is exclusive. That is, the range applies only after the first event is received and before the last event is received.

A `between` clause is the only place in which you can specify a coassignment identifier that was assigned in a `wait` clause. You cannot specify identifiers used in a `without` clause. Also, the same event cannot match both the coassignment identifier in the `without` clause and an identifier in a `between` clause.

The condition that the `between` clause is part of must occur in the range of identifiers specified in the `between` clause.

It is illegal to have two `within` clauses with identical `between` ranges. This would be redundant, as only the shortest `within` duration would have any effect. It is, however, legal to have more than one `without` clause with the same `between` range. Typically, these would refer to different event types or where conditions.

Example:

```
find A as a -> B as b -> (C as c and D as d)
  within 10.0 between (a b)
  within 10.0 between (c d)
```

See [“Query condition ranges” on page 118](#) for an explanation of this example.

Select clause

A `find` statement that specifies the `every` keyword can specify a `select` clause to calculate an aggregate value in order to find data based on many sets of events.

Specify the `select` keyword followed by a projection expression coassigned to an identifier. The projection expression contains aggregate function(s) that operate on one or more input events. See [“Built-in aggregate functions” on page 642](#) as well as [“Custom aggregates” on page 642](#).

The projection expression can use coassignments from the pattern if the coassignments are within a single aggregate function call. For example, the following pattern computes the average value of the `x` member of event type `A` in the query's input and coassigns that average value to `aax`.

```
find every A as a select avg(a.x) as aax
```

A `select` clause can use parameter and key values.

In an aggregating `find` statement, only the projection expression can use the coassignments from the pattern. The procedural block of code can use projection coassignments and any parameters or key values, but it cannot use coassignments from the pattern.

In `find` statements without the `every` modifier, only the most recent set of events that match the pattern are used to invoke the procedural code block. With the `every` modifier, every set of events that matches the pattern is available for use by the aggregate function, provided that the latest event is present in one of the sets of events. Any events or combinations of events that do not match the pattern or do not match the `where` clause, or are invalidated due to a `within` or `without` clause, are ignored; their values are not used in the aggregate calculation.

Examples:

```
find every ATMWithdrawal as w
  select last(w.transactionId) as tid
  having last(w.amount) > THRESHOLD * avg(w.amount) {
    route SuspiciousTransaction(tid);
  }
```

```
find every A as a -> B as b
  where b.x >= 2
  select avg(a.x + b.x) as aabx {
    print aabx.toString();
  }
```

See [“Aggregating event field values” on page 122](#) for explanations of these examples, as well as additional examples.

Having clause

A `find` statement that specifies the `every` keyword can specify a `having` clause to restrict when procedural code is invoked.

Specify the `having` keyword followed by a Boolean projection expression. The Boolean projection expression refers to an aggregate calculation. Procedural code is executed only when the Boolean projection expression evaluates to `true`.

You can specify zero, one, or more `having` clauses. When you specify more than one `having` clause, it is equivalent to specifying the `and` operator. That is, each Boolean projection expression must evaluate to `true` for the procedural code to be executed.

A `having` clause can refer to an aggregate value by using the `select` coassignment name.

When you want to test for an aggregate condition but you do not want to use the aggregate value, you can specify a `having` clause without specifying a `select` clause.

Examples:

```
find every A as a
  select avg(a.x) as aax
  having aax > 10.0 {
```

```
    print aax.toString();  
}
```

```
find every A as a  
  having avg(a.x) > 10.0 {  
    print "Average value is greater than ten!";  
}
```


37 Aggregate Functions

■ Built-in aggregate functions	642
■ Custom aggregates	642

In Apama queries and in EPL stream queries, you can specify aggregate functions in the `select` clause. An aggregate function calculates a single value across all items currently in the window. EPL provides a number of commonly used aggregate functions. If a supplied aggregate function does not meet your needs, you can define a custom aggregate function.

See [“Select clause” on page 638](#) for information about the `select` clause in Apama queries.

See also [“Stream queries” on page 671](#).

Built-in aggregate functions

EPL provides built-in aggregate functions in the `com.apama.aggregates` package. All of these functions are available for either bounded or unbounded use. See the `com.apama.aggregates` package in the *API Reference for EPL (ApamaDoc)* for detailed information on each built-in aggregate function.

To use a built-in aggregate function in a query, you must do one of the following:

- Specify the full name of the aggregate function. For example:

```
select com.apama.aggregates.sum(x)
```

- For each aggregate function you want to use in your code, add a `using` statement. This lets you specify aggregate function names without specifying the package name. For example:

```
using com.apama.aggregates.mean;
using com.apama.aggregates.stddev;
...
...select MeanSD( mean(s), stddev(s) );
```

Insert the `using` statement after the optional package declaration and before any other declarations in the `.mon` file.

See also:

- [“Select clause” on page 638](#) for information about the `select` clause in Apama queries.
- [“Working with Streams and Stream Queries” on page 187](#)
- [“Aggregating items in projections” on page 218](#)

Custom aggregates

In an Apama query and in a stream query, you can specify an aggregate function in the `select` clause. If one of the supplied aggregate functions does not meet your needs, you can define a custom aggregate function for use in a `select` clause.

You define custom aggregate functions in a `.mon` file and outside of an event or a monitor. The aggregate function's scope is the package in which you declare it. To use custom aggregate functions in monitors and in Apama queries in other packages, specify the aggregate function's fully-qualified name, for example:

```
from a in all A() select com.myCorporation.custom.myCustomAggregate(a)
```

Alternatively, you can specify a `using` statement. See [“The using declaration” on page 622](#).

Specify `bounded` when you are defining a custom aggregate function that will work with only a bounded window. That is, a stream query cannot specify `retain all`. Specify `unbounded` when you are defining a custom aggregate function that will work with only an unbounded window. That is, a stream query must specify `retain all`. Do not specify either `bounded` or `unbounded` when you are defining a custom aggregate function that will work with either a bounded or an unbounded window.

A custom aggregate function that you want to use in an Apama query must either be a bounded function or it must support both bounded and unbounded operation.

The name of a custom aggregate function must be unique within a package; you cannot overload it or define an event, monitor, or query with the same name as an aggregate function.

The list of formal parameters consists of zero or more comma-separated type/name pairs. Each pair indicates the type and the name of an argument that you are passing to the aggregate function. For example, `(float price, integer quantity)`.

The data type name must be an EPL type. This is the type of the value that your aggregate function returns.

The body of a custom aggregate function can contain fields that are specific to one instance of the custom aggregate function and actions to operate on the state.

Actions

In a custom aggregate function, the `init()`, `add()`, `remove()` and `value()` actions are special. They define how Apama queries and stream queries interact with custom aggregate functions.

- `init()` — If a custom aggregate function defines an `init()` action, it must take no arguments and must not return a value. The correlator executes the `init()` action once for each new aggregate function instance it creates in a query (stream query or Apama query).
- `add()` — A custom aggregate function must define an `add()` action. The `add()` action must take the same ordered set of arguments that are specified in the custom aggregate function signature. That is, the names, types, and order of the arguments must all be the same. The correlator executes the `add()` action once for each item added to the set of items that the aggregate function is operating on.
- `remove()` — A bounded aggregate function must define a `remove()` action. An unbounded aggregate function must not define a `remove()` action. If you do not specify either `bounded` or `unbounded`, the `remove()` action is optional. The `remove()` action must take the same ordered set of arguments as the `add()` action, followed by an argument of the type returned by `add()`, if any, and must not return a value. The correlator executes the `remove()` action once for each item that leaves the set of items that the aggregate function is operating on. The value that `remove()` is called with is the same value that `add()` was called with.

- `value()` — All custom aggregate functions must define a `value()` action. The `value()` action must take no arguments and its return type must match the return type in the aggregate function signature. The correlator executes the `value()` action as follows:
 - In an Apama query, once for each match set and returns the current aggregate value to the query.
 - In a stream query, once per batch per group and returns the current aggregate value to the query.

Custom aggregate functions can declare other actions, including actions that are executed by the above named actions. A custom aggregate function cannot contain a field whose name is `onBeginRecovery`, `onConcludeRecovery`, `init`, `add`, `value`, or `remove`, even if, for example, the custom aggregate function does not define a `remove()` action.

Fields

In the body of a custom aggregate function, you can define fields that are specific to the custom aggregate instance they are in.

38 Statements

■ Simple statements	646
■ Compound statements	651
■ Transfer of control statements	656

Sequences of EPL statements define the steps that are performed by a program. They are executed in the order they are written: sequentially from top to bottom and left to right within a statement block. (For expressions, the evaluation order is affected by parentheses, associativity, and operator precedence.)

The order in which statements are executed is called the flow of control or the control path. Some statements can contain other statements enclosed within their structure and can be used to execute statements conditionally, thus altering the normal control path. You can use the `break`, `continue`, and `return` statements to change the normal control path.

A block is zero or more statements enclosed in curly braces. A block can be used wherever a single statement can be used. Variables declared in a block are able to be referenced only in the block in which they are declared, and only in statements that come after the variable's declaration.

Simple statements

Simple statements are statements that do not enclose other statements or statement blocks and that do not cause a transfer of control. They are executed in the order they are written.

Assignments

An assignment binds a value to a variable, member, sequence element or dictionary value. The value is determined by evaluating the expression on the right side of the assignment operator `:=`. The result type of the expression must match the type of the variable.

Example:

```
integer x := 100;
sequence<float> seq := [0.0, 200.0];
seq[0] := x.toFloat();
dictionary<string, string> dict := {};
dict["key"] := "value";
```

For variables of the reference types, the same value can be bound to more than one variable. See the *API Reference for EPL (ApamaDoc)* for detailed descriptions of all types.

The emit statement

The `emit` statement publishes an event to a named channel of the correlator's output queue. If a channel name is not specified, then the event goes to the default channel whose name is the empty string (`""`). External receivers get events on the default channel only if they are subscribed to all channels.

Note:

The `emit` statement will be deprecated in a future release. Use the `send` statement instead. See [“The send . . . to statement” on page 649](#).

The first expression is an expression whose result type is either an event type or string. If the type is string, then the value of the string is assumed to be in the same format as that produced by the event's `toString()` method.

The expression following the keyword `to` must be of type `string` and is the name of the channel to which the event will be sent.

The `emit` method dispatches events to external registered event receivers. That is, the `emit` statement causes events to go out of the correlator. Active event listeners will not receive events that are emitted.

Events are emitted onto named channels. For an application to receive events from the correlator it must register itself as an event receiver and subscribe to one or more channels. Then if events are emitted to those channels they will be forwarded to it.

Channels effectively allow both *point-to-point* message delivery as well as through *publish-subscribe*. Channels can be set up to represent topics. External applications can then subscribe to event messages of the relevant topics. Otherwise a channel can be set up purely to indicate a destination and have only one application connected to it.

You cannot emit an event whose type is defined inside a monitor.

The `emit` statement can operate on any values as well as events, provided that the any value is of a routable event type.

You cannot emit an event that has a field of type `action`, `chunk`, `listener`, or `stream`. There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type `action`, `chunk`, `listener`, or `stream`.

