Developing Apama Applications in EPL

5.1.0

November 2013

This document applies to Apama 5.1.0 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 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA, and/or Terracotta Inc., San Francisco, CA, USA, and/or Software AG (Canada) Inc., Cambridge, Ontario, Canada, and/or, Software AG (UK) Ltd., Derby, United Kingdom, and/or Software A.G. (Israel) Ltd., Or-Yehuda, Israel 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 http://documentation.softwareag.com/legal/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are located at http://documentation.softwareag.com/legal/ 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 and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products." This document is located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

Document ID: -Developing_Apama_Applications_In_EPL-5.1-20131107

Table of Contents

Preface....13
About this documentation....14
How this book is organized....14
Documentation roadmap....15
Contacting customer support....16

Getting Started with Apama EPL....19
Introduction to Apama Event Processing Language....20
How EPL applications compare to applications in other languages....21
About dynamic compilation in the correlator....21
About Apama Studio development environment....21
Terminology....22
Defining event types....26
Allowable event field types....27
Format for defining event types....27
Example event type definition....29
Working with events....29
Methods on events....29
Making event type definitions available to monitors....32
Allocating events....33
Event processing order....35

Defining Monitors....37
About monitor contents....38
Loading monitors into the correlator....39
Terminating monitors....40
Unloading monitors from the correlator....40
Example of a simple monitor....40
Spawning monitor instances....42
How spawning works....43
Sample code for spawning....44
Terminating monitor instances....46
About executing ondie() actions....47
Specifying parameters when spawning....47
Communication among monitor instances....48
Organizing behavior into monitors....48
Sending events to other monitors....49
Defining your application's message exchange protocol....50
Using events to control processing....51
About service monitors....51
Adding a bundle to your project....52
Utilities for operating on monitors....53

Defining Event Listeners....55
About event expressions and event templates....56

Specifying the on statement....58
Using a stream source template to find events of interest....59
Defining event expressions with one event template....60
Listening for one event....60
Listening for all events of a particular type....60
Listening for events with particular content....61
Using positional syntax to listen for events with particular content....61
Using name/value syntax to listen for events with particular content....61
Listening for events of different types....62
Terminating and changing event listeners....63
Specifying multiple event listeners....65
Listening for events that do not match....66
Specifying completion event listeners....67
Example using unmatched and completed....68
Improving performance by ignoring some fields in matching events....69
Defining event listeners for patterns of events....69
Specifying and/or/not logic in event listeners....71
Specifying the or operator in event expressions....72
Specifying the and operator in event expressions....72
Example event expressions using and/or logic in event listeners....72
Specifying the not operator in event expressions....73
Specifying and not logic to terminate event listeners....73
Pausing event listeners....74
Choosing which action to execute....75
How the correlator executes event listeners....76
How the correlator evaluates event expressions....76
Avoiding event listeners that trigger upon instantiation....77
When the correlator terminates event listeners....77
How the correlator evaluates event listeners for a series of events....78
Evaluating event listeners for all A-events followed by B-events....78
Evaluating event listeners for an A-event followed by all B-events....80
Evaluating event listeners for all A-events followed by all B-events....81
Defining event listeners with temporal constraints....83
Listening for event patterns within a set time....84
Waiting within an event listener....85
Triggering event listeners at specific times....86
Using variables to specify times....87
About timers and their trigger times....88
Understanding time in the correlator....89
Disabling the correlator's internal clock....90
Generating events that keep time....91
About repeating timers....91
Setting the time in the correlator....92
Out of band connection notifications....92
Out of band notification events....93
Enabling out of band notifications....94

Working with Streams and Stream Queries....97

Introduction to streams and stream networks....98
Defining streams....99
Creating streams from event templates....99
Terminating streams....100
Using output from streams....100
Listener variables and streams....102
Coassigning to sequences in stream listeners....102
Defining stream queries....102
Linking stream queries together....103
Simple example of a stream network....104
Stream query definition syntax....104
Stream query processing flow....106
Specifying input streams in from clauses....106
Adding window definitions to from and join clauses....108
Window definition syntax....108
Defining time-based windows....110
Defining size-based windows....111
Combining time-based and size-based windows....112
Defining batched windows....113
Partitioning streams....114
Partitions and aggregate functions....115
Using multiple partition by expressions....116
Partitioning time-based windows....117
Defining content-dependent windows....117
Joining two streams....118
Defining cross-joins with two from clauses....119
Defining equi-joins with the join clause....120
Filtering items before projection....122
Generating query results....123
Aggregating items in projections....124
Filtering items in projections....129
IEEE special values in stream query expressions....130
Defining custom aggregate functions....131
Example of defining a custom aggregate function....132
Defining actions in custom aggregate functions....132
Overloading in custom aggregate functions....133
Distinguishing duplicate values in custom aggregate functions....133
Working with lots that contain multiple items....134
Stream queries that generate lots....134
Behavior of stream queries with lots....135
Size-based windows and lots....136
Join operations and lots....136
Grouped projections and lots....138
Stream network lifetime....138
Disconnection vs termination....139
Rules for termination of stream networks....140
Using dynamic expressions in stream queries....141
Behavior of static and dynamic expressions in stream queries....141

When to avoid dynamic expressions in stream queries ... 141
Ordering and side effects in stream queries ... 142
Understanding when the correlator evaluates particular expressions ... 143
Using dynamic expressions in windows ... 143
Using dynamic expressions in equi-joins ... 144
Using dynamic expressions in where predicates ... 144
Using dynamic expressions in projections ... 144
Examples of using dynamic expressions in stream queries ... 145
Example of altering query window size or period ... 145
Example of altering a threshold ... 145
Example of looking up values in a dictionary ... 146
Example of actions and methods in dynamic expressions ... 147
Troubleshooting and stream query coding guidelines ... 147
Prefer on statements to from statements ... 148
Know when to spawn and when to partition ... 148
Filter early to minimize resource usage ... 148
Avoid duplication of stream source template expressions ... 149
Avoid using large windows where possible ... 149
In some cases prefer retain all to a timed window ... 150
Prefer equi-joins to cross-joins ... 150
Be aware that time-based windows can empty ... 150
Be aware that fixed-size windows can overflow ... 150
Beware of accidental stream leaks ... 151

Defining What Happens When Matching Events Are Found ... 153
Using variables ... 154
Using global variables ... 154
Using local variables ... 156
Using variables in listener actions ... 157
Specifying named constant values ... 158
Defining actions ... 159
Format for defining actions ... 159
Invoking an action from another action ... 160
Specifying actions in event definitions ... 161
Using action type variables ... 163
Getting the current time ... 169
Generating events ... 170
Generating events with the route command ... 171
Generating events with the enqueue command ... 171
Enqueuing to contexts ... 172
Generating events to emit to outside receivers ... 173
Assigning values ... 174
Defining conditional logic ... 174
Defining loops ... 175
Catching exceptions ... 176
Logging and printing ... 178
Specifying log statements ... 179
Log levels determine results of log statements ... 179

Where do log entries go? 181
Examples of using log statements 181
Strings in print and log statements 182
Sample financial application 182

Implementing Parallel Processing 185
Introduction to contexts 186
What is inside/outside a context? 186
About context properties 187
Context lifecycle 187
Comparison of a correlator and a context 187
Creating contexts 188
Calling context methods 189
How many contexts can you create? 189
Obtaining context references 189
Spawning to contexts 190
Enqueuing an event to a particular context 191
Enqueuing an event to a sequence of contexts 193
Common use cases for contexts 194
Samples for implementing contexts 194
Simple sample implementation of contexts 195
Running samples of common concurrency problems 195
About the samples of concurrency problems 196
About the race sample 197
About the deadlock sample 198
About the compareswap sample 199
Contexts and correlator determinism 201
How contexts affect other parts of your Apama application 201
About replay logs and parallel processing 202
Deadlock avoidance when parallel processing 203
Clock ticks when parallel processing 203
Using correlator plug-ins in parallel processing applications 203

Using Correlator Persistence 205
Description of state that can be persistent 206
When persistence is useful 206
When non-persistent monitors are useful 207
How the correlator persists state 207
Enabling correlator persistence 208
How the correlator recovers state 210
Recovery order 211
Defining recovery actions 212
Simplest recovery use case 213
Designing applications for persistence-enabled correlators 213
Upgrading monitors in a persistence-enabled correlator 213
Sample code for persistence applications 214
Sample code for discarding stale state during recovery 215
Sample code for recovery behavior based on downtime duration 215
Sample code that recovers subscription to non-persistent monitor 216

Requesting snapshots....217
Developing persistence applications....217
Using correlator plug-ins when persistence is enabled....217
Using the MemoryStore when persistence is enabled....218
Comparison of correlator persistence with other persistence mechanisms....219
Restrictions on correlator persistence....220

Common EPL Patterns....221
Contrasting using a dictionary with spawning....222
Translation using a dictionary....222
Translation using spawning....222
Factory pattern....223
Canonical factory pattern....223
Alternate factory pattern....224
Using quit() to terminate event listeners....224
Combining the dictionary and factory patterns....225
Testing uniqueness....225
Reference counting....226
Inline request-response pattern....228
Routing events for request-response behavior....228
Canonical form for synchronous requests....229
Writing echo monitors for debugging....229

Using Correlator Plug-ins in EPL....231
Overhead of using plug-ins....232
When to use plug-ins....232
When not to use plug-ins....232
Using the Payload Extraction plug-in....233
Accessing the payload in the correlator....233
The direct-access functions....234
Parameters for primary direct-access functions....234
Secondary direct-access functions....235
Parameters for secondary direct-access functions....236
The indirect-access functions....236
Secondary indirect-access functions....237
Using the Time Format plug-in....238
Time Format plug-in format functions....239
getTime()....239
Time Format plug-in parse functions....240
parseTimeFromPattern()....241
Time Format plug-in compile pattern functions....242
getMicroTime()....243
Format specification for the Time Format plug-in functions....243
Using the Log File Manager plug-in....248
Plug-in functions....248
Events that can be routed to the LoggingManager monitor....250
Including the Log File Manager in a project....251
Loading the Log File Manager when deploying with the Management & Monitoring console....252
Using the MemoryStore....252

Introduction to using the MemoryStore....253
Overview of MemoryStore events....254
Overview of steps for using the MemoryStore....254
Adding the MemoryStore bundle to your project....255
Preparing and opening stores....255
Description of row structures....257
Preparing and opening tables....259
Using transactions to manipulate rows....260
Determining which commit action to call....261
Creating and removing rows....262
Iterating over the rows in a table....262
Requesting persistence....263
Exposing in-memory or persistent data to dashboards....264
Restrictions affecting MemoryStore disk files....265
Using the distributed MemoryStore....265
Overview of the distributed MemoryStore....265
Distributed store transactional and data safety guarantees....267
Using a distributed store....268
Configuring a distributed store....269
Adding distributed MemoryStore support to a project....269
Adding a distributed store....270
Configuring a distributed store....271
Launching a project that uses a distributed store....272
Interacting with a distributed store....273
Configuration files for distributed stores....274
BigMemory Max driver specific details....277
Changing bean property values when deploying projects....281
Creating a distributed MemoryStore driver....281
Using the Management interface....283
Interfacing with user-defined correlator plug-ins....285
About the chunk data type....286

Making Application Data Available to Clients....287
Adding the DataView service bundle to your project....288
Creating DataView definitions....289
DataViewAddDefinition....289
DataViewDefinition....290
DataViewException....290
Example....290
Deleting DataView definitions....291
DataViewDeleteDefinition....291
DataViewDefinitionDeleted....291
Creating DataView items....292
DataViewAddItem....292
DataViewItem....293
DataViewItemException....293
Example....293
Deleting DataView Items....294