When you emit an event type that has a dictionary field, the items in the dictionary are sorted in ascending order of their key values.

The `enqueue . . . to` statement

The `enqueue . . . to` statement sends an event to a context you identify.

Note:

The `enqueue . . . to` statement is superseded by the `send . . . to` statement. The `enqueue . . . to` statement will be deprecated in a future release. Use the `send . . . to` statement instead. See [“The `send . . . to` statement” on page 649](#).

You must enqueue an expression of type `event`, and the destination must be one of the following:

- `context` — The `enqueue . . . to` statement sends an event to the back of the input queue of the specified context. The expression is evaluated and the resulting event is sent to the input queue of only the specified context.
- `sequence<context>` — The `enqueue . . . to` statement sends a copy of the event to the back of the input queue of each context in the specified sequence. The expression is evaluated and the resulting event is sent to the input queue of all the contexts in the sequence.

For example:

```
sequence <context> ctxs := [ c1, c2, c3 ];
Ping ping = Ping();
enqueue ping to ctxs;
```

You cannot enqueue an event to a `com.apama.Channel` object that contains a context. You cannot enqueue an event to a dictionary of contexts. However, it is a common pattern to enqueue to a sequence generated by `dictionary.values()`. For example:

```
enqueue x to d.values;
```

If the target context's input queue is full the sending context blocks and waits for space on the queue unless doing so would cause a deadlock. See [“Deadlock avoidance when parallel processing” on page 305](#).

Note that enqueued events are processed in the order they are enqueued. Enqueued events are put on the back of the input queue, behind any externally sourced events already queued.

You must create the context before you enqueue an event to the context. You cannot enqueue an event to a context that you have declared but not created. For example, the following code causes the correlator to terminate the monitor instance:

```
monitor m {  
    context c;  
    action onload()  
    {  
        enqueue A() to c;  
    }  
}
```

If you enqueue an event to a sequence of contexts and one of the contexts has not been created first then the correlator terminates the monitor instance. For details, see [“Sending an event to a particular context” on page 294](#).

Enqueueing an event to a sequence of contexts is non-deterministic. For details, see [“Sending an event to a sequence of contexts” on page 295](#).

The `enqueue...to` statement can operate on any values as well as events, provided that the any value is of a routable event type.

You cannot enqueue an event that has a field of type `action`, `chunk`, `listener`, or `stream`. There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type `action`, `chunk`, `listener`, or `stream`.

The log statement

The `log` statement writes messages and accompanying date and time information to the correlator's log file, if one was specified when the correlator was started.

If there is no log file, then the message is written to the correlator's standard output stream `stdout`.

The expression that you log must be of type `string`. The value is written only if the current logging level in effect is a priority equal to or higher than the log level specified in the `log` statement, with the exception of `OFF`. If you do not specify a level, `INFO` is used. At a log level equal to `OFF`, only logs explicitly set to this level are written. For details, see [“Logging and printing” on page 279](#).

For example:

```
log "Your message here" at INFO;
```


This EPL statement produces a log message that looks like this:

```
2020-01-11 09:08:49.200 INFO [3716] - MyMonitor[1] Your message here
```

The print statement

The `print` statement writes textual messages followed by a newline to the correlator's standard output stream — `stdout`. The expression you `print` must be of type `string`.

For example:

```
print "Your message here.";
```

This EPL statement produces output that looks like this:

```
Your message here.
```

The `print` statement is less useful for reporting diagnostic information than the `log` statement, as it does not contain any information about the time or origin of the message, and cannot be turned off by changing the log level.

For more detailed information, see [“Logging and printing” on page 279](#).

The route statement

The `route` statement evaluates the expression and then sends the resulting event to the front of the current context's input queue.

The expression you `route` must be an event. The event is processed only within the same context that executes the `route` statement.

Routed events are put on the input queue, ahead of any externally sourced events, and ahead of any previously routed events that have not yet been processed. For more details, see [“Event processing order for monitors” on page 611](#).

The `isExternal()` property on events is not changed by routing an event.

The `route` statement can operate on any values as well as events, provided that the any value is of a routable event type.

You cannot route an event that has a field of type `action`, `chunk`, `listener`, or `stream`. There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type `action`, `chunk`, `listener`, or `stream`.

The send . . . to statement

The `send . . . to` statement sends an event to the channel, context, sequence of contexts, or `com.apama.Channel` object that you specify.

You must send an expression of type `event`, and the destination must be one of the following:

- `string` — The `send...to` statement sends the event to the specified channel. All contexts and external receivers subscribed to that channel receive the event. If there are no subscribers to the specified channel or if no receivers are listening on the specified channel then the event is discarded.
- `context` — The `send...to` statement sends the event to the back of the input queue of the specified context. The event expression is evaluated and the resulting event is sent to the input queue of only the specified context.
- `sequence<context>` — The `send...to` statement sends a copy of the event to the back of the input queue of each context in the specified sequence. The event expression is evaluated and the resulting event is sent to the input queue of each context in the sequence.

For example:

```
sequence <context> ctxs := [ c1, c2, c3 ];
Ping ping = Ping();
send ping to ctxs;
```

- `com.apama.Channel` — The `send...to` statement sends the event to the specified `Channel` object. If the `Channel` object contains a string, the event is sent to the channel with that name. If the `Channel` object contains a context, the event is sent to that context. You cannot send an event to an empty context object.

You cannot send an event to a dictionary of contexts. However, it is a common pattern to send to a sequence generated by `dictionary.values()`. For example:

```
send x to d.values;
```

If the target context's input queue is full the sending context blocks and waits for space on the queue unless doing so would cause a deadlock. See [“Deadlock avoidance when parallel processing” on page 305](#).

Sent events are processed in the order they are sent. Sent events are put on the back of the input queue, behind any events already queued.

You must create the context before you send an event to the context. You cannot send an event to a context that you have declared but not created. For example, the following code causes the correlator to terminate the monitor instance:

```
monitor m {
    context c;
    action onload()
    {
        send A() to c;
    }
}
```

If you send an event to a sequence of contexts and one of the contexts has not been created first then the correlator terminates the monitor instance. For details, see [“Sending an event to a particular context” on page 294](#).

Sending an event to a sequence of contexts is non-deterministic. For details, see [“Sending an event to a sequence of contexts” on page 295](#).

The `send...to` statement can operate on any values as well as events, provided that the any value is of a routable event type.

You cannot send an event that has a field of type `action`, `chunk`, `listener`, or `stream`. There is a runtime check if the event (or one of its members) can contain an any field; an exception is thrown if the any field contains an object of type `action`, `chunk`, `listener`, or `stream`.

The spawn statement

The `spawn` statement creates a copy of the currently executing monitor instance in the current context.

See also [“Spawning monitor instances” on page 39](#).

The spawn *action to context* statement

The `spawn action() to context` statement creates a copy of the currently executing monitor instance in the specified context. A monitor instance must have a reference for the specified context in order to spawn to that context.

The expression that you spawn must be of type `context`. The `spawn action()to context` statement spawns a new monitor instance in the specified context.

For more detailed information, see [“Spawning to contexts” on page 291](#).

The throw statement

The `throw` statement causes an exception to be thrown. If it is not caught by a `try ... catch` in that action or any calling actions, then the monitor instance is terminated along with any listeners it has. The syntax is:

```
throw expression;
```

where *expression* must be of the `Exception` type.

Example:

```
action getAverageReading() returns float {
    if readingsCount <= 0 {
        throw Exception("No readings", "IllegalArgumentException");
    }
    return readingsSum / readingsCount.toFloat();
}
```

See also [“Exception handling” on page 276](#).

Compound statements

Compound statements enclose other statements or blocks and affect how the enclosed statements are executed.

The for statement

The `for` statement is used to iterate over the members of a sequence and execute the enclosing statement or block once for each member.

The iteration variable is assigned a value successively obtained from each element of the sequence, starting with the first, and if the last sequence entry has not been reached, the statement that forms the loop body is executed.

The iteration variable's type must match the type of the sequence elements.

The loop body is either a single statement or a block.

Within the loop body, the `break` statement can be used to cause early termination of the loop by transferring control to the next statement after the loop body. The `continue` statement can be used to transfer control to the end of the body, after which the sequence size is tested to determine if the last entry has been reached. If it has not, then the loop body is executed. The `return` statement can be used to terminate both the loop and the action that contains it.

For more information, see [“Defining loops” on page 275](#).

The from statement

The `from` statement is used to create a stream listener. A stream listener watches for items from a stream and passes output items to procedural code.

A `from` statement is similar to an `on` statement, which listens for events processed by the correlator and then executes an event listener action for each matching event or pattern. See [“The on statement” on page 654](#).

You can assign the result of a `from` statement to a `listener` variable. This lets you call `quit()` on the stream listener.

A stream listener passes output items from a stream to procedural code. The stream, specified in the expression, can be a reference to an existing stream or a stream source template. Alternatively, it can be the stream created by an in-line stream query.

A colon and an identifier follow the expression or in-line stream query. This signifies a coassignment: when new items are available from the stream, the stream listener coassigns each output item to the specified variable.

The statement following the identifier can be a single EPL statement or a block of EPL statements. The `from` statement passes the output item to this statement or block and executes the statement or block once for each output item. If the output of the query is a lot that contains more than one item, and you want to execute the statement or block just once for the lot, coassign the output to a sequence. See [“Working with Streams and Stream Queries” on page 187](#), and [“Working with lots that contain multiple items” on page 225](#).

The if statement

The `if` statement is used to conditionally execute a block of code. It checks whether a condition is true or false, that is, the result type must be boolean. The conditional statement may optionally be followed by the keyword `then` followed by a block of code.

If the condition result is `true`, the block is executed. After the body of the `if` has been executed, control is transferred to the next statement following the `if` statement.

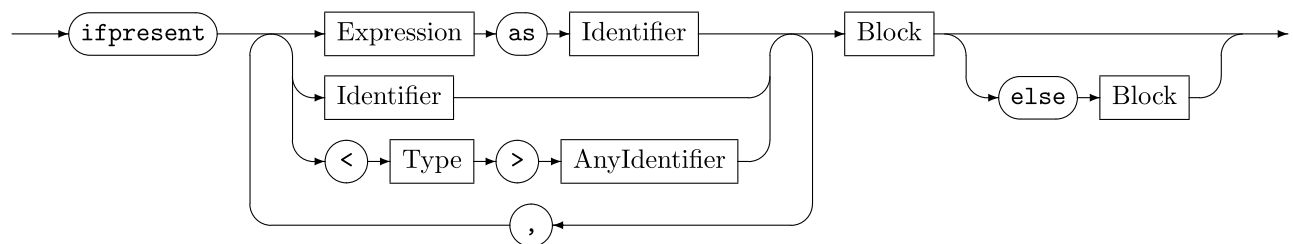
If the condition result is `false` and an `else` clause is present, the statement or block following the `else` is executed. After the body of the `else` clause has been executed, control is transferred to the next statement following the `if` statement.

If the condition result is `false` and the `else` clause is not present, control is transferred to the next statement following the `if` statement.

For more information, see [“Defining conditional logic with the if statement” on page 273](#).

The ifpresent statement

The `ifpresent` statement is used to check if one or more values are empty. It unpacks the values into new local variables and conditionally executes a block of code.



If all of the expressions are non-empty, then their values are unpacked into new local variables (in the scope of the first block of the `ifpresent` statement) and the first block of code is executed. If the expression is a simple identifier (that is, it is referring to a variable or parameter) or casting an any type, then the `as identifier` part can be omitted; the new local retains the same name. An optional `else` block is executed if any of the expressions have an empty value.

`ifpresent` operates on expressions of the following types:

- optional (in which case the new local variable is of the unpacked type)
- chunk (`ifpresent` treats chunk as empty only if its value is the default value)
- stream (`ifpresent` treats stream as empty if its value is the default value or if it has been quit)
- listener (`ifpresent` treats listener as empty if its value is the default value or if it has been quit)
- context (`ifpresent` treats context as empty only if its value is the default value)
- action (`ifpresent` treats action as empty only if its value is the default value)

- any (ifpresent treats any as empty only if it has an empty value)

See the *API Reference for EPL (ApamaDoc)* for more information on these types.

For more information on the default values, see [“Default values for types” on page 586](#).

For more information on the ifpresent statement, see [“Defining conditional logic with the ifpresent statement” on page 274](#).

The on statement

The on statement is used to create an event listener that looks for input events that match the pattern specified by an event condition. When a matching event is detected, the event listener fires (also referred to as triggers) and the specified event listener action is executed.

A listener assignment clause is used to obtain a reference to the event listener that is created by the on statement. One can either define a new variable of type listener or specify a reference to an existing listener variable.

An Apama query cannot specify an on statement.

Example:

```
listener l := on ...  
sequence <listener> aSequence;  
aSequence[0] := on ...
```

The event condition specifies what events are of interest. See [“Event expressions” on page 613](#).

A listener action defines the processing that will be performed when a matching event is detected and the event listener fires. The listener action can be one of the following:

- A statement
- A block

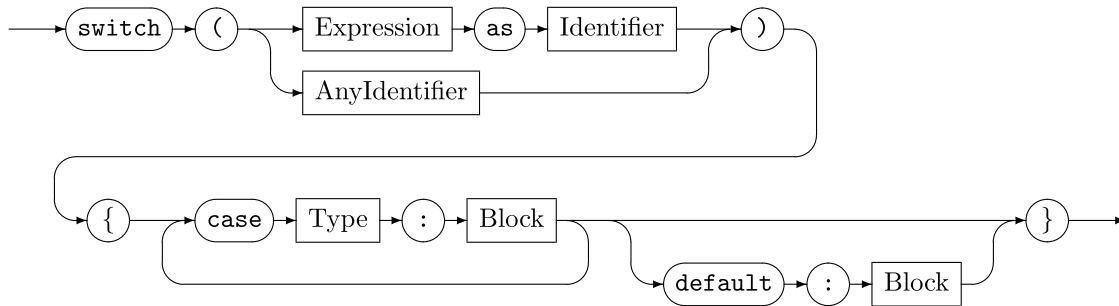
The listener action is invoked automatically by the correlator when the event condition is satisfied. This may be:

- When a matching event is detected.
- If unmatched is specified in the condition, the event matches the condition, and there are no matching event listeners that do not specify the unmatched keyword.
- If completed is specified in the condition, and any matching events have been completely processed by other event listeners.

For more information, see [“Specifying the on statement” on page 147](#).

The switch statement

The `switch` statement is used to conditionally execute a block of code based on the type of an any expression. Unlike the `if` and `if ... else` statements, the `switch` statement can have a number of possible execution paths.



The `switch` statement names the expression as an identifier with the `as` keyword followed by an identifier to name the value. In each case clause block, the identifier has the same type as the case clause.

If the expression is a simple identifier (that is, it is referring to a variable or parameter), then the `as Identifier` part can be omitted. The new local retains the same name.

For more details, see [“Handling any values of different types with the switch statement” on page 272](#).

The try ... catch statement

The `try ... catch` statement is used to handle runtime exceptions.

The catch clause must specify a variable whose type is `com.apama.exceptions.Exception`.

You can nest `try ... catch` statements in an action, and you can specify multiple actions in a `try` block and specify a `try ... catch` statement in any number of actions.

See also [“Exception handling” on page 276](#).

Example:

```

using com.apama.exceptions.Exception;
...
action getExchangeRate(
    dictionary<string, string> prices, string fxPair) returns float {
    try {
        return float.parse(prices[fxPair]);
    } catch(Exception e) {
        return 1.0;
    }
}

```

The while statement

The `while` statement is used to repeatedly evaluate a boolean condition and execute a block as many times as the condition result is found to be true.

The condition, whose result type must be boolean, is evaluated and if the result is `true`, the block is executed. Control then transfers to the top of the loop and the condition is evaluated again. When the condition result is `false`, control is transferred to the next statement following the `while` statement.

The body of the loop must be a block; it must be inside curly braces.

Within the loop body, the `break` statement can be used to cause early termination of the loop by transferring control to the next statement after the loop body. The `continue` statement can be used to transfer control to the end of the body, after which the condition will be evaluated again and the loop body executed if the condition result is `true`. The `return` statement can be used to terminate both the loop and the action that contains it.

For more information, see [“Defining loops” on page 275](#).

Transfer of control statements

Transfer of control statements alter the normal control path by stopping the sequential execution of statements within a block. All of them end execution of the block that contains them. After a `continue` statement is executed, the containing block might be executed again in a new loop iteration. The `die` and `return` statements also end the action in which they are executed.

The break statement

The `break` statement transfers control to the next statement following the loop (`for` or `while` statement) that encloses the `break` statement. A `break` statement can only be used within a `for` or `while` statement. Any statements between the `break` statement and the end of the block are not executed. For more information, see [“Defining loops” on page 275](#).

The continue statement

The `continue` statement can be used in a block enclosed by a `for` or `while` statement to end execution of the current iteration and transfer control to the beginning of the loop. When a `continue` statement is executed, control is immediately transferred to the beginning of the inner most enclosing `for` or `while` statement. Any statements between the `continue` statement and the end of the block are not executed. For more information, see [“Defining loops” on page 275](#).

The die statement

The `die` statement terminates the execution of a monitor. When the correlator executes a `die` statement, it terminates only the monitor instance that contains the `die` statement being executed. If the monitor instance that spawned the monitor instance being terminated is still active, that

monitor instance is not affected. If that original monitor instance spawned any other monitor instances, those monitor instances are not affected. If the monitor instance being terminated defines an `ondie()` action, the correlator executes the `ondie()` action for just the monitor instance being terminated, and then terminates the monitor instance.

An Apama query cannot specify a `die` statement.

For more information, see [“Terminating monitor instances” on page 43](#).

The return statement

The `return` statement ends the execution of an action and control is transferred to the action's caller, at the point following the action call (which might be in the middle of an expression). Any statements between the `return` statement and the end of action are not executed.

If the action does not have a `returns` clause, then an expression is not permitted in the `return` statement.

If the action has a `returns` clause, then an expression whose value is the action's return value is required in the `return` statement. The expression type must match the type specified in the `returns` clause.

For more information, see [“Format for defining actions” on page 251](#).

39 Expressions

■ Introduction to expressions	660
■ Using an expression as a statement	660
■ Primary expressions	661
■ Bitwise logical operators	661
■ Logical operators	663
■ Shift operators	664
■ Comparison operators	665
■ Additive operators	666
■ Multiplicative operators	667
■ Unary additive operators	668
■ Expression operators	668
■ Expression operator precedence	669
■ Postfix expressions	670
■ Stream queries	671
■ Stream source templates	675

In many programs, much work is performed by evaluating expressions, which are combinations of operators, operands, and punctuation. They are used to detect events of interest to the program, perform calculations, comparisons, invoke actions, invoke inbuilt methods, compute parameter values passed to action and method calls, and so on.

Introduction to expressions

EPL has several kinds of expressions:

- Primary expressions, bitwise expressions, logical expressions etc. are used for computations.
- In a monitor, a stream query definition creates a derived stream from an existing stream.
- In a monitor, a stream source template creates a new stream from an event template.

Event expressions are used in on statements for event pattern matching and sequence detection. Event expressions are not ordinary EPL expressions. See [“Event expressions” on page 613](#).

When an expression is evaluated (that is, it is executed), it will produce a result value if the expression is a variable, a literal, or a combination of values and operators. If the expression is an action or inbuilt method call, then evaluating the expression produces a result value when the action or inbuilt method returns a value, but if the action or inbuilt method does not return a value, then the expression does not produce a result. Note that when an expression includes action or method calls, then evaluating the expression might produce side effects. A side effect is a change in the state of the execution environment. For example, a called action might change the value of a global variable or generate a derived event. If evaluating an expression produces a result, then in addition to a value, the expression result has a type. This is the expression type. An expression's type is always known at compile time.

The elements of an expression are evaluated roughly from left to right, taking into account parentheses and operator precedence. Binary operators have a left operand and a right operand. If an operator is left-associative, its left operand is evaluated first, followed by the right, and then the operation is performed. If an operator is right-associative, its right operand is evaluated first, followed by the left, then the operation is performed. In action calls, the actual parameter list expressions are evaluated from left to right. Many of the operators used in expressions are polymorphic and can operate on operands of several types. For example, the addition operator performs floating point addition when its operands are of type `decimal` or `float` and performs integer addition when its operands are of type `integer`. Here are some examples of expressions:

```
i := (a.size() + b[3]) / (n - 1);  
i := "foo" + s + " " + b.toString() + f.formatFixed(8);
```

Using an expression as a statement

An expression that does not return a value can be used as a statement. For example, an action or method call can be used in this way.

Example:

```
action doSomething() { // Action has a side-effect but returns nothing  
    ...  
}
```

The primary expression is the simplest form of expression. It can take the following forms:

- The bitwise logical operators examine one bit at a time in their operands and compute the corresponding bit value in the result.

The bitwise operators `and`, `or`, and `xor` are binary operators that have a left and right operand. The bitwise operator `not` is a unary operator that has only a right operand.

The result type of all four bitwise operators is `integer`. Note that EPL integers are 64 bits wide.

The bitwise intersection operator `&` produces a result by comparing all 64 bits of its left and right operands, which must be expressions of type `integer`, one bit at a time. For each bit in the two operands, the corresponding bit in the result value is set to 1 if both operand bit values are 1 and set to 0 if either operand bit value is 0.

The following illustrates this using 64-bit binary values.

- ## Developing Apama Applications 10.7.1

Bitwise union (or)

The bitwise union or produces a result by comparing all 64 bits of its left and right operands, which must be expressions of type integer, one bit at a time. For each bit in the two operands, the corresponding bit in the result value is set to 1 if either or both operands bit values is 1 and set to 0 if both operand bit values are 0.

Example

The following illustrates this using 64-bit binary values.