DataViewDeleteItem....294
DataViewItemDeleted....295
Example....295
DataViewDeleteAllItems....296
DataViewAllItemsDeleted....296
Updating DataView Items....296
DataViewUpdateItem....297
Example....297
DataViewUpdateItemDelta....298
Example....298
Looking Up Field Positions....299
DataViewGetFieldLookup....299
DataViewFieldLookup....299
Using multiple correlators....300

Testing and Tuning EPL....301
Optimizing EPL programs....302
Best practices for writing EPL....302
Wildcard fields that are not relevant....303
Avoiding unnecessary allocations....303
Implementing states....303
Structure of a basic test framework....304
Using event files....304
Handling runtime errors....305
What happens....305
Using ondie() to diagnose runtime errors....305
Using logging to diagnose errors....306
Standard diagnostic log output....306
Capturing test output....307
Avoiding listeners and monitor instances that never terminate....307
Handling slow or blocked receivers....308
Diagnosing infinite loops in the correlator....309
Tuning contexts....309
Parallel processing for instances of an event type....310
Parallel processing for long-running calculations....311

Generating Documentation for Your EPL Code....313
Code constructs that are documented....314
Steps for using ApamaDoc....314
Inserting ApamaDoc comments....315
Inserting ApamaDoc tags....316
Inserting ApamaDoc references....318
Generating ApamaDoc in headless mode....319

EPL Naming Conventions....321

EPL Keyword Quick Reference....323

EPL Methods Quick Reference....337
action methods....338

boolean methods....338
chunk methods....338
context methods....338
decimal and float methods....339
dictionary methods....341
event methods....342
Exception methods....343
integer methods....343
listener methods....344
location methods....344
sequence methods....345
StackTraceElement methods....345
stream methods....346
string methods....346
built-in monitor methods....347

Preface

- About this documentation 14
- How this book is organized 14
- Documentation roadmap 15
- Contacting customer support 16

About this documentation

The event 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), which is the new name for Apama MonitorScript, or the Apama Java in-process API. Alternatively, you can define application logic in the Event Modeler, which provides a graphic user interface. The information presented here focuses exclusively on EPL.

Note: Within the product, both EPL and MonitorScript are used and should be treated as synonymous.

Developing Apama Applications in EPL teaches you how to write EPL programs. While some programming experience is assumed, no prior knowledge of EPL is assumed.

Apama Studio provides tutorials that can help you get started with EPL. In Apama Studio, select Tutorials from the Welcome page.

Preface

How this book is organized

The information in this book is organized as follows:

- "Getting Started with Apama EPL" on page 19
- "Defining Monitors" on page 37
- "Defining Event Listeners" on page 55
- "Working with Streams and Stream Queries" on page 97
- "Defining What Happens When Matching Events Are Found" on page 153
- "Implementing Parallel Processing" on page 185
- "Common EPL Patterns" on page 221
- "Using Correlator Plug-ins in EPL" on page 231
- "Making Application Data Available to Clients" on page 287
- "Testing and Tuning EPL" on page 301
- "Generating Documentation for Your EPL Code" on page 313
- "EPL Naming Conventions" on page 321
- "EPL Keyword Quick Reference" on page 323

Preface

Documentation roadmap

On Windows platforms, the specific set of documentation provided with Apama depends on whether you choose the Developer, Server, or User installation option. On UNIX platforms, only the Server option is available.

Apama provides documentation in three formats:

- HTML viewable in a Web browser
- PDF
- Eclipse Help (if you select the Apama Developer installation option)

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.1 > Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

The following table describes the PDF documents that are available when you install the Apama Developer option. A subset of these documents is provided with the Server and User options.

Title	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write Java programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.
Developing Apama Applications in Java	Introduces the Apama Java in-process API and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.

Title	Contents
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to: • Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. • Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. • Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. • Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. • Use the Apama Web Services Client adapter to invoke Web Services. • Use correlator-integrated JMS messaging to reliably send and receive JMS messages in Apama applications.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

Preface

Contacting customer support

You may open Apama Support Incidents online via the eService section of Empower at http://empower.softwareag.com. If you are new to Empower, send an email to `empower@softwareag.com` with your name, company, and company email address to request an account.

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

Preface

1 Getting Started with Apama EPL

- Introduction to Apama Event Processing Language 20
- How EPL applications compare to applications in other languages 21
- About dynamic compilation in the correlator 21
- About Apama Studio development environment 21
- Terminology 22
- Defining event types 26
- Working with events 29

Apama Event Processing Language (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.

Introduction to Apama Event Processing Language

EPL is a flexible and powerful Java-like ”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. EPL programs, called 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.

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 then you inject or load these monitors into a running correlator. As streams of events pass into a correlator, the monitors and their listeners watch for the events or patterns of events that you identified in your monitors. There are a variety of actions that you can specify that you want the correlator to perform when a listener detects an event or event pattern of interest. The most common action is to generate and dispatch a message to an external receiver.

EPL is case-sensitive.

Getting Started with Apama EPL

How EPL applications compare to applications in other languages

EPL is an event-oriented programming language, as opposed to an object-oriented language. While EPL is primarily an imperative language (it provides conditional statements and looping), it is part of an event-processing framework, and that 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, loops, conditions, and procedures (called actions in EPL). But EPL supports a paradigm that is different from that supported by other scripting languages:

- A monitor 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.

EPL requires a different way of developing applications.

Getting Started with Apama EPL

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.

Getting Started with Apama EPL

About Apama Studio development environment

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

- Add and manage the component files that make up the application.
- Write the EPL for your application.
- Specify the adapters, dashboards, and scenarios 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 and scenario files to a Correlator Deployment Package (CDP).

As you add components to your application, Apama Studio 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 feature of Apama Studio is the EPL editor. The EPL editor provides the following support for writing EPL:

- 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
- Abilty to define templates for frequently-used fragments of EPL

In Apama Studio, you can examine the EPL files that are part of the sample applications. On the Apama Studio Welcome page, click Samples > Apama Samples, select the Process Monitor demo, and then double-click a `.mon` file to view it in the EPL editor. If necessary, click the Show All Folders icon to display the monitors.

Getting Started with Apama EPL

Terminology

This topic provides a definition of each important EPL term. The definitons are organized into the following groups:

- "Basic modules" on page 22
- "Data types" on page 23
- "Monitors" on page 24
- "Events" on page 24
- "Streams" on page 25

Table 1. EPL terminology

Basic modules	Application	An Apama application consists of one or more collaborating monitors.

	Package	A mechanism for qualifying monitor and event names. Monitors 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	A monitor is the basic unit of program execution. Monitors have both data and logic. Monitors communicate by sending and receiving events.
	Event (*data type*)	An event is a data object, used primarily for communication between monitors and between the correlator and its environment. 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.
	Field	A data element of an event.
	Method	A method is a pre-defined action. A given EPL data type has a given set of methods that it supports.
Data types	Data type	EPL supports the following value data types: `boolean`, `decimal`, `float`, `integer`, and the following reference data types: `action`, `chunk`, `context`, `dictionary`, `event`, `Exception`, `listener`, `location`, `sequence`, `StackTraceElement`, `stream`, `string`.
	`sequence`	An EPL data type used to hold an ordered set of objects (referenced by position).
	`dictionary`	An EPL data type used to hold a keyed set of objects (referenced by key).
	`location`	An EPL data type that represents a rectangular area in a two-dimensional unitless Cartesian coordinate plane.
	`chunk`	An EPL data type that references an opaque data set, the data items of which are manipulated only in a correlator plug-in.
	`listener`	You can assign an event listener or a stream listener to a variable of this type and then subsequently call `quit()` on the listener to remove the listener from the correlator.
	`action`	An EPL data 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 definitions, event type definitions, and custom aggregate function definitions.

	`context`	An EPL data type that provides a reference to a context. A context lets the correlator concurrently process events.
	`stream`	An EPL data type that refers to a stream object. Each stream is a conduit or channel through which items flow. A stream transports items of only one type, which can be any Apama data type. Streams are internal to a monitor.
	`Exception`	Values of `Exception` type are objects that contain information about runtime errors.
	`StackTrace Element`	A `StackTraceElement` type value is an object that contains information about one entry in the stack trace.
Monitors	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.
	Sub-monitor	A monitor instance. When a monitor spawns it creates a sub-monitor, which is just another monitor instance.
Events	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	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.

	`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: "stream" on page 24, "streamlistener" on page 25, and "from statement" on page 25.	Stream query	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.
	Stream source template	An event template preceded by the `all` 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.
	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.

Getting Started with Apama EPL

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 $80 for a share of IBM stock at noon on May 1, 2013
- Purchase of 1000 shares of IBM stock at $80 per share at 12:01 PM on May 1, 2013
- 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 Apama Studio 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 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 data 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.

The following topics discuss how to define event types:

- "Allowable event field types" on page 27
- "Format for defining event types" on page 27
- "Example event type definition" on page 29

See also "Specifying named constant values" on page 158.

Getting Started with Apama EPL

Allowable event field types

A field in an event can be any Apama data type. The Apama data types are:

- `boolean`
- `decimal`
- `float`
- `integer`
- `string`
- `action`
- `chunk`
- `context`
- `dictionary`
- `event`
- `listener`
- `location`
- `sequence`
- `stream`

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 enqueue or route the event but you cannot emit the event. See "Generating events" on page 170.
- `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 enqueue, route, or emit 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 details about each type allowed in an event, see "event" in the "Types" section of the *Apama EPL Reference*.

Defining event types

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

`event`	This EPL keyword is required. It indicates an `event` type definition.
`event_type`	Replace *`event_type`* 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: `StockTick`.
`{ }`	Enclose the field definitions in curly braces.
`wildcard`	Specify the `wildcard` 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, when the correlator watches for certain events of this type, the value of a wildcard field is always irrelevent. For more details, see "Improving performance by ignoring some fields in matching events" on page 69.
`field_type`	Replace *`field_type`* with the name of a data 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: `dictionary<integer,string>`. See the *Apama EPL Reference*, "Types", or "dictionary", or "sequence".
`field_name`	Replace *`field_name`* with a name that you choose for this field. 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 is needed to start some process. 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 `wildcard` keyword for the additional fields. Note that if the type of an event field is `location`, that field counts as 2. For example, if you have 28 non-`location` type fields and 2 `location` 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 `wildcard` keyword for additional fields, the correlator rejects the file. You must add the `wildcard` keywords to be able to inject the file.
`constant`	Specify the `constant` keyword in front of a field defintion whose type is `boolean`, `decimal`, `float`, `integer`, or `string` and whose value never changes.
`literal`	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 *`field_type`* you specified for this field.
`action_definition`	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 161.

Defining event types

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.

Defining event types

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. An understanding of the order in which the correlator processes events helps you determine where and when to allocate events.

This section discusses the following topics:

- "Methods on events" on page 29
- "Making event type definitions available to monitors" on page 32
- "Allocating events" on page 33
- "Event processing order" on page 35

Getting Started with Apama EPL

Methods on events

You can call the following methods on any `event` type:

- `getName()` retrieves the fully qualified event name. You can call this method on an event type or an event instance. For example:

  ```
  event Foo {
     string bar;
     integer count;
  }

  monitor m {
     action onload() {
     print Foo.getName();
     }
  ```

```
}
```

The previous code prints the following:

```
Foo
```

- `getFieldNames()` retrieves a sequence of strings, `sequence<string>`, that contains the event field names. You can call this method on an event type or an event instance. For example:

```
event Foo {
   string bar;
   integer count;
}

monitor m {
   action onload() {
    print Foo.getFieldNames().toString();
    }
}
```

The previous code prints the following:

```
["bar","count"]
```

- `getFieldTypes()` retrieves a sequence of strings, `sequence<string>`, that contains the event field types. You can call this method on an event type or an event instance. For example:

```
event Foo {
   string bar;
   integer count;
}

monitor m {
   action onload() {
   print Foo.getFieldTypes().toString();
   }
}
```

The previous code prints the following:

```
["string","integer"]
```

- `getFieldValues()` retrieves a sequence of strings, `sequence<string>`, that contains the string version of the event's field values. For `string` type fields, there is no quoting or escaping. You can call this method only on an event instance. For example:

```
event Foo {
   string bar;
   integer count;
}

monitor m {
   action onload() {
   Foo f:=Foo("Hello",1);
   print f.getFieldValues().toString();
   }
}
```

The previous code prints the following:

```
["Hello","1"]
```

- `isExternal()` — returns a `boolean` that indicates whether the event was generated by an external source. Such an event came from an external event sender, triggered an event listener, and was coassigned to a monitor instance variable. When a monitor instance enqueues an event, the event is considered to be generated by an external source.