[illegible]

Bitwise exclusive (xor)

The bitwise exclusive or operator `xor` produces a result by comparing all 64 bits of its left and right operands, which must be expressions of type `integer`, one bit at a time. For each bit in the two operands, the corresponding bit in the result value is set to 1 if either operand's bit value is 1 and the other is 0 and set to 0 if both operand bit values are 0 or both are 1. In other words, the result bit is 1 if both bit values are different and 0 if they are the same.

Example

The following illustrates this using 64-bit binary values.

[illegible]

Note that the expression `a xor b` yields the same result as `not (a and b)`.

Unary bitwise inverse

The unary bitwise not operator produces a result by computing the bitwise complement or inverse of its right operand, which must be an expression of type `integer`. For each bit in the operand's value, the corresponding bit in the result value is set to 1 if the operand's bit value is 0 and 0 if the operand's bit value is 1.

Example

The following illustrates this using 64-bit binary values.

■ `b := 42;`

[illegible]

■ not b

[illegible]

Logical operators

The logical operators `and`, `or`, `xor` and `not` perform Boolean arithmetic on their operands.

The logical operators' left and right operands are expressions whose result type must be `boolean`. The result type of all four operators is `boolean`.

Logical intersection (and)

The `and` operator produces a result of `true` if both of its operand values are `true` and `false` otherwise.

When the correlator evaluates a logical and expression, it evaluates the left operand first. If the left operand evaluates to `false`, then the correlator does not evaluate the right operand since the expression cannot be `true`. For example:

a and b

If `a` is false, then whether or not `b` is true, the expression will be false so the correlator does not evaluate `b`. This lets you write code such as the following:

```
if (dict.containsKey(k) and dict[k] = "someValue")
```

If `k` is not in the dictionary then the left operand evaluates to `false` and so the entire logical expression is `false`. The correlator never evaluates `dict[k] = "someValue"`, which would cause an error if `k` is not in the dictionary.

Logical union (or)

The or operator produces a result of true if either of its operand values is true and false otherwise.

When the correlator evaluates a logical or expression, it evaluates the left operand first. If the left operand evaluates to true, then the correlator does not evaluate the right operand since the expression will always be true. For example:

a or b

If `a` is `true`, then regardless of what `b` evaluates to, the expression will be `true` so the correlator does not evaluate `b`.

Logical exclusive or (xor)

The xor operator produces a result of true if either of its operand values is true and the other is false and false if both are true or both are false.

Unary logical inverse (not)

The unary not operator produces the result `true` if its right operand value is `false`, and `false` if the operand value is `true`.

Shift operators

The shift operators << and >> perform a shift of an integral value, moving bits in the result a specified number of positions to the right or left. The result type of both shift operators is integer.

The left operand is an expression of type `integer` whose value is to be shifted. The right operand is the shift count, an expression of type `integer` whose value is the number of bits the left operand value is to be shifted.

The shift count must be a nonnegative value less than 64. If the shift value is zero, then the result value is equal to the left operand value. Values less than zero or greater than 63 will produce unpredictable results and should not be used.

Left shift operator

The left shift operator << produces a result by moving the left operand value's bits to the left and filling the vacated bits on the right with 0 bits. Bits that are moved beyond the leftmost bit (the sign bit) position are discarded.

Example

The following illustrates this using 64-bit binary values.

■ `i := 42;`

[illegible]

- $i \ll 24$

[illegible]

Operator	Operation	Description
=	Equality	Produces the result <code>true</code> if the left operand's value is equal to the right operand's value and <code>false</code> if they are not equal.
!=	Inequality	Produces the result <code>true</code> if the left operand's value is not equal to the right operand's value and <code>false</code> if they are equal.
=>	Greater than or equal to	Produces the result <code>true</code> if the left operand's value is larger than or equal to the right operand's value and <code>false</code> otherwise.
>	Greater than	Produces the result <code>true</code> if the left operand's value is larger than the right operand's value and <code>false</code> otherwise.

Additive operators

The additive operators are used to perform arithmetic on two operands of matching type: both of type `decimal`, both of type `integer`, or both of type `float`. The result type of the additive operators is the same as the type of the operands.

The additive operators are:

Operator	Operation	Description
+	Addition	Produces a result by computing the numeric sum of its left and right operands. If the two operands are both expressions of type <code>integer</code> , then integral addition is performed and the result is of type <code>integer</code> . If the two operands are both of type <code>decimal</code> or both of type <code>float</code> , then floating-point addition is performed and the result type is the same as the operand type.
-	Subtraction	Produces a result by computing the numeric difference between the left and right operands by subtracting the value of the right operand from the left. If the two operands are both expressions of type <code>integer</code> , then integral subtraction is performed and the result is of type <code>integer</code> . If the two operands are both of type <code>decimal</code> or both of type <code>float</code> , then floating-point subtraction is performed and the result type is the same as the operand type.
+	String concatenation	Produces a result by “adding” two strings together. The result is a new string whose value is the value of the right operand, an expression of type <code>string</code> , appended to the value of the left operand, an expression of type <code>string</code> . The result type of the string concatenation operator is <code>string</code> .

Multiplicative operators

The multiplicative operators are used to perform arithmetic on two operands of matching type: both `decimal`, or both `float`, or both `integer`.

The left and right operands must both be expressions of type `decimal`, or both be of type `float`, or both be of type `integer`.

The result type of the multiplicative operators is the same as the type of the operands.

The multiplicative operators are:

Operator	Operation	Description
<code>*</code>	Multiplication	<p>Produces a result by computing the numeric product of its two operands. If the two operands are both expressions of type <code>integer</code>, then integral multiplication is performed and the result is of type <code>integer</code>. If the two operands are both of type <code>decimal</code> or both of type <code>float</code>, then floating-point multiplication is performed and the result type is the same as the operand type.</p>
<code>/</code>	Division	<p>Produces a result by computing the numeric quotient of its two operands. The left operand value, the dividend, is divided by the right operand value, the divisor. If both operands are of type <code>integer</code>, any fractional part of the result value is discarded. In other words, the result is truncated toward zero. For example, the expression <code>13/5</code> yields a result of 2. If both operands are of type <code>integer</code>, then integral division is performed and the result is of type <code>integer</code>. If both operands are of type <code>decimal</code> or both are of type <code>float</code>, then floating-point division is performed and the result type is the same as the operand type.</p> <p>If the right operand's value is zero, a runtime error is raised.</p>
<code>%</code>	Remainder	<p>Produces a result by computing the numeric remainder from dividing the left operand value by the right operand value. For example, the expression <code>13%5</code> yields a result of 3. If both operands are of type <code>integer</code>, then the integral remainder is computed and the result is of type <code>integer</code>. If both operands are of type <code>decimal</code> or both of type <code>float</code>, then the floating-point remainder is computed and the result type is the same as the operand type.</p> <p>If the right operand's value is zero, a runtime error is raised.</p>

Unary additive operators

The unary additive operators are used to perform arithmetic on one right operand of type `decimal`, `float` or `integer`. The result type of the unary arithmetic operators is the same as the type of the operand.

Both of the unary arithmetic operators have one operand, which must be an expression of type `decimal`, `float` or `integer`. The result type is the same as the type of the operand.

Unary inverse

The unary additive inverse operator produces a result that is its right operand value with the sign reversed. If the operand value is negative, the result value is positive. If the operand value is positive, the result value is negative. If the operand value is zero, the result value is zero.

Unary identity

The unary additive identity operator `+` produces a result that is its right operand value.

Expression operators

You can use the following operators wherever you can specify an expression. Note that they are all binary operators.

Operator	Operation	Description
<code>+</code>	Addition	Returns a <code>decimal</code> , <code>float</code> or an <code>integer</code> according to the operands, or concatenation in the case of string operands
<code>-</code>	Subtraction	Returns a <code>decimal</code> , <code>float</code> or an <code>integer</code> according to the operands
<code>%</code>	Modulus	Returns an <code>integer</code> and is a valid operator only for integers
<code>/</code>	Division	Returns a <code>decimal</code> , <code>float</code> or an <code>integer</code> according to the operands
<code>*</code>	Multiplication	Returns a <code>decimal</code> , <code>float</code> or an <code>integer</code> according to the operands
<code>></code>	Greater than	Returns a <code>boolean</code> value indicating whether the condition expressed is true or false
<code><</code>	Less than	Returns a <code>boolean</code> value indicating whether the condition expressed is true or false
<code>>=</code>	Greater than or equal to	Returns a <code>boolean</code> value indicating whether the condition expressed is true or false

Operator	Operation	Description
<=	Less than or equal to	Returns a boolean value indicating whether the condition expressed is true or false
=	Equivalence	Returns a boolean value indicating whether the condition expressed is true or false
!=	Not equals	Returns a boolean value indicating whether the condition expressed is true or false
or	Logical or, bitwise or	On boolean types, on integers
and	Logical and, bitwise and	On boolean types, on integers
xor	Logical xor, bitwise xor	On boolean types, on integers
not	Logical not	On boolean types

Expression operator precedence

The following table lists the primary and bitwise expression operators in order by their precedence, from lowest to highest. See also [“Event expression operator precedence” on page 617](#).

Operation	Operator	Precedence
Logical or bitwise union	or	1
Logical or bitwise exclusive or	xor	2
Logical or bitwise intersection	and	3
Unary logical or bitwise inverse	not	4
Relational	<, <=, >, >=, !=, =	5
Additive	+, -	6
String concatenation	+	6
Multiplicative	*, /, %	7
Unary additive	+, -	8
Cast	<type>	9
Name qualifier (dot)	.	10
Object constructor	new	10
Subscript	[]	10
Action call	actionName()	11

Operation	Operator	Precedence
Parenthesized expression	()	11
Stream query	from	11
Stream source template	all	11

For clarity, it is strongly recommended to use brackets in expressions. This makes it very easy to understand what an expression means. For example:

```
(10 * 10) > (9 * 9)
```

Do not use un-bracketed expressions, except where trivial. The following is equivalent to the above expression and is just about acceptable:

```
10 * 10 > 9 * 9
```

It is bad practice, however, to use an expression such as the following as it relies on intimate knowledge of the precedence:

```
1 + 2 - 3 * 4 / 5 < -1 or 5
```

Postfix expressions

A primary followed by a "." symbol, and an identifier must represent a variable reference, an action call, or a method call. Action and method calls are described in [“Action and method calls” on page 670](#).

An expression enclosed by the [and] symbols denotes a subscript operation for a sequence or dictionary. This can be used on the right or left side of an assignment statement.

The new operator is used to create an instance of a reference type or event type.

Action and method calls

An action call within an expression transfers control to the statements within the action body during expression evaluation and temporarily suspends the expression evaluation. If the action has parameters, then their values are copied to the action's formal parameter variables. When the control flow reaches the action's end or the action executes a return statement, control is transferred back to the expression and evaluation continues.

The actual parameters are a comma-separated list of expressions. The entire list is enclosed in parentheses. It forms the set of parameter values that are passed when the action is called. Each expression value is copied to the corresponding parameter variable specified in the action definition's formal parameters, and the expression result type must match the parameter variable's type. The number and order of actual parameters passed by a caller must also match those listed in the action definition's formal parameters.