  A return value of `true` indicates an event that was generated by an external source.

When the correlator spawns a monitor instance, it preserves the value that the `isExternal()` method returns. In other words, if you coassign an external event in a monitor instance, and then spawn that monitor instance, the `isExternal()` method returns true in the spawned monitor instance.

This method takes no parameters.

The `isExternal()` method returns `false` when a monitor instance

- Routes an event that was external
- Creates an event inside the correlator
- Clones an event

This method is useful when you need to determine whether an event came from outside or was in some way derived inside the correlator. Although this distinction is often clear from the event type, that is not always the case.

- `clone()` — returns a new `event` that is an exact copy of the `event`. All the `event`'s contents are cloned into the new `event`, and if they were complex types themselves, their contents are cloned as well. Takes no parameters.
- `getTime()` – returns a `float` that indicates a time expressed in seconds since the epoch, January 1st, 1970. The particular time returned is as follows:
  - If the correlator created this event, the `getTime()` method returns the time that the correlator created the event. This is the creation time in the context in which the correlator created the event.
  - Coassignment sets the timestamp of an event. A call to `getTime()` on a coassigned event returns the time that the correlator performed the coassignment. This is the time in the context in which the correlator performed the coassignment and it can be the time the event was received or routed. For an enqueued event, a call to `getTime()` returns the receiving context's current time when the enqueued event arrived in the context.

    An event's timestamp might not match the time when an event listener for that event fires. For example, consider the following:

    ```
    on A():a and B():b {
       ...
    }
    ```

    Suppose that `currentTime` is `1` when the correlator processes `A` and `currentTime` is `2` when the correlator processes `B`. A call to `a.getTime()` returns `1`, while a call to `b.getTime()` returns `2`. Of course, the event listener fires only after processing `B`.
- `parse()` — method that returns the `event` object represented by the `string` argument. You can call this method on a parseable event type or on an instance of a parseable event type. The more typical use is to call `parse()` directly on the event type. You can call the `string.canParse()` method to determine whether a particular string can be parsed.

  The `parse()` method takes a single string as its argument. This string must be the string form of an `event` object. The string must adhere to the format described in "Event file format" in the "Event Correlator Utilities Reference" section of *Deploying and Managing Apama Applications*. For example:

  ```
  A a := new A;
  a := A.parse("A(10, \"foo\")");
  ```

You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.

See the description of parseable types in the *EPL Reference*, "Type properties summary".

- `toString()` — returns a `string` representation of the event. Takes no parameters. The return value is constructed by calling the `toString()` method on each of the referenced event's fields and catenating the individual return values into the result.

Working with events

Making event type definitions available to monitors

A monitor must have information about the type definitions of the events that it processes. You can provide this information in several ways:

- 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.
- 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.

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 and then use one of the following ways to bring the appropriate event type definition into your monitor:

- In your monitor, specify the fully qualified name of the event type, for example:

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

- In your EPL file, 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 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.

See also "Name precedence" in the "Lexical Elements" section of the *Apama EPL Reference*.

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 Apama Studio 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:
  1. In Apama Studio, in the Apama Developer perspective, in the Developer Project View, select the project name.
  2. Press Alt-Enter.
  3. Select MonitorScript Build Path.
  4. Click the External Dependencies tab.
  5. Click Add External.
  6. Navigate to the event type definition file, and select it.
  7. Click Open.
- 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.
  1. Create the project that contains the common event type definition file and keep that project open in Apama Studio.
  2. In the Developer Project View, select the name of a project that needs to use the common event definition file.
  3. Press Alt-Enter.
  4. Select MonitorScript Build Path.
  5. Select the Projects tab.
  6. Click Add.
  7. Select the project that contains the event definition file, and click OK.

Working with events

Allocating events

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 intialize 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.
   }
```

Working with events

Event processing order

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.

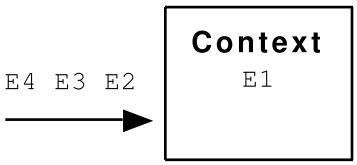
Concurrently, in each context, the correlator

- 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

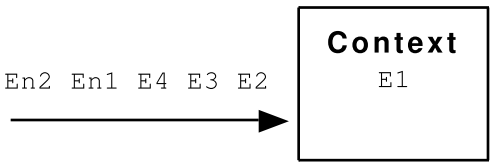
When the correlator processes an event within a given context, it is possible for that processing to route an event. A routed event goes to the front of that context's input queue. The correlator processes the routed event before it processes the other events in that input queue.

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.

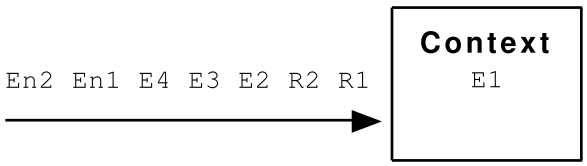
For example, suppose the correlator is processing the `E1` event and events `E2`, `E3`, and `E4` are on the input queue in that order.



While processing `E1`, suppose that events `En1` and `En2` are created in that order and enqueued. These events go to the special queue for enqueued events. Assuming that there is room on the input queue of each public context, the enqueued events go to to the end of the input queue of each public context:



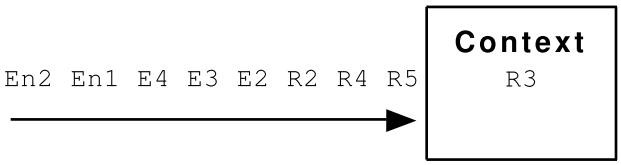
While still processing `E1`, suppose that events `R1` and `R2` are created in that order and routed. These events go to the front of the queue:



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 89.

Working with events

2 Defining Monitors

- About monitor contents 38
- Example of a simple monitor 40
- Spawning monitor instances 42
- Communication among monitor instances 48
- About service monitors 51
- Adding a bundle to your project 52

A monitor is the basic unit of EPL program execution. 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. When you inject files into the correlator, be sure to load event type definitons before you load the monitors that process events of those types.

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 the *Apama EPL Refererence* for a list of keywords.

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.

Defining Monitors

Loading monitors into the correlator

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

At any time, you can use the correlator utility, `engine_inject`, to load EPL files into the correlator. See "Injecting EPL code" in the "Event Correlator Utilities Reference" section of *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` utility.
- Write a program in C, C++, Java, or .NET and use the corresponding Apama client API.
- Use the Apama Management & Monitoring tool.

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 {
    …
}
```

About monitor contents

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 151.

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

About monitor contents

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

About monitor contents

Example of a simple monitor

The empty monitor discussed in "About monitor contents" on page 38 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` command 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 56.

The `all` keyword extends the `on` command 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 assignment operator (`:`) to place that event in a *global* or *local* variable. For example:

```
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 154.

About the processTick() action

The `processTick()` action executes the `log` command 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 178.

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` command requires strings as fields, so the `processTick()` action specifies the built-in `.toString()` operation on the nonstring value:

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

Defining Monitors

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 is by spawning monitor instances.

The following topics discuss spawning:

- "How spawning works" on page 43
- "Sample code for spawning" on page 44

- "Terminating monitor instances" on page 46
- "About executing ondie() actions" on page 47
- "Specifying parameters when spawning" on page 47

See also "Spawning to contexts" on page 190.

Defining Monitors

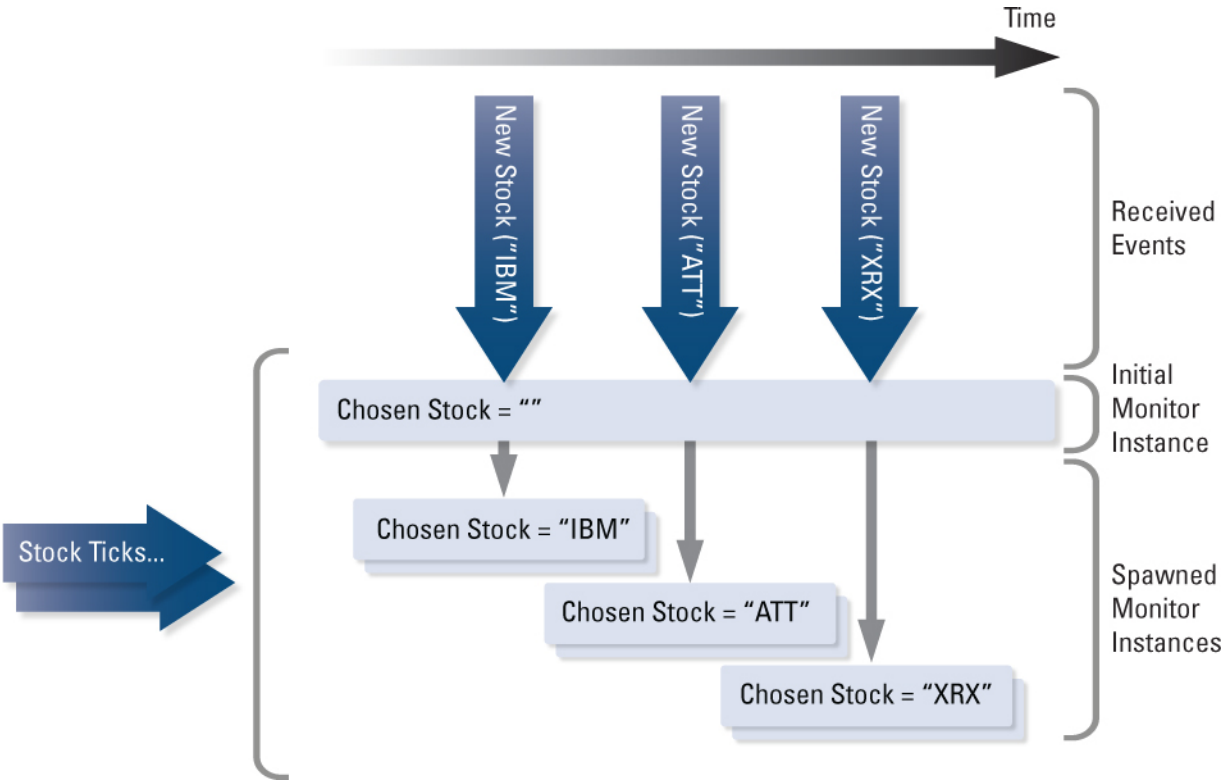
How spawning works

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 spawining 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:

Figure 1. Spawning 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.

Spawning monitor instances

Sample code for spawning

EPL that implements the example described in "How spawning works" on page 43 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 + " 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 40is 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 162.

Spawning monitor instances

Terminating monitor instances

The example discussed in "Sample code for spawning" on page 44 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 instace 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.

Spawning monitor instances

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.
- The `die()` operation is called on it.
- The `engine_delete` utility or an Apama client API removes the monitor from the correlator.
- A run-time 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.

Spawning monitor instances

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) {
      Event e;
      on all Event(arg):e {
         emit e to channel;
      }
      on StopForwarding(arg) {
         die();
      }
   }
}
```

The following are equivalent:

```
spawn actionName(); // This is the correct syntax.
spawn actionName;   // This is deprecated. Do not use it.
```

Spawning monitor instances

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. Also, you might want to use the MemoryStore correlator plug-in to share state between monitors. See "Using the MemoryStore" on page 252.

The following topics provide information for making these decisions:

- "Organizing behavior into monitors" on page 48
- "Sending events to other monitors" on page 49
- "Defining your application's message exchange protocol" on page 50
- "Using events to control processing" on page 51

Defining Monitors

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 51.

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.

Communication among monitor instances

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" on page 35.
- 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 enqueue an event to the context that contains the target monitor instance. The target monitor instance must be listening for the enqueued event. See "Enqueuing an event to a particular context" on page 191.

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 67.

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 command" on page 171.

Communication among monitor instances

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

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 emit an `OrderProcessingState` event each time there is a significant state change.

Communication among monitor instances

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.

Communication among monitor instances

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:

- Dataview service — exposes read-only data to dashboards. This data comes from EPL and Java applications.
- Password service — supports retrieval of passwords from implementation-specific providers.
- Scenario service — provides support for all scenario-based applications.

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

- ADBC adapter — provides event capture and playback in conjunction with Apama Studio's Data Player. Also monitors Java database connectivity (JDBC) and open database connectivity (ODBC).
- IAF status manager — monitors connectivity with an adapter.

Defining Monitors

Adding a bundle 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 Apama Studio.

To add a bundle to your project:

1. In the Apama Developer perspective, open the project that you want to add the bundle to.
2. In the Developer Project View, right-click the project name and select Properties from the menu that appears.
3. In the Properties dialog, select MonitorScript Build Path.
4. Select the Bundles tab.
5. Click Add to display a list of Apama Studio bundles.
6. Select the bundle you want to add.
7. Click OK twice.

The bundle now appears in the Developer Project View panel. Expand the bundle directory to see the contents. To understand exactly what each service monitor provides, open the service's EPL file in Apama Studio. 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.

Defining Monitors

Utilities for operating on monitors

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

- `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, sentinel agent processes, and continuous availability processes.

Adding a bundle to your project

3 Defining Event Listeners

- About event expressions and event templates 56
- Specifying the on statement 58
- Using a stream source template to find events of interest 59
- Defining event expressions with one event template 60
- Terminating and changing event listeners 63
- Specifying multiple event listeners 65
- Listening for events that do not match 66
- Specifying completion event listeners 67
- Improving performance by ignoring some fields in matching events 69
- Defining event listeners for patterns of events 69
- Specifying and/or/not logic in event listeners 71
- How the correlator executes event listeners 76
- Defining event listeners with temporal constraints 83
- About timers and their trigger times 88
- Understanding time in the correlator 89
- Disabling the correlator's internal clock 90
- Out of band connection notifications 92

An `on` statement specifies an event expression and a listener action. When the correlator executes an `on` statement it creates an event listener. An event listener observes each event processed by the correlator 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 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:

Table 2. Event expressions that specify one event template

Event Expression	Description
`StockTick(*,*)`	The event listener that uses this event expression is interested in all `StockTick` events regardless of the event's field values.
`NewsItem("ACME",*)`	The event listener that uses this event expression is interested in `NewsItem` events that have a value of `ACME` in their first field. Any value can be in the second field.
`ChainedResponse(reqId="req1")`	The event listener that uses this event expression is interested in `ChainedResponse` events whose `reqId` field has a value of `req1`. If a `ChainedResponse` event has any other fields, their values are irrelevent.

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.

Table 3. Operators used in an event expression

Category	Operator	Operation
Followed by	`->`	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	`all`	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	`and`	Logical intersection. The event listener detects a match after it finds events that match the event templates on both sides of the `and` operator. The order in which the listener detects the matching events does not matter.
	`not`	Logical negation. The event listener detects a match only if an event that matches the event template that follows the `not` operator has not occurred.
	`or`	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 `or` operator.
	`xor`	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 `xor` operator. For example, consider this event: `A(1,1)`. This event does not trigger the following listener because it matches the event templates on both sides of the `xor` operator: `on A(1,*) xor A(*,1).`
Temporal operators	`at`	The event listener triggers at specific times or repeatedly at a specified interval.
	`wait`	Limits the amount of time that an event listener can detect a match.
	`within`	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.

Defining Event Listeners

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

identifier	Optionally, you can specify a variable of type `listener` 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 `quit()` method on the listener.
event_expr	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.

coassignment	Optionally, you can coassign the matching event to a 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.
listener_action	The statement or block that you want the correlator to perform when the event listener triggers.

Examples

For example:

```
on StockTick(*,*) processTick();
```

In this example, the event expression contains one event template: `StockTick(*,*)`. The asterisks indicate that the values of the `StockTick` event's two fields are not relevent when matching. When this event listener detects a `StockTick` event, the listener triggers and causes the correlator to execute the `processTick()` listener action.

Following is an example that 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(*,*):newTick processTick();
```

Suppose that after finding a matching event, the listener action includes specification of an `on` statement. For example:

```
listener l := on StockTick(*,*):newTick {
   on StockTick(newTick.symbol, > newTick.value):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.

Defining Event Listeners

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 97.

Defining Event Listeners

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.

This section discusses the following topics:

- "Listening for one event" on page 60
- "Listening for all events of a particular type" on page 60
- "Listening for events with particular content" on page 61
- "Using positional syntax to listen for events with particular content" on page 61
- "Using name/value syntax to listen for events with particular content" on page 61
- "Listening for events of different types" on page 62

Defining Event Listeners

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 irrelevent, 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.

Defining event expressions with one event template

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.

Defining event expressions with one event 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. 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();
```

Defining event expressions with one event template

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):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();
```

Defining event expressions with one event template

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;
}
```

Table 4. Comparison of positional and name/value syntax in event expressions

Comparison Criterion	Positional Syntax	Equivalent Name/Value Syntax
Equality	`on A(3,"string")` `on A(=3,="string")`	`on A(a=3,b="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(*,*,*)`
		`on C(b=4)`
		`on C(a=*,b=4,c=*)`
		`on C()`

For details about the operators and expressions that you can specify in event templates, see the "Expressions" section of the *Apama EPL Reference*.

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")`

Defining event expressions with one event template

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 40 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`.

Defining event expressions with one event template

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 62 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 73.

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 62 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.

Defining Event Listeners

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` commands 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(*,*):newTick1 { print newTick1.name; }
on all StockTick(*,*):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"):currentStock processTick();
}
action action2() {
   on all StockChoice("ACME"):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 42 for another way to have multiple event listeners.

Defining 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.

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 TiickProcessor {
   Tick tick;
   ...
   action onload() {
      on all unmatched Tick():tick spawn processTick();
   }
   action processTick() {
      on all Tick( stock=tick.stock ) ...;
   }
   ...
}
```

See also:

- "Example using unmatched and completed" on page 68.
- "Writing echo monitors for debugging" on page 229

Defining Event Listeners

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;
   BestPriceReply reply;
   sequence< float > prices;

   action onload() {
      on all RequestSmartBestPrice(*,*):request spawn getPrices();
   }

   action getPrices() {
      on all BestPriceReply( request.id, * ):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 );
      }
}
```

Defining Event Listeners

Example using unmatched and completed

The followiing 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) : a {
   print "Match: " + a.toString();
   a.count := a.count+1; // count is second field of A
   route A;
}

on all completed A("foo", < 10) : a {
   print "Completed: " + a.toString();
}

on all unmatched A(*,*): 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 229.

Specifying completion event listeners

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 relevent in match conditions but it might be input to calculations within an action block, or it might need to be included in an emitted event.

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

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 83) has been applied to any part of the event expression.

Defining Event Listeners

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.

The following topics provide information for doing this:

- "Specifying the or operator in event expressions" on page 72
- "Specifying the and operator in event expressions" on page 72
- "Specifying the not operator in event expressions" on page 73

- "Specifying and not logic to terminate event listeners" on page 73

Defining Event Listeners

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 and/or/not logic in event listeners

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.

Specifying and/or/not logic in event listeners

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.

Table 5. Event expressions using and/or logic in event listeners

Event Expression	Description
`A -> (B or C)`	Match on an `A` followed by either a `B` or a `C`.
`(A -> B) or C`	Match on either the pattern `A` followed by a `B`, or just a `C` on its own.
`A -> ((B -> C) or (C -> D))`	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 `or` operator treats its operands atomically, that is, in this case it is looking

Event Expression	Description
	for the patterns themselves rather than their constituent events.
`(A -> B) and (C -> D)`	Find the pattern `A` followed by a `B` (that is, `A -> B`) followed by the pattern `C -> D`, or else the pattern `C -> D` followed by the pattern `A -> B`. The `and` 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 {`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 patterns 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'`}

Specifying and/or/not logic in event listeners

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 77.

Specifying and/or/not logic in event listeners

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 aparticular stock while the market is open.

```
event Price {
   string stock;
   float price;
}
Price p;
on all Price("IBM",>targetPrice):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.

Specifying and/or/not logic in event listeners

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):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.

Specifying and not logic to terminate event listeners

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 74, the event listeners would be watching for these events:

- `Pattern_1`
- `PatternMatched`
- `Pattern_2`
- `Pattern_3`

Initially, all `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 terminate event listeners

How the correlator executes event listeners

An understanding of how the correlator executes event listeners can help you correctly define event listeners.

The following topics provide the needed background:

- "How the correlator evaluates event expressions" on page 76
- "Avoiding event listeners that trigger upon instantiation" on page 77
- "When the correlator terminates event listeners" on page 77
- "How the correlator evaluates event listeners for a series of events" on page 78
- "Evaluating event listeners for all A-events followed by B-events" on page 78
- "Evaluating event listeners for an A-event followed by all B-events" on page 80
- "Evaluating event listeners for all A-events followed by all B-events" on page 81

Defining Event Listeners

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 89.

How the correlator executes event listeners

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 75.

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`.

How the correlator executes event listeners

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 63.

How the correlator executes event listeners

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 *Xn* indicates an event instance that matches the event template *X*. Likewise, *Xn+1* indicates another event instance that matches against *X*, but which need not necessarily be identical to *Xn*.

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.