The action or method being invoked in the expression must return a value. The action's return type becomes the expression result type.

The subscript operator []

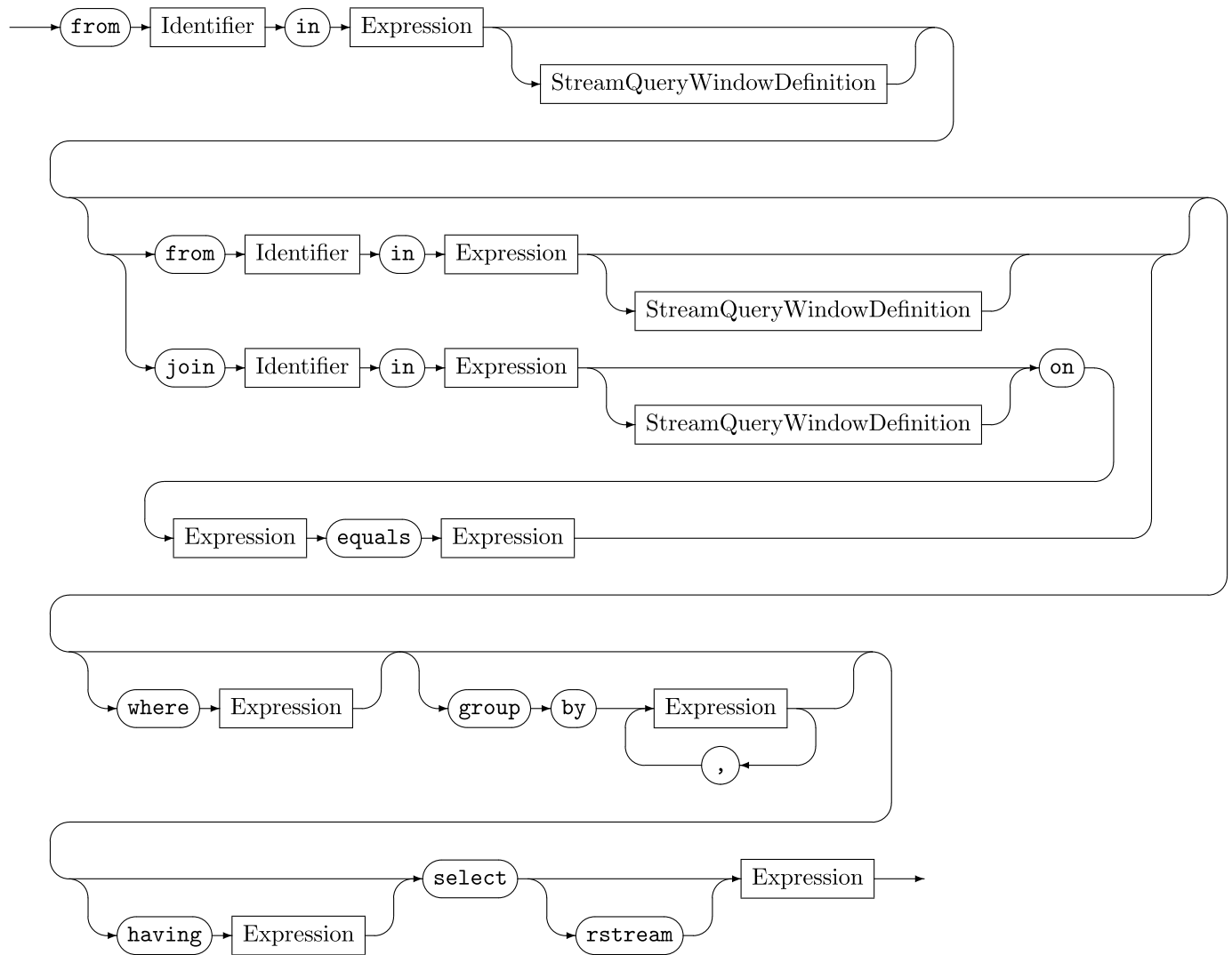
The subscript operator takes one operand. The operand can be an integer index into a sequence or a key type index of a dictionary. The subscript operator produces a result of the same type as the sequence's entry type or dictionary's item type.

The new object creation operator

The operator `new` produces a result whose type is the type of the object parameter. It has one operand, the name of the type of object to be created.

Stream queries

A stream query defines an operation that the correlator applies continuously to one or two streams of items. The output of a stream query is a continuous stream of derived items, `stream<X>`, where `X` is the type returned by the expression in the `select` clause. See also [“Defining stream queries” on page 193](#).



A `from` clause specifies a stream that the query is operating on.

An item in a stream can be an event, a simple type (boolean, decimal, float, integer or string) or a location type. The first *Identifier* is the identifier that represents the current item in the stream you are querying. You use this identifier in subsequent clauses in the stream query.

The first *Expression* identifies the stream that you want to query.

A stream query window definition is optional. If you do not specify any window then the stream query operates on only the items that arrive on the stream for a given activation of that query. See [“Stream query window definitions” on page 673](#).

A subsequent `from` clause indicates a cross-join operation.

Alternatively, a subsequent `join` clause indicates an equi-join operation. An equi-join has a key expression for each of the two streams that are being joined. Two items are joined into an output item only if the values of their key expressions are equal.

A `where` clause qualifies the items produced from a window or a join operation.

A `group by` clause organizes the qualified items, or the items produced from a window or join operation.

A `having` clause filters the output items produced from the projection.

The required `select` clause specifies how to generate the output items.

Semantic constraints

from Identifier in Expression join Identifier in Expression

The identifier can be any legal identifier and, within the stream query's scope, is associated with items from the source stream and therefore has their type. In a joined stream query, the two identifiers must be distinct.

The expression's result must be a value of some stream type. The correlator evaluates the expression outside the stream query's scope. For example:

```
stream<A> a := all A();
from a in a ...
```

This is legal, because the identifier `a` is not in scope for evaluation of the expression `a`.

on Expression1 equals Expression2

The correlator evaluates both expressions within the stream query's scope.

Expression1 must contain the first item identifier and cannot contain the second. *Expression2* must contain the second item identifier and cannot contain the first.

The two expressions must return the same type, and that type must be a comparable type.

where Expression group by Expression, Expression, ...

The item identifier or identifiers are in scope and should be used in these expressions. The `where` expression must return a `boolean` value. The `group by` expressions can return any comparable types.

having Expression

The item identifier or identifiers are in scope and can be used in this expression. The presence of this clause implies that the projection must be an aggregate projection. The expression must return a `boolean` value.

You can use one or more aggregate functions in the `having` expression. In fact, you can use aggregate functions only in `having` expressions and `select` expressions.

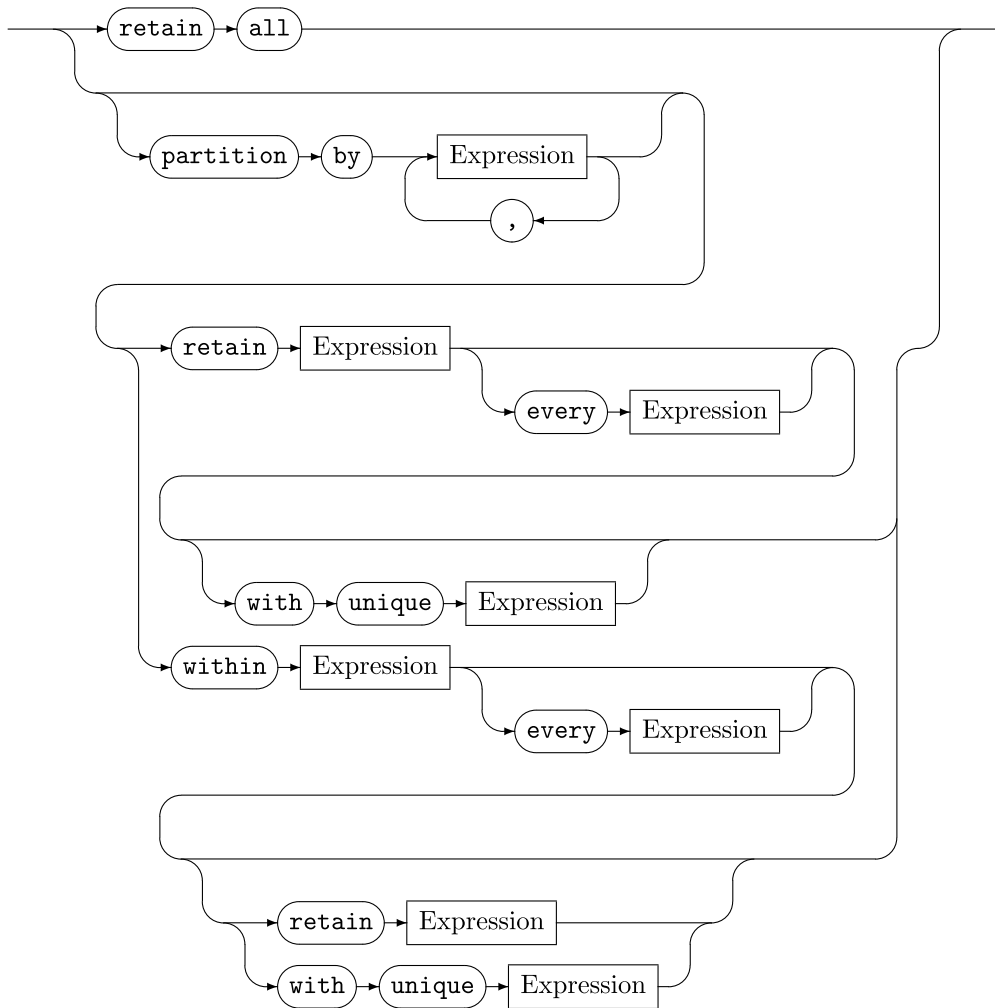
select [rstream] Expression

The item identifier or identifiers are in scope and can be used in this expression. The expression must return a value.

You can use one or more aggregate functions in a `select` expression. In fact, you can use aggregate functions only in `having` expressions and `select` expressions. If you specify an aggregate function you cannot specify the `rstream` keyword.

Stream query window definitions

In a stream query, the optional window definition specifies which items in a stream to operate on. See also [“Adding window definitions to from and join clauses” on page 199](#).



Typically, stream queries process a window over a stream. A stream is an ordered sequence of items over time. A window specifies which items to operate on. Windows can contain a portion of the stream based on number of items, time of item arrival, content of item, or other criteria.

When the stream query window definition is `retain all`, the window contains all items that have ever been in the stream. Conceptually, once an item enters a `retain all` window, it remains in the window indefinitely, or until the stream query is terminated. The `retain all` clause specifies an unbounded window. Unbounded windows have restrictions on their use:

- You cannot have a partitioned or batched unbounded window.
- You cannot perform a join operation on an unbounded window.
- You cannot specify an unbounded window when you use `rstream` in the `select` clause of a stream query.

When you use a custom aggregate function in a stream query that contains an unbounded window, you cannot use a bounded aggregate function. You should also be aware that, if you use a badly implemented custom aggregate function in a stream query that contains an unbounded window, then this can result in uncontrolled memory usage.