How the correlator executes event listeners

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 `A`s 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() )
```

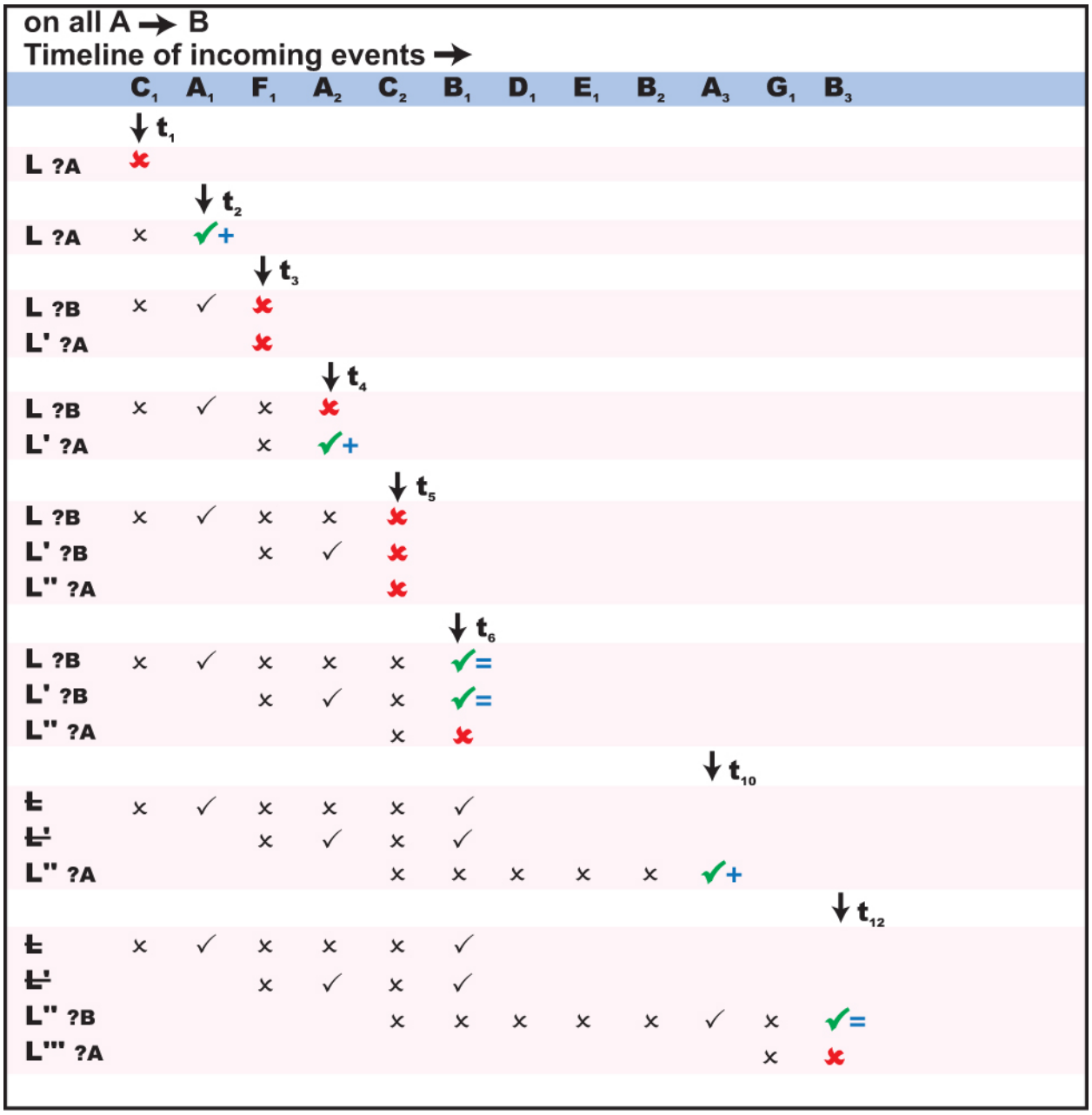
The following table illustrates how the execution of `on all A() -> B()` proceeds over time as the pattern of input events is processed by the correlator. The timeline is from left to right, and each stage is labeled with a time `tn`, where `tn+1` occurs after `tn`. To the left are listed the event listeners, 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:

- ↓ indicates a specific point in time when a particular event is received
- ✱ indicates that at that time no match was found
- ✓ indicates that the listener has successfully located an event that matches its current active template
- = is used to indicate that a listener has successfully triggered
- \+ indicates that a new listener is going to be created.

The master 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`.

Figure 2. Event listeners for all A events followed by B events



How the correlator executes event listeners

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:

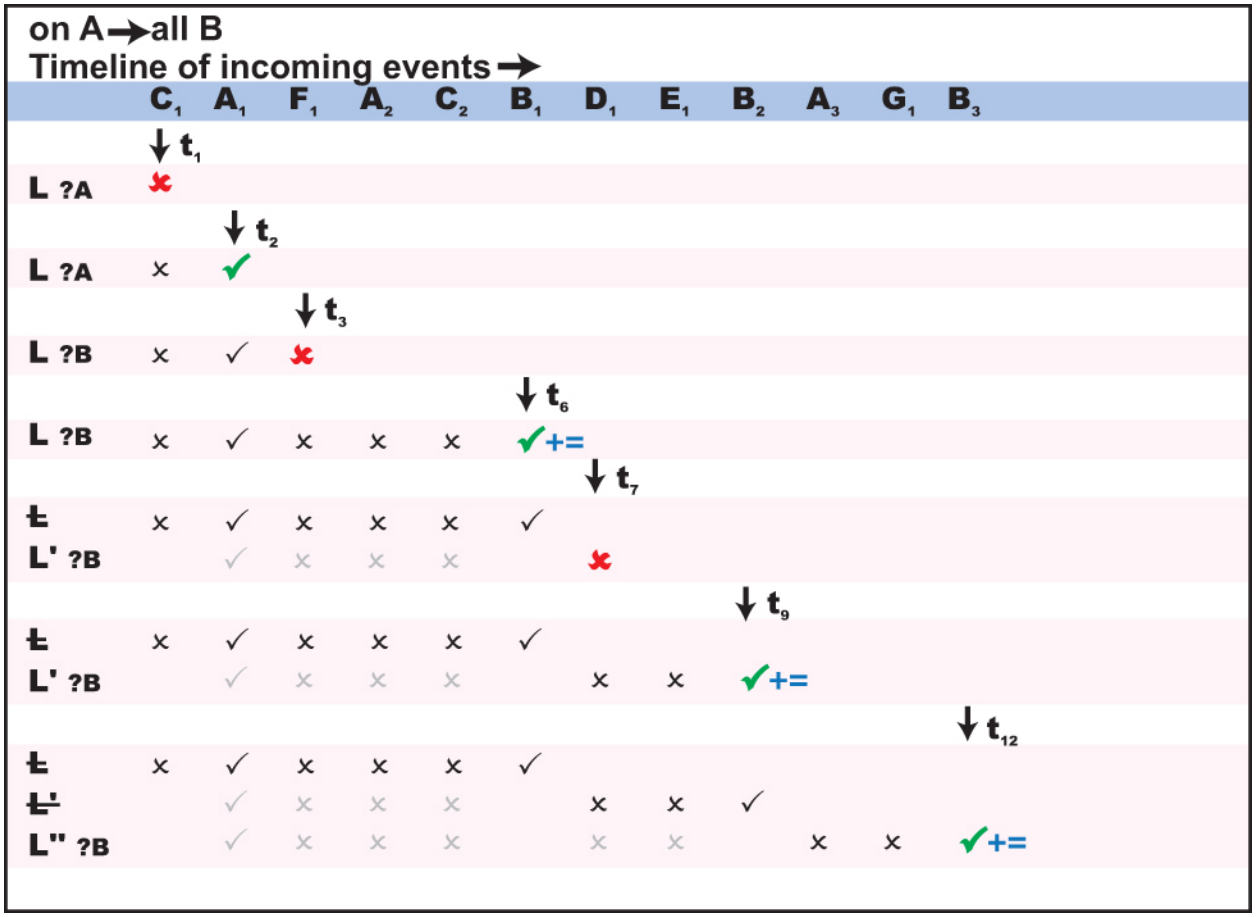
```
on ( A() ->  ( all B() ) ) success();
```

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() -> all B()` would match {`A1`, `B1`}, {`A1`, `B2`} and {`A1`, `B3`}.

Graphically this would now look as follows:

Figure 3. Event listeners for an A event followed by all B events



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 master event listener denoted by `on (A() -> all B())` will never terminate as there will always be a sub-event-listener looking for a `B`.

How the correlator executes event listeners

Evaluating event listeners for all A-events followed by all B-events

Consider the following event listener defintion:

```
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 `A`s. 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.

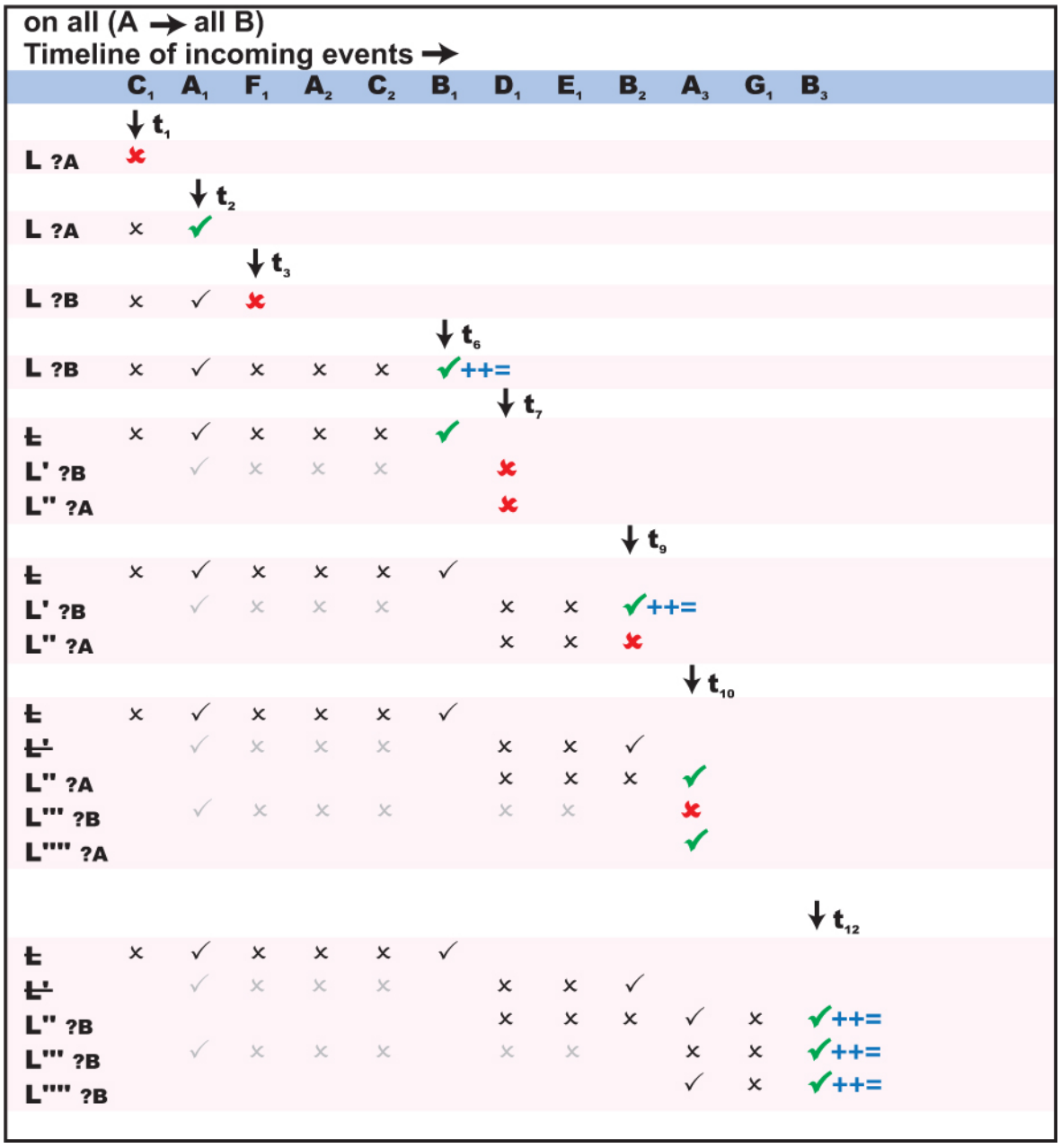
Now consider the example,

```
on all ( A() -> all B() ) success();
```

This will match the first `A` followed by all subsequent `B`s. 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 `B`s 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 78, this would be evaluated as follows:

Figure 4. Event listeners for all A events followed by all B events



Thus matching against {A1, B1}, {A1, B2}, {A1, B3}, and twice against {A3, B3}. Notice how the number of active event listeners is progressively increasing, until after t12 there would actually be six active event listeners, three looking for a B and three looking for an A.

How the correlator executes event listeners

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.

This section discusses the following topics:

- "Listening for event patterns within a set time" on page 84
- "Waiting within an event listener" on page 85
- "Triggering event listeners at specific times" on page 86
- "Using variables to specify times" on page 87

Defining Event Listeners

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 `NewsItem` 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.

Defining event listeners with temporal constraints

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 instaniates 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.

Defining event listeners with temporal constraints

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

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
`minutes`	`0` to `59`, indicating minutes past the hour.
`hours`	`0` to `23`, indicating the hours of the day.
`days_of_month`	`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.
`months`	`1` to `12`, indicating months of the year, with `1` corresponding to January
`days_of_week`	`0` to `6`, indicating days of the week, where `0` corresponds to Sunday.
`seconds`	`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.

Defining event listeners with temporal constraints

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.

For example,

```
string minutes = "*/30";
on all at(minutes,9:17,[*/2,1],*,*) success();
```

shows how this can be done. Each of the parameters can be replaced with a `string` variable in this way.

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;
```

Sequences are described in "sequence" in the "Types" section of the *Apama EPL Reference*.

Defining event listeners with temporal constraints

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();
  ```

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 exacty 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) then {
         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.

Defining Event Listeners

Understanding time in the correlator

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 each input queue.

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 actions. However, the correlator uses event timestamps, and not real time, to determine when to trigger actions.

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 it 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 size of an input queue never affects temporal comparisons.

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.

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 benefit 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 also "Generating events that keep time" on page 91.

Defining Event Listeners

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 correlator plug-in.)

Defining Event Listeners

Generating events that keep time

`&TIME` events have the following format:

```
&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.

When the correlator processes an `&TIME` event by taking it off an input queue, the correlator sets its internal time (the current time) 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.

Disabling the correlator's internal clock

About repeating timers

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" on page 92.

Disabling the correlator's internal clock

Setting the time in the correlator

The format of an `&SETTIME` event is as follows:

```
&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`.

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 *Deploying and Managing Apama*, "Event Correlator Utilities Reference", "Starting the correlator", "Determining whether to disconnect slow receivers".

Disabling the correlator's internal clock

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 to 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:

- ReceiverConnected
- SenderConnected
- ReceiverDisconnected
- SenderDisconnected

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.

Defining Event Listeners

Out of band notification events

The out of band events are defined as follows:

```
package com.apama.oob;
// Note that while the logicalId and physicalId are integers, they are
// unsigned 64-bit values. Using EPL integer types would result in some
// IDs being negative, and thus not matching the values given in log files.
/** Notification that a sender has connected */
event SenderConnected {
  /**
  * Component name, as supplied with the -N command line argument
  * to iaf/correlator or engineInit method
  */
  string componentName;
  /**
  * Representation of the address component is connecting from
  */
  string address;
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string logicalId;
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string physicalId;
  }
/** Notification that a sender has disconnected */
event SenderDisconnected {
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string logicalId;
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string physicalId;
```

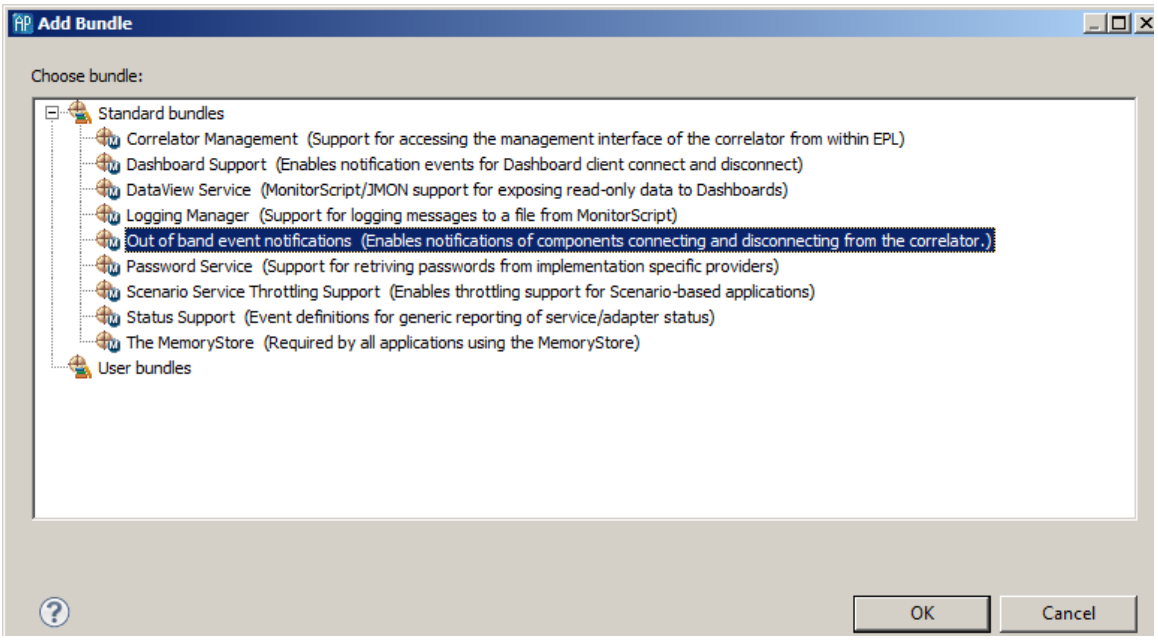
```
  }
/** Notification that a receiver has connected */
event ReceiverConnected {
  /**
  * Component name, as supplied with the -N command line argument
  * to iaf/correlator or engineInit method
  */
  string componentName;
  /**
  * Representation of the address component is connecting from
  */
  string address;
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string logicalId;
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string physicalId;
  }
/** Notification that a receiver has disconnected */
event ReceiverDisconnected {
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string logicalId;
  /**
  * Opaque representation of IDs; these are unique per
  * instance of a process.
  */
  string physicalId;
  }
```

Out of band connection notifications

Enabling out of band notifications

To enable out of band notifications in your Apama applications, you add the Out of band event notifications bundle to your project in Apama Studio.

1. From Project Explorer right-click on the project and select Apama > Add Bundle from the pop-up menu. The **Add Bundle** dialog is displayed.



2. From the **Add Bundle** dialog, select the Out of band event notifications bundle and click OK. The bundle is added to your Apama project.

   The Out of band event notifications bundle contains the event definitions and the monitor that enables the notifications.

3. In you Apama application, create a listener for out of band events specific to the components you are interested in.

Note, you can also enable out of band notifications for a correlator with the `engine_management` utility using the `engine_management -r "setOOB on"` command. You should also inject the event definitions before running that command. For more information about using the `engine_management` utility, see "Shutting down and managing components" in *Deploying and Managing Apama Applications*.

Out of band connection notifications

4 Working with Streams and Stream Queries

- Introduction to streams and stream networks 98
- Defining streams 99
- Using output from streams 100
- Defining stream queries 102
- Defining custom aggregate functions 131
- Working with lots that contain multiple items 134
- Stream network lifetime 138
- Using dynamic expressions in stream queries 141
- Troubleshooting and stream query coding guidelines 147

EPL lets you create queries that operate on streams of items to generate more valuable streams that contain derived items. Derived items can be events, `location` types or simple types (`boolean`, `decimal`, `float`, `integer`, `string`). In stream queries, 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/monitorscript` directory of your Apama installation directory. There is also an introductory whitepaper, *Apama EPL Streams: A Short Tour*, available in the `doc\pdf` 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 99). 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 102) 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 100) 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 108).

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 134, and "Coassigning to sequences in stream listeners" on page 102).

The items in a lot are always ordered, and the lots themselves are always ordered.

Working with Streams and Stream Queries

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 data 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 emit 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 99.
- From the result of a stream query on some other stream. See "Defining stream queries" on page 102.

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

Note: It is permissible to define a `stream` variable that references a stream of `stream` type items. In such a definition, be sure to insert a space between the consecutive right-angle brackets. For example: `stream<stream<float> >`. You must insert this extra space in all stream definitions that contain a type that encloses another type. For example: `stream<sequence<integer> >`.

Working with Streams and Stream Queries

Creating streams from event templates

A stream can be created from an event template using the `all` keyword. This is referred to as a stream source template. For 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 any operators such as `and` or followed-by to combine several event templates. See also "Stream network lifetime" on page 138.

Defining streams

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 steams.

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 138. It is also possible to terminate connected streams by quitting a stream listener.

Defining streams

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: 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 : a {
   /* Code here executes whenever the query produces output. */
}
```

The syntax for the first form is as follows:

`[`*`listener`*`:= ] from` *`streamExpr : variable statement`*

`listener`	Optional. You can specify a `listener` variable to refer to the stream listener that the `from` statement creates. You can declare a new `listener` variable or a use an existing `listener` variable.
`streamExpr`	Specifies any expression of type `stream` except a stream query. This can be, for example, a `stream` variable or a stream source template. If you want to specify a stream query, use the other form of the `from` statement.
`variable`	Specifies a variable that you want to use to hold the stream output. You must have already declared the variable and the type of the variable must be the same type as the stream output. The `from` statement coassigns the stream output to this variable. For details about the characters you can specify, see "Identifiers" in the "Lexical Elements" section of the *Apama EPL Reference*.

	The output from a stream is referred to as a *lot*. 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 `from` statement coassigns each item, in turn, to the variable. See "Working with lots that contain multiple items" on page 134. A `from` statement cannot specify multiple coassignments.
statement	Specifies an EPL statement. Specify a single statement or enclose multiple statements in braces. The `from` statement coassigns each stream output item to the specified variable and executes the statement or block once for each output item. If the steam output is a lot that contains more than one item, and you want to execute the statement or 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 102.

The syntax for the second form of the `from` statement is as follows:

```
[listener:=] StreamQueryDefinition : variable statement
```

listener	Optional. You can specify a `listener` variable to refer to the stream listener that the `from` statement creates. You can declare a new `listener` variable or a use an existing `listener` variable.
StreamQueryDefinition	Specifies a stream query. See "Defining stream queries" on page 102.
variable	Specifies a variable that you want to use to hold the query results. You must have already declared the variable and the type of the variable must be the same type as the query results. The `from` statement coassigns the query result to this variable. For details about the characters you can specify, see "Identifiers" in the "Lexical Elements" section of the *Apama EPL Reference*. If the query outputs lots that contain more than one item, the `from` statement coassigns each item in the lot, in turn, to the variable. See "Working with lots that contain multiple items" on page 134. A `from` statement cannot specify multiple coassignments.
statement	Specifies an EPL statement. You can specify a single statement or you can enclose multiple statements in braces. The `from` statement coassigns each stream output item to the specified variable and executes the statement or block once for each output item. If you want the statement to be executed once per lot rather than once per item coassign the results to a `sequence`. See "Coassigning to sequences in stream listeners" on page 102

Working with Streams and Stream Queries

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 138.

Using output from streams

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 108.

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 */
}
```

Using output from streams

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 : 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.

`a`	This is an identifier that represents the current item in the stream being queried. See "Specifying input streams in from clauses" on page 106.
`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.