A `partition by` clause divides the input data into several partitions and then applies the stream query window definition separately to each partition. The `partition by` expressions must be comparable types.

The `retain` clause specifies the maximum number of items to be retained by the window. The `retain` expression must be an integer expression. In a size-based window, as each new item arrives in the stream, it is added to the window. After the number of items in the window reaches the window size limit specified in the `retain` clause, the arrival of a new item causes removal of the oldest item from the window.

The `within` clause specifies the number of seconds to keep each new item in the window. The `within` expression must be a float expression. In a time-based window, as each new item arrives in the stream, it is added to the window. As soon as an item has been in the window for the number of seconds specified by the `within` expression, the correlator removes the item from the window.

By default, the contents of a window change upon the arrival of each item. The `every` keyword can be used to control when the contents of the window change, which causes the items to be added to the window in batches of several items at once. Time-based windows can be controlled to update only every p seconds and size-based windows can be controlled to update only every m events.

The contents of the window can also depend on the content of individual items in the stream. Specify `with unique Expression` to limit the window to containing only the most recent item for each key value identified by the expression.

Semantic constraints

In a stream query window definition for one of a joined stream query's input streams, it is always an error to refer to the other input stream's item identifier.

`partition by Expression, Expression, ...`

You should use the item identifier in each expression. Expressions can return any comparable types.

`retain Expression [every Expression]`

You cannot use the item identifier in these expressions. These expressions must return integer values.

`within Expression [every Expression]`

You cannot use the item identifier in these expressions. These expressions must return float values.

`with unique Expression`

You should use the item identifier in this expression. The expression can return any comparable type.

Stream source templates

A stream can be created from an event template using the `all` keyword. This is referred to as a stream source template.

A stream source template is the `all` keyword followed by a single event template. The output of a stream source template is a continuous stream of items, `stream<X>`, where *X* is the type specified by the event template.

See also [“Creating streams from event templates” on page 189](#).

40 Variables

■ Variable declarations	678
■ Variable scope	678
■ Provided variables	679
■ Specifying named constant values	681

Variables are names that are bound to data values (in the case of primitive types) or the location of data values (in the case of reference types). Variables are declared by specifying a type, a name, and optionally, an initial value. With the exception of the `string` type, once declared, new values can be computed and assigned to variables as needed. Strings are immutable and variable assignment causes a new string value to be created and bound to the string variable.

Variable declarations

Before a variable can be referenced in a program, it must be declared. The declaration gives the variable a unique name, a type and, optionally, an initial value. See also [“Assignments” on page 646](#).

A variable declaration statement can appear anywhere in a block. Variables declared in a block are in scope in that block and can be used in statements that follow the declaration.

Example:

```
location rect := location(1.0, 1.0, 5.0, 5.0);
integer i;
boolean c := true, d := false;
sequence <integer> s := [1, 3, 5, 7, 11, 13, 17];
string s1 := "abcdefghijklmnopqrstuvwxy";
```

Variable scope

The parts of a program in which a particular variable can be referenced (that is, its value used or a new value assigned) is called the scope of the variable. In EPL, variables can have scopes that include:

- All monitors. These are global variables that are part of EPL, also called predefined variables.
- The monitor within which they are declared.
- The action within which they are declared.
- The block within which they are declared.
- The event within which they are declared.
- The custom aggregate function in which they are declared.
- The stream query within which they are identified.

Regardless of the scope of a variable, it cannot be referenced in statements or expressions until after it has been declared or specified as an item identifier in a stream query. Further, variables scoped to actions or blocks cannot be referenced until a value has been assigned.

Within a scope at a particular level, variables declared at that level must have unique names. They can, however, have names that are the same as variables defined at an outer scope and in that case the variables declared at the inner level hide or mask the ones defined at the outer level(s) until the end of their scope.

Predefined variable scope

Predefined variables are defined by the correlator and are accessible in all monitors. See [“Provided variables” on page 679](#).

Monitor scope

A variable that is defined in a monitor is visible and can be referenced in all parts of the monitor. Such variables are also called global variables.

Action scope

A variable that is declared in an action (also called a local variable) can only be referenced within the action. A variable that is a formal parameter of an action can only be referenced within the action. If a local variable declared in an action has the same name as a global variable declared at the monitor level, the local variable hides the global variable until the end of the action.

Block scope

A variable that is declared within a block can only be referenced within the block. A block is one or more statements enclosed within curly braces (the characters { and }). If a local variable declared in a block has the same name as a global variable declared at the monitor level, or a local variable declared at the action level, the block's local variable hides the global variable or the action's variable, or both if all three have the same name, until the end of the block (the closing }).

Event action scope

The fields of an event are part of the event declaration. An event field's scope depends on where it is declared. When an event also includes action definitions, the statements in the action can reference the event's fields as simple identifiers. From the point of view of an event's action, the fields can be said to be scoped to the event.

Custom aggregate function scope

A variable that is declared in a custom aggregate function (also called a local variable) can only be referenced within the custom aggregate function. If a local variable declared in a custom aggregate function has the same name as a global variable declared at the monitor level, the local variable hides the global variable until the end of the custom aggregate function.

Provided variables

The EPL execution environment provides several variables. You can use these variables in the same way as variables you declare yourself, except that you cannot assign values to them. Instead, the correlator automatically assigns values to these variables.

currentTime

The `currentTime` variable is a read-only `float` global variable that contains a timestamp value with the current time and date as read from the correlator's clock. Timestamps are encoded as the number of seconds and fractional seconds elapsed since midnight, January 1, 1970 UTC and do not have a time zone associated with them.

The current time is the time indicated by the most recent clock tick. Use the `currentTime` variable to obtain the current time. The value of the `currentTime` variable is always changing to reflect the correlator's current time.

If you have multiple contexts, it is possible for the current time to be different in different contexts. A particular context might be doing so much processing that it cannot keep up with the time ticks on its queue. In other words, if contexts are mostly idle, then they would all have the same current time.

In a context, the current time is never the same as the current system time. In most circumstances it is a few milliseconds behind the system time. This difference increases when the context's input queue grows.

When a listener executes an action, it executes the entire action before the correlator starts to process another event. Consequently, while the listener is executing an action, time and the value of the `currentTime` variable do not change. Consider the following code snippet,

```
float a;
action checkTime() {
    a := currentTime;
}
// ... Lots of additional code
// A listener calls the following action some time later
action logTime() {
    log a.toString(); // The time when checkTime was called
    log currentTime.toString(); // The time now
}
```

In this code, an event listener sets `float` variable `a` to the value of `currentTime`, which is the time indicated by the most recent clock tick. Some time later, a different event listener logs the value of `a` and the value of `currentTime`. The values logged might not be the same. This is because the first use of `currentTime` might return a value that is different from the second use of `currentTime`. If the two event listeners have processed the same event, the logged values are the same. If the two event listeners have processed different events, the logged values are different.

The correlator maintains a clock that advances at a fixed interval (default) of 0.1 seconds. The clock does not advance while an event is being processed.

Event timestamps

The correlator defines an arrival timestamp for every event it receives. The arrival time value is set from the main context's clock when an event is received by the correlator, just before it is placed on the input queue of each public context.

You can access the arrival timestamp by calling the event's inbuilt `getTime()` method (see the description of `event` in the *API Reference for EPL (ApamaDoc)*). After the correlator creates an event or after you coassign an event, the `getTime()` method returns the time in the context when the event was created or coassigned. An event's arrival timestamp has the same scope as the event itself.

self

The predefined variable `self` is an event reference that can be used to refer to an event instance within the event's definition.

Within an event action body, you can use the `self` variable to refer an event instance of that event type. In other words, the scope of `self` is each action body in the event definition. For example:

```
event Circle
{
    float radius;
    action circumference() returns float
    {
        return (2.0 * float.PI * self.radius);
    }
    action area() returns float
    {
        // Note the use of "radius" is equivalent to
        // "self.radius" in the statement below.
        return (float.PI * radius * self.radius);
    }
}
```

Note:

You cannot use the `self` variable in an Apama query or in a static action.

Specifying named constant values

A constant is a named literal and its value cannot be changed during runtime. It resembles a variable declaration with `constant` before it.

You can declare an identifier for a constant value in an event type definition or in a monitor. A constant appears in memory once. Spawning a monitor that contains a constant does not make copies of the constant.

The type of a constant must be `boolean`, `decimal`, `float`, `integer`, or `string`.

The name you assign to a constant must be unique within the event type or monitor that contains the constant definition.

The literal that you assign to the constant must be the specified type.

When you define a constant event field, you can refer to that constant from outside the event. Qualify the name of the constant with the event name, for example, `MyEvent.myConstant`.

You cannot declare a constant in an action, directly in a package, or in a custom aggregate function.

See also [“Specifying named constant values”](#) on page 249.

41 Lexical Elements

■ Program text	684
■ Comments	684
■ White space	685
■ Line terminators	686
■ Symbols	687
■ Identifiers	687
■ Keywords	687
■ Operators	688
■ Separators	689
■ Literals	689
■ Names	693
■ Annotations	694

The lexical rules of the EPL grammar describe how sequences of characters are used to form the basic elements of the language, that is, identifiers, constants (string, numeric, and so on), operators, separators, white space, comments, and language keywords. These elements, after discarding any white space and comments, form the symbols used in the syntactical grammar of the language.

Program text

A program's source text is composed of an optional UTF-8 byte-order marker followed by characters that form a sequence of symbols, white space, comments, and line terminators, up to the end of file (denoted by the EOF symbol).

The UTF-8 byte order marker is a sequence of three consecutive bytes with the values 0xEF, 0xBB, and 0xBF respectively, appearing at the beginning of a file containing EPL source text. The UTF-8 character encoding format does not need a byte-order marker to indicate the byte order because UTF-8 is by definition a bitwise encoding. A UTF-8 byte-order marker at the start of a file just indicates that the program text is encoded in the UTF-8 format. It is inserted automatically by some text editors, such as Notepad on Windows systems.

A program's source text can be encoded as Unicode UTF-8, as 7-bit ASCII (which is a proper subset of UTF-8), or various other encodings. The compiler will convert the source text from the locale's encoding to UTF-8 if necessary. In practice, this really only affects comments, white space, and string literals because all other EPL constructs are limited to the ASCII subset. [“Identifiers” on page 687](#), for example, are limited to only a few of the many possible Unicode characters.

Comments

Comments are explanatory notes or text intended for human readers to help them understand what a program or section of a program does.

There are several kinds of comments:

- **Block comments**

Block comments begin with the character sequence slash-asterisk `/*`, which is followed by any number of other characters and line breaks, followed by a closing asterisk-slash `*/` sequence. The entire contents of all block comments are ignored.

- **End-of-line comments**

End-of-line comments begin with two consecutive slash characters `//` followed by any number of characters up to and including the end of the current line. The entire contents of all end-of-line comments are ignored.

- **ApamaDoc comments**

ApamaDoc comments are a special kind of block comment which begin with the character sequence slash-asterisk-asterisk `/**`. Their content is used when generating documentation for your EPL code. See also [“Generating Documentation for Your EPL Code” on page 429](#).