Working with Streams and Stream Queries

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:

```
float vwap;
from p in (from t in ticks where t.price > threshold select t.price)
within period
select wavg(t.price,t.volume): 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 : 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 : tickPrice{
   print tick.price.toString();
}
```

Defining stream queries

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 VWAP (Volume-Weighted Average Price) for a stock you can add the following `from` statement to a monitor:

```
float vwap;
from t in all Tick(symbol="APMA")
   within period
   select wavg(t.price,t.volume) : 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 sample application, which you can find in the `samples\monitorscript\statarb` directory of your Apama installation directory:

```
action newStatArbOrder(StatArbOrder o) {
   integer BUY:=1, HOLD:=0, SELL:=-1, instruction;

   stream<float> spreads:=
      from a in all Price(symbol=o.primary.symbol) retain 1
      from b in all Price(symbol=o.secondary.symbol) retain 1
      select (a.price - b.price);

   stream<MeanSd> meanSds := from s in spreads within 20.0
      select MeansSd(mean(s), stddev(s) );

   stream<integer> comparison := from s in spreads from m in meanSd
      select compareSpreadAndBands(s, m.mean, m.sd, o.factor);

   stream<integer> prevComparison := from c in comparison
      retain 1
      select rstream c;

   from c in comparison from p in prevComparison
      where c!=HOLD and c!=p select c: instruction {
         if instruction = BUY {
            buyPrimarySellSecondary();
         } else {
            sellPrimaryBuySecondary();
         }
      }
}
```

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.

Defining stream queries

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
FromClause	Required	Specifies the input stream for the query. See "Specifying input streams in from clauses" on page 106. 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 108. If a second `from` 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 119.
JoinClause	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 120. 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 108.
WhereClause	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 122.
ProjectionDefinition	Required	Defines how the query generates output items. See "Generating query results" on page 123.

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 steam queries:

```
integer a := 42;
float p;
from a in ticks select a.price:p {
   print a.toString(); // Prints "42" rather than one of the ticks. }
```

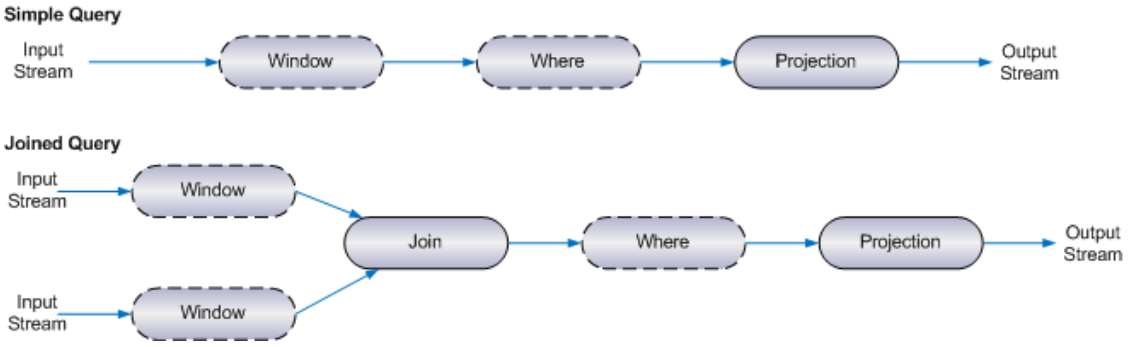
The previous code fragment illustrates that identifiers in the listener action can have the same name as identifiers in the stream query. While this is not good practice, it is important to recognize that the

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.

Defining stream queries

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 108

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 118

The `where` clause, if there is one, filters items. See "Filtering items before projection" on page 122.

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 104. See "Generating query results" on page 123.

Defining stream queries

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 description

itemIdentifier	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" in the "Lexical Elements" section of the *Apama EPL Reference*.

	The type of the identifier is the same as the type of the items that are in the stream you are querying. 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.
streamExpr	Specify an expression that returns a `stream` type. This is the stream that you want to query.
WindowDefinition	Define which portion of the stream to query. See "Adding window definitions to from and join clauses" on page 108.

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 123.

```
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 103.) 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 118.

Defining stream queries

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 124.

If the projection is a simple, non-aggregate projection, the default output is the insertion stream or *istream* for short, of new projected items. Alternatively, if the `restream` keyword is specified in the `select` clause, the output is the remove stream (or *rstream*) of items that have become obsolete.

Defining stream queries

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 description

partitionByExpr	Optionally specifies an EPL expression that should involve the input item in some way and that returns a comparable type. A `partition by` clause effectively creates a separate window for each encountered distinct value of *partitionByExpr*.
windowDurationExpr	Specifies a `float` expression that indicates a duration of a number of seconds. The window contains the items received within the last *windowDurationExpr* seconds. See "Defining time-based windows" on page 110.

batchPeriodExpr	Specifies a `float` expression that indicates an interval period of a number of seconds. The window updates its contents every *batchPeriodExpr* seconds. See "Defining batched windows" on page 113.
windowSizeExpr	Specifies an `integer` expression that indicates the number of items you want to retain in the window. The window contains the most recent *windowSizeExpr* items. See "Defining size-based windows" on page 111.
keyExpr	Specifies an EPL expression that must contain at least one reference to the input item and must return a comparable type. See "Comparable types" in the "Types" section of the *Apama EPL Reference*. 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. See "Defining content-dependent windows" on page 117.
batchSizeExpr	Specifies an `integer` expression that indicates a number of items. The window updates its contents after every *batchSizeExpr* items that match the query are found. See "Defining batched windows" on page 113.

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(): 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 134.

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 131.

Adding window definitions to from and join clauses

Defining time-based windows

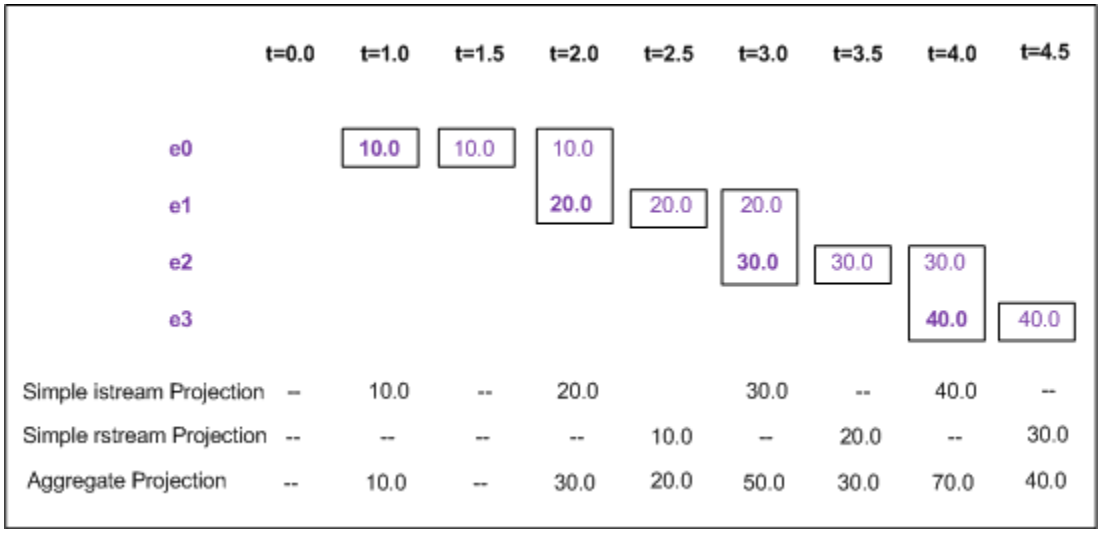
In a time-based window, the items are held in the window for a specific duration. The syntax for defining a time-based window is:

```
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.

Adding window definitions to from and join clauses

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.

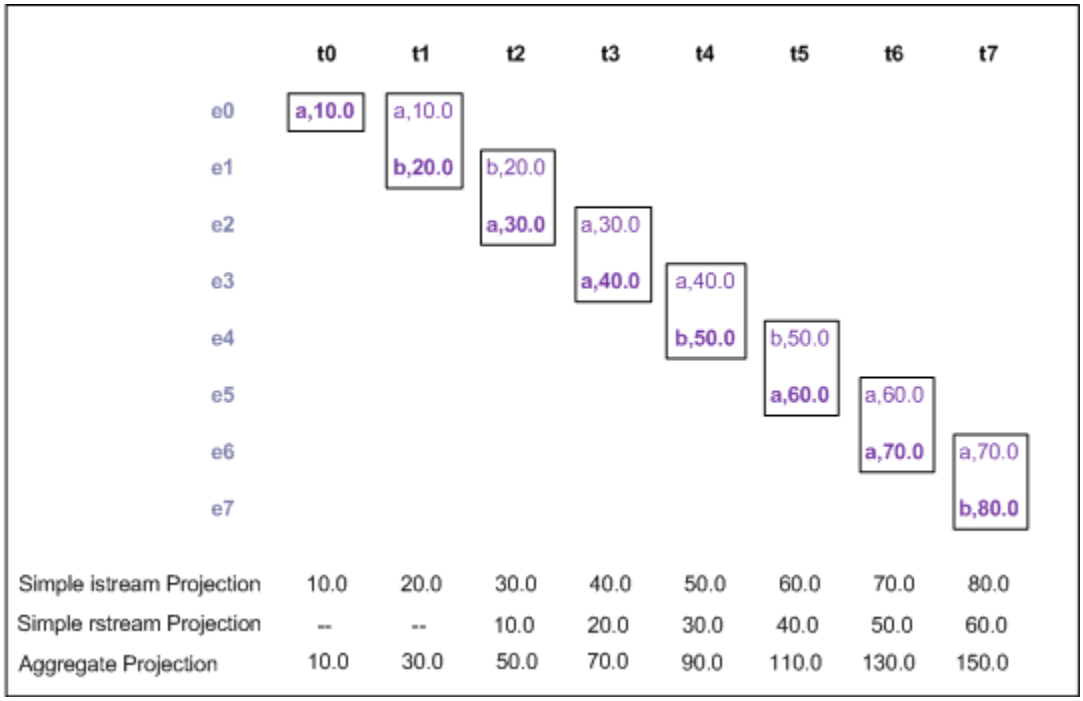
The syntax for defining a size-based window is as follows:

```
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 2 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.

Adding window definitions to from and join clauses

Combining time-based and size-based windows

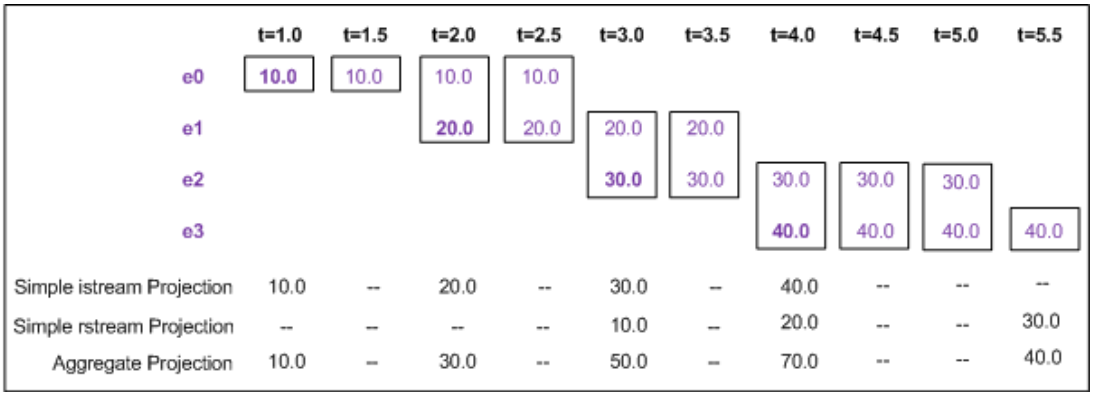
Sometimes you might want to focus on the last *n* items received in the last *d* seconds. To define a window that retains items based on both time and size, use the following format in the `from` clause:

```
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.

Adding window definitions to from and join clauses

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.

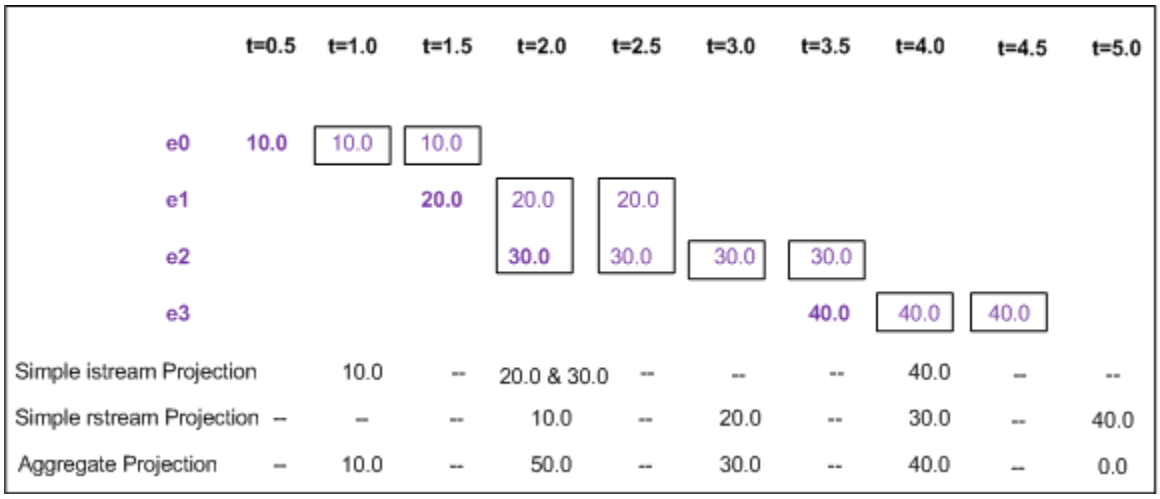
When you specify `within` followed by `every` followed by `retain`, the `every` keyword always indicates a number of seconds. That is, the window updates its content every *p* seconds.

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 things 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.

Adding window definitions to from and join clauses

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" in the "Types" section of the *Apama EPL Reference*.) 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() ...
```

`partition by a.x`	Partition on a single primitive type field of the input event. This is likely to be the most common case.