White space

White space characters are characters such as spaces and tabs that are used between symbols to separate them. White space characters are sometimes required between symbols when they would otherwise be misinterpreted or unrecognizable. For example, the symbol `/` is used as the division operator and the symbol `*` is used as the multiplication operator, but the character pair `/*` with no white space between them marks the beginning of a block comment.

Though they act as separators between symbols, white space characters are otherwise ignored and discarded during program compilation.

Judicious use of white space improves a program's readability.

The ASCII white space characters and their encodings are listed below:

Code Point	UTF-8 Encoding	ASCII Encoding	Name
0x0020	0x20	0x20	Space
0x0009	0x09	0x09	Horizontal Tab
0x000c	0x0c	0x0c	Form Feed
0x001c	0x1c	0x1c	File Separator
0x001d	0x1d	0x1d	Group Separator
0x001e	0x1e	0x1e	Record Separator
0x001f	0x1f	0x1f	Unit Separator

The Unicode white space characters, as defined by the Unicode character dictionary, and their encodings are listed below:

Code Point	UTF-8 Encoding	Name
0x0085	0xc2 0x85	unnamed control character
0x00a0	0xc2 0xa0	NO-BREAK SPACE
0x1680	0xe1 0x9a 0x80	OGHAM SPACE MARK
0x180e	0xe1 0xa0 0x8e	MONGOLIAN VOWEL SEPARATOR
0x2000	0xe2 0x80 0x80	EN QUAD
0x2001	0xe2 0x80 0x81	EM QUAD
0x2002	0xe2 0x80 0x82	EN SPACE
0x2003	0xe2 0x80 0x83	EM SPACE
0x2004	0xe2 0x80 0x84	THREE-PER-EM SPACE

Code Point	UTF-8 Encoding	Name
0x2005	0xe2 0x80 0x85	FOUR-PER-EM SPACE
0x2006	0xe2 0x80 0x86	SIX-PER-EM SPACE
0x2007	0xe2 0x80 0x87	FIGURE SPACE
0x2008	0xe2 0x80 0x88	PUNCTUATION SPACE
0x2009	0xe2 0x80 0x89	THIN SPACE
0x200a	0xe2 0x80 0x8a	HAIR SPACE
0x2028	0xe2 0x80 0xa8	LINE SEPARATOR
0x2029	0xe2 0x80 0xa9	PARAGRAPH SEPARATOR
0x202f	0xe2 0x80 0xaf	NARROW NO-BREAK SPACE
0x205f	0xe2 0x81 0x9f	MEDIUM MATHEMATICAL SPACE
0x3000	0xe3 0x80 0x80	IDEOGRAPHIC SPACE

All white space characters appearing between two symbols are ignored. However, note that white space appearing within string literals is not ignored. See [“Literals” on page 689](#).

Line terminators

Line terminators are used to mark the end of a line of source text. Different operating systems use different characters or character sequences to mark the end of a line.

The following terminators are used on various operating systems:

Operating System	Line Terminator
Mac OS X	ASCII Carriage Return (0x0D)
UNIX	ASCII Newline (0x0A)
Linux	ASCII Newline (0x0A)
Windows	ASCII Carriage Return (0x0D) followed by ASCII Newline (0x0A)

In general, any number of line terminators can be used between any two symbols in a program and they are treated the same as other white space. A line terminator appearing at the end of an end-of-line comment terminates the comment.

Symbols

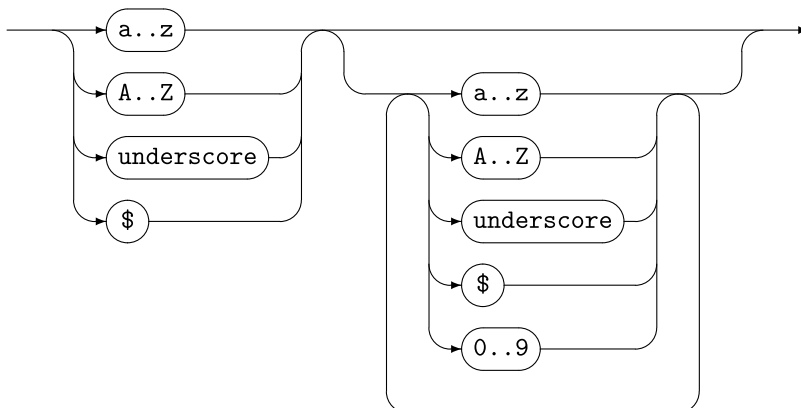
Symbols (also called tokens, atoms, or lexemes) are the elements and words of the language, consisting of identifiers, keywords, operators, separators, and literals. Symbols are composed of one or more characters, excluding white space, comments, and line terminators.

Identifiers

An identifier is a character sequence composed of a combination of the following characters:

- The 26 letters of the Roman alphabet in upper and lower case
- Digits 0 through 9
- Underscore character (_)
- Dollar sign (\$)

The first character must not be a digit. Identifiers are case sensitive. An identifier must not have the same spelling as a keyword. For example, the word `action` is a keyword and cannot be used as an identifier. An identifier can also contain a hash symbol (#) as the first character. This is helpful if you want to use a keyword as an identifier. See also [“Keywords” on page 687](#).



Keywords

EPL keywords are case sensitive. They are reserved words that are an intrinsic part of the language, and must not be used as identifiers (for example, `monitor`, `print` and `event`). See also [“Identifiers” on page 687](#).

There are also some words that EPL may use as keywords in a future release, and you should avoid using them as identifiers (for example, `public`, `class` and `byte`).

Software AG Designer and the correlator will warn you if you attempt to use a keyword or a word reserved for future use.

If you absolutely have to use a keyword or a word reserved for future use as an identifier, then you have to prefix this word using the hash symbol (#). This may be the case if an external system mandates a particular field name in an event type. For example, if an externally created digital event type has a field name that is an EPL keyword, such as `action`, you would have to specify `#action` as the field name. Adding the hash symbol (#) makes sure that EPL accepts a keyword as an identifier. Internally, however, Apama treats `#action` as `action`.

If using JMon, a Java class that contains an EPL keyword may need to map to an equivalent EPL event definition using the hash symbol (#) in EPL.

Operators

Operators are symbols used in expressions and statements to perform a computation on or test a relation between data values or, in event expressions, to detect sequences and patterns of events. As you will see, the same symbol is sometimes used for different operations, depending on the context in which the operator is used. For example, the `and` operator is used both in logical expressions, and event sequencing and the `*` operator is used both for integer and floating point multiplication and to match any value in event templates.

Ordinary operators

The ordinary operators are used in primary and bitwise expressions. See [“Expressions” on page 659](#) to perform calculations and comparisons on variables, data values, and other constructs. The descriptions of the built-in types in the *API Reference for EPL (ApamaDoc)* provide information about the operators that you can use with values of each type.

The ordinary operators are grouped into the following subcategories:

- **Arithmetic operators.** See the corresponding topics in [“Expressions” on page 659](#).
- **Comparison operators.** See [“Comparison operators” on page 665](#).
- **Logical operators.** See the corresponding topics in [“Expressions” on page 659](#).

Event operators

Event operators are special operators that are used in the `on` statement's event expression. An `on` statement defines an event listener. See [“Event expressions” on page 613](#) and [“Event expression operator precedence” on page 617](#).

An `on` statement is not allowed in an Apama query.

Field operators

Field operators are used within event expressions to define conditions on individual fields in an event template. See [“Field operators” on page 607](#).

Separators

Separators are symbols that are used in certain statements and expressions. These are:

```
{  
}  
[  
]  
(  
)  
.  
;  
,  
:  
  
white space
```

Separators are used to:

- Keep the various parts from bumping into each other, for example commas between parameter values in an action call.
- Group related elements together, for example the left and right braces at the beginning and end of a block of statements.

Literals

A literal is a source text representation of a constant value of a primitive type, or a location, dictionary, or sequence type.

You might want to declare a constant for a frequently used literal so that you can refer to it by name. See [“Specifying named constant values” on page 681](#).

Boolean literals

There are two Boolean literal values: `true` and `false`.

Example:

```
a := true;  
b := false;
```

Integer literals

Integer literal values can be written either base 10 (decimal) or base 16 (hexadecimal).

Base 10 literals

Base 10 integral literal values are a sequence of one or more of the digits 0 through 9.

Examples:

```
i := 0;
i := 11;
i := 1023;
i := 9223372036854775807;
```

The value can optionally be preceded by a sign. If the sign is omitted, + is assumed.

The number 9223372036854775807 or $(2^{63} - 1)$ is the largest base 10 integer literal value that can be represented.

Base 16 literals

Base 16 integral literal values begin with the characters 0x, and consist of a combination of the decimal digits 0 through 9 and the hexadecimal digits a through f and A through F.

Examples:

```
j := 0x0;
j := 0xd;
j := 0xAFF;
j := 0xffffffffffffffff;
```

The number 0xffffffffffffffff or $(2^{63} - 1)$ is the largest base 16 integer literal value that can be represented.

Floating point and decimal literals

Floating-point literal values can take one of the following forms:

- Optional sign, integer digits followed by an exponent.
- Optional sign, integer digits, a decimal point, and an optional exponent,
- Optional sign, integer digits, a decimal point, fraction digits, and an optional exponent.
- Optional sign, a decimal point, fraction digits, and an optional exponent.

If the sign is omitted, + is assumed. If the exponent is omitted, e0 is assumed.

The exponent is the letter “e” followed by an optional sign, and one or more decimal digits.

Examples:

```
f := 0.0;
f := 1.;
f := 200128.00005
f := 3.14159265358979;
f := 1e4;
f := 1e-4;
f := 10000e0;
f := .1234;
f := .1234e4;
f := 1.E-32;
f := 1.E-032;
f := 6.0221415E23;
f := 1.7976931348623157e308;
```

The largest positive floating point literal value that can be represented in EPL is $1.7976931348623157 \times 10^{308}$. The smallest positive nonzero value that can be represented is $2.2250738585072014 \times 10^{-308}$. If you write a floating-point literal whose value would be outside the range of values that can be represented, the compiler raises an error.

String literals

A string literal is a sequence of characters enclosed in double quotes.

The backslash character is used as an escape character to allow inclusion of special characters such as newlines and horizontal tabs.

To include a double quote in a string literal, precede it with a `\` character which serves as an escape character, which means "do not treat this quote as the end of the string literal".

To include a newline, use `\n`.

To include a tab character, use `\t`.

To include a single `\` character, use two: `\\`. The compiler will remove the extra backslashes.

Examples:

```
s := "Hello, World!";
s := "\ta\tstring\twith\ttabs\tbetween\twords";
s := "a string on\n two lines";
s := "a string with \\ a backslash and a \" quote";
```

The length of a string literal is limited only by available memory at compile time and runtime. In practice, this means you can make them as long as you need.

Location literals

The four float literals form the location's corner point coordinates, `x1`, `y1` and `x2`, `y2`.

Example:

```
location(0.0, 0.0, 10.0, 10.0)
```

Dictionary literals

A dictionary literal can contain one or more pairs of key/item values.

The first expression in a dictionary literal entry is the key value and the second expression is the item value. In a dictionary literal, all key values must be the same type and all item values must be the same type. Both must be of a type that matches the types specified in the dictionary variable's definition.

A dictionary literal must contain at least one key/item pair except when the dictionary literal is in an initializer. For example, the following statement is valid:

```
myDictionary := {};
```

The following statement is not valid:

```
takesADictionaryArgument({});
```

Example:

```
{1:"One", 2:"Two", 3:"Three"}
```

Sequence literals

A sequence literal can contain one or more sequence item values.

Each expression in the comma separated list is one entry in the sequence literal. The types must all be the same and must match the sequence type.

A sequence literal must contain at least one item except when the sequence literal is in an initializer. For example, the following statement is valid:

```
mySequence := [];
```

The following statement is not valid:

```
takesASequenceArgument([]);
```

Example:

```
[1,2,3,4]
```

Time literals

In Apama query definitions, time literals can be in `within` clauses. They are either `float` or `integer` literals followed by a unit. Not all units are required, but they have to be in order.

You can specify the following time literals, in the following order:

- `day/days`
- `hour/hours`

- `min/minute/minutes`
- `sec/second/seconds`
- `msec/millisecond/milliseconds`

For example:

- `10 hours`
- `1.5 days`
- `1 day 2.5 hours 10 min 4 sec`
- `2 day 3.5 minutes`

A space is required between a `float` or `integer` literal and its associated time unit. A space is required between a time unit and a `float` or `integer` literal that follows it. Additional whitespace is also allowed.

You cannot specify a negative number.

Outside a query, you can use these keywords as identifiers. Inside a query, you cannot use these keywords as identifiers unless you prefix them with a hash symbol (`#`). See also [“Keywords” on page 687](#).

Names

Names are used in EPL programs to refer to the various different kinds of entities in the program. Actions, variables and reference variable members, parameters, monitors, queries, methods, aggregate functions, events, packages, and plug-ins all have names.

Description

Names are either simple or qualified. Simple names consist of a single identifier. Qualified names consist of a sequence of identifiers separated by `.` symbols, with an optional `.` prefix.

Every name has a scope, which is the part of a program's text where the name can be used as a simple identifier. The scope is determined by where in the program the name is declared. See [“Variable scope” on page 678](#).

Do not create EPL structures in the `com.apama` namespace. This namespace is reserved for future Apama features. If you inadvertently create an EPL structure in the `com.apama` namespace, the correlator might not flag it as an error in this release, but it might flag it as an error in a future release.

Name Precedence

When there are duplicate unqualified names for types, the correlator searches for the associated definition in the following order, and uses the first one it finds:

1. The monitor-internal type definitions, for example, event type definitions and custom aggregate function definitions.
2. Definitions that have been brought in with a `using` declaration in the current file.
3. Definitions in the current package (this could be the root namespace if a package was omitted).
4. The root namespace.

The fully qualified name of a type can always be named by using a dot (.) followed by the fully qualified name. For example, `select .com.apama.aggregates.avg(x)` uses the built-in `avg` type, even if `com` is a name in the current package.

If you try to create a package-level type that has the same name as a definition brought in with a `using` declaration, it causes a compiler error and the code does not inject. For example:

```
package foo;
using bar.Bar;
event Bar { // Causes an error when injecting as Bar has already been
            // defined by a "using" declaration.}
```

You cannot define a type that has the same fully-qualified name as another type.

If two types have the same name but are in different packages, either one can take precedence over the other depending on their ordering in the precedence list. The correlator uses the first match it finds even if that results in an error when a lower-priority match would have worked. For example:

```
X x;
```

This causes an error if, for example, there is an aggregate function called `x` in the current package even if there is an event type called `x` in the root namespace. You can use a `.` prefix on the name to force it to be looked up from the root namespace, in which case the fully qualified name must be used.

Annotations

A program can contain predefined annotations before specific language elements. For detailed information, see [“Adding predefined annotations” on page 52](#).

42 Limits

EPL enforces the limits described in the following table.

EPL Limit	Value
Lowest integer	-2^{63} (-9223372036854775808)
Highest integer	$2^{63} - 1$ (9223372036854775807)
Integer precision	64 bits (about 18 decimal digits)
Maximum integer left shift	63 bits
Maximum integer right shift	63 bits
Lowest negative floating point value	$-1.7976931348623157 \times 10^{308}$
Highest negative nonzero floating point value	$-2.2250738585072014 \times 10^{-308}$
Lowest positive nonzero floating point value	$2.2250738585072014 \times 10^{-308}$
Highest positive floating point value	$1.7976931348623157 \times 10^{308}$
Floating point precision	About 15 decimal digits
Lowest negative decimal floating point value	$-9.999999999999999 \times 10^{384}$
Highest negative nonzero decimal floating point value	-10^{-398}
Lowest positive nonzero decimal floating point value	10^{-398}
Highest positive decimal floating point value	$9.999999999999999 \times 10^{384}$
Decimal precision	Exactly 16 decimal digits
Maximum identifier length	Limited by available memory
Maximum number of entries in a sequence	Limited by available memory

EPL Limit	Value
Maximum number of entries in a dictionary	Limited by available memory
Maximum number of characters in a string	Limited by available memory
Maximum number of active listeners	Limited by available memory, typically many tens of thousands
Maximum number of active monitors	Limited by available memory
Maximum number of fields in an event	2^{16} (65536)
Maximum number of actions in an event	2^{16} (65536)
Maximum indexed fields in an event	32
Memory address space available to EPL runtime	The correlator stops if it runs out of memory
Maximum number of active stream queries	Limited by available memory
Maximum stream window size	Limited by available memory

A EPL Naming Conventions

It is recommended that you use the following naming conventions in EPL. These conventions closely follow Java naming conventions. Using these conventions makes it easier to collaborate and makes it faster for Software AG Global Support personnel to follow your code.

Item	Convention	Notes and Examples
Acronyms	Do not always use all capitals	Names often contain standard abbreviations, such as IAF for Integration Adapter Framework. Names such as <code>iafInterface</code> for an attribute or <code>IafInterface</code> for a monitor are easier to read than <code>iaFInterface</code> and <code>IAFInterface</code> .
Actions	<code>lowerCamelCase</code>	Actions should be verbs, in mixed case with the first letter lowercase, and the first letter of each internal word capitalized. For example: <pre>handleQuery(); startDaemonProcess(); quit();</pre>
Channels	<code>package.UpperCamelCase</code>	Channel names should start with an EPL package name (lowercase), optionally followed by an <code>UpperCamelCase</code> noun. Qualifying channel names with a package is important because channel names form a global namespace that is shared by all applications running in a correlator. For example: <pre>com.mycompany.AllTransactions</pre>
Constants	<code>ALL_CAPITALS</code>	Identifiers for constants should be all uppercase with words separated by underscores. For example: <pre>constant integer MAX_SIZE; constant string DEFAULT_HOST;</pre>
Contexts	<code>UpperCamelCase</code>	Context names should be nouns, initial capital, in mixed case with the first letter of each internal word capitalized. Context names should be simple and should describe the work being done in the context. Use whole words. Avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as URL or IAF. For example:

Item	Convention	Notes and Examples
		<pre>context("Calculation"); context("Inventory", true);</pre>
Custom aggregate functions	lowerCamelCase	<p>Custom aggregate functions should be in mixed case with the first letter lowercase, and the first letter of each internal word capitalized.</p> <pre>aggregate bounded myCustomAggregate() returns integer { aggregateBody }</pre>
Events	UpperCamelCase	<p>Event names should have an initial capital, and mixed case with the first letter of each internal word capitalized. Event names should be simple and descriptive. Use whole words. Avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as URL or IAF. For example:</p> <pre>event Tick event SubscriptionConfiguration event IafEvent</pre>
Monitors	UpperCamelCase	<p>Monitor names should be nouns, initial capital, in mixed case with the first letter of each internal word capitalized. Monitor names should be simple and descriptive. Use whole words. Avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as URL or IAF. For example:</p> <pre>monitor SubscriptionManager monitor IafMonitorService</pre>
Packages	lowercase	<p>The prefix of a unique package name is always written in all-lowercase ASCII letters and should preferably be one of the top-level domain names (com, edu, gov, mil, net, org) or one of the two-letter codes identifying countries as specified in ISO 3166-1 alpha-2.</p> <p>Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names. For example:</p> <pre>com.apamax.accounting</pre>
Queries	UpperCamelCase	<p>Query names should be nouns, initial capital, in mixed case with the first letter of each internal word capitalized. Query names should be descriptive. Use whole words. Avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form, such as URL or IAF. For example:</p>

Item	Convention	Notes and Examples
		<pre>query FaultyWithdrawalLocations query CloseInTimeButDistantTransactions</pre>
Variables	lowerCamelCase	<p>Variables and parameters should have initial lowercase. This is left to your discretion, but lowercase is preferable. Internal words start with capital letters.</p> <p>Variable names should be short yet meaningful. The choice of a variable name should be mnemonic: that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary, throwaway, variables. Common names for temporary variables are <i>i</i>, <i>j</i>, <i>k</i>, <i>m</i>, and <i>n</i> for integers.</p> <pre>integer i; float myPrice; MyEvent myEvent;</pre>

B Testing Apama Applications Using PySys

Developing a comprehensive suite of automated tests is an essential aspect of building reliable applications in any language. So that you can easily create tests for your Apama applications, Apama distributes an open-source automated system-testing framework called PySys.

PySys is an easy-to-use cross-platform framework for writing and orchestrating tests in which a test case consists of a simple Python file that specifies the required logic for starting processes, waiting for the expected events and log messages, and validating the results.

PySys provides a comprehensive library of utility methods to make writing these tests really easy, and has powerful features such as concurrent test-case execution, automatic cleanup of processes and files, collection of code-coverage information, memory monitoring, and reporting of performance statistics. It also provides a pluggable framework for recording test results with built-in support for the widely used Apache Ant JUnit XML output format that is supported by most continuous integration servers.

Apama includes PySys itself, as well as some helper classes for starting Apama processes such as the correlator from PySys. Apama also provides a template that we recommend using to create your project configuration file.

To get started with PySys, proceed as follows:

1. Open the Apama Command Prompt (or on Unix, use `source bin/apama_env`; see also "Setting up the environment using the Apama Command Prompt" in *Deploying and Managing Apama Applications*).
2. Go to the directory containing your application, create a subdirectory for the tests and change to it.
3. Use the following command to create a default `pysysproject.xml` configuration file in this directory, using the Apama template:

```
pysys makeproject --template=apama
```

4. Create your first test:

```
pysys make MyApplication_001
```

5. Run your first test:

```
pysys run
```

A set of sample PySys testcases for Apama can be found in the `samples/pysys` directory of your Apama installation.

Reference documentation for PySys and the Apama helper classes for PySys can be found in the *API Reference for Python*. This is located in the `doc/pydoc` directory of your Apama installation.

See "Using EPL code coverage with PySys tests" in *Deploying and Managing Apama Applications* for information on how to enable EPL code coverage reporting for PySys.