`partition by a`	Partition on an event's field values. The events that have identical values for all fields are in the same partition. For example:
	`from a in all A()`
	`  partition by a retain 2 select a;`
	Given the following input events:
	`A(1,1)`
	`A(1,2)`
	`A(1,1)`
	The first and third events are in the same partition, the second is not. In this case, the event type `A` must itself be a comparable type.

`partition by 1`	This is a valid partition expression, but it is not recommended. A partition expression should reference the input item in some way.
`partition by f(a)`	This is a valid partition expression if `f()` is a function that returns an appropriate type.
`partition by a.x*globaldict[a.y]`	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.

Adding window definitions to from and join clauses

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 128 . 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 130.

Adding window definitions to from and join clauses

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.

Adding window definitions to from and join clauses

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 117. 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.

Adding window definitions to from and join clauses

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:

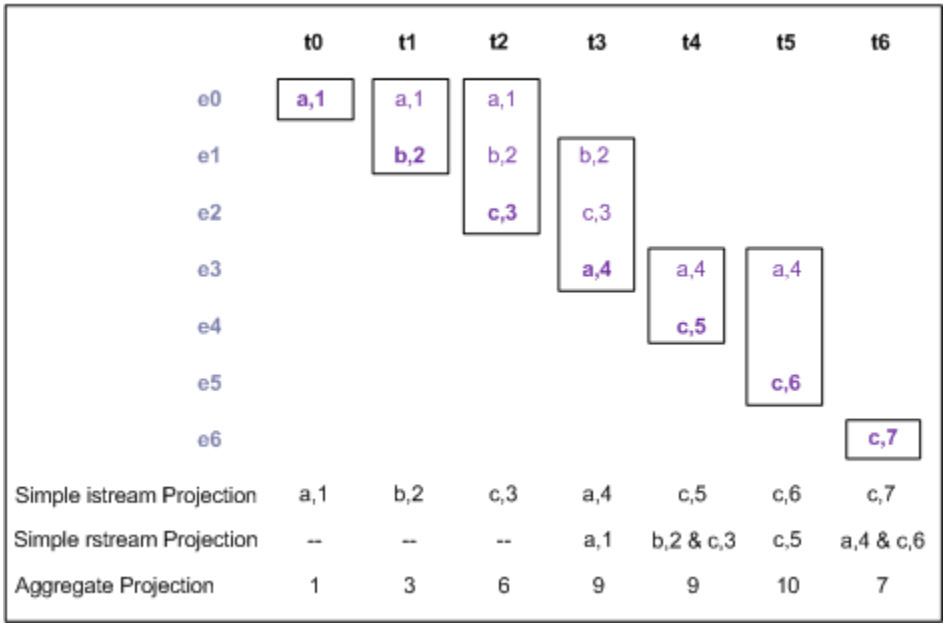
```
with unique keyExpr
```

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" in the "Types" section of the *Apama EPL Reference*.) Some examples are in the following table.

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 3 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 3 values for each key" and "using `with unique` gives you one value of each key, from the last 3". 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 130.

Adding window definitions to from and join clauses

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. Each form takes two input streams and produces a

single output stream of combined items. 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.

Join operations, particularly cross-joins, can create many more output events than input events, not just the same or fewer.

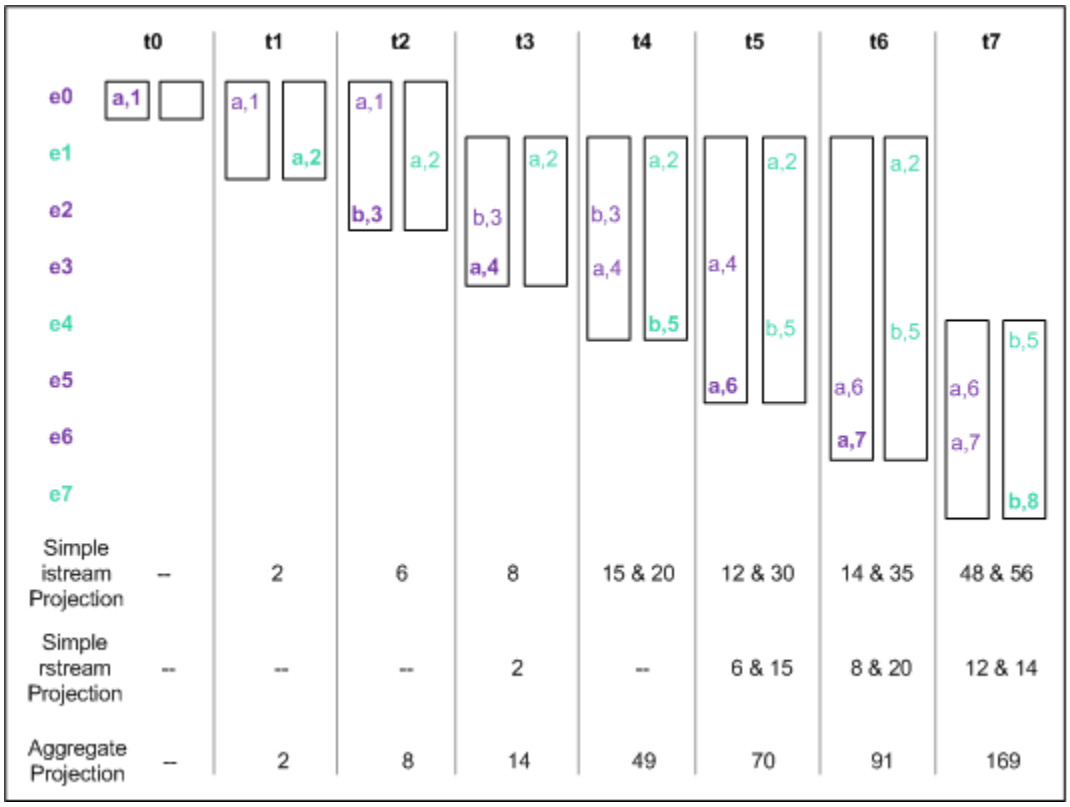
Defining stream queries

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 sample application (see the `samples/monitorscript/statarb` directory of your Apama installation directory), which includes the following statement:

```
stream <decimal> spreads :=
   from a in all Price(symbol=symbolA) retain 1
   from b in all Price(symbol=symbolB) retain 1
   select (a.price - b.price);
```

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.

Joining two streams

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]
```

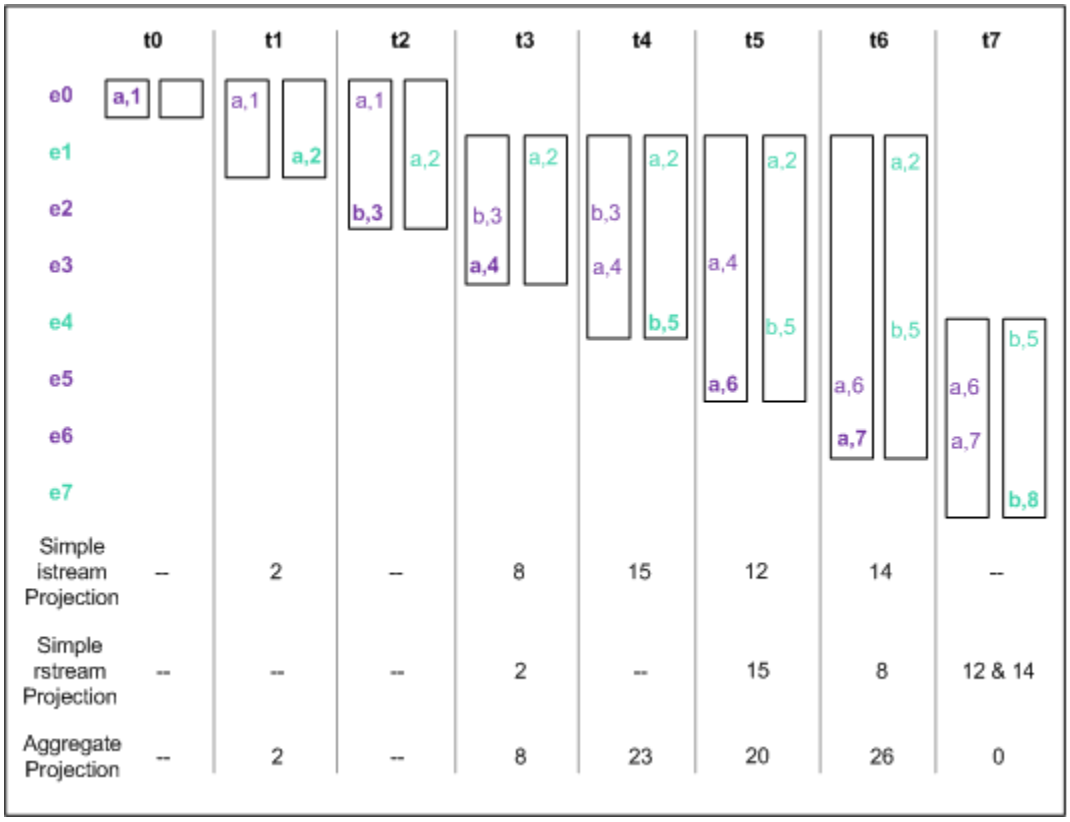
```
join itemIdentifier2 in streamExpr2 [windowDefinition2]
on joinKeyExpr1 equals joinKeyExpr2
```

As with the partition and unique key expressions, each join key expressions must return a comparable type. See the "Comparable types" in the "Types" section of the *Apama EPL Reference*.

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): alert{
      emit alert; }
```

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 emitted.

See also "IEEE special values in stream query expressions" on page 130.

Joining two streams

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.

Format

```
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.

Defining stream queries

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`*

Each *`groupByExpr`* 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 aggregate functions in the `having` or `select` expression operate over each group separately. See "Grouping output items" on page 128.

The *`havingExpr`* expression filters output items. See "Filtering items in projections" on page 129 .

The value you specify for *`selectExpr`* defines the items that are the result of the query. The correlator evaluates *`selectExpr`* to generate each item that appears in the query's output stream. The type of *`selectExpr`* 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.	`select currentTime;`
Item expression	Simple	A reference to the input item or a non-aggregate expression that contains at least one reference to the input item.	`select a.i;` `select sqrt(a.x)*5.0/a.y`
Group key expression	Aggregate	An expression that returns one of the group keys can also occur in the projection.	`group by a.i/10 select (a.i/10)*mean(a.x);`
Aggregate function expression	Aggregate	An expression that contains at least one aggregate function. Arguments to the aggregate function can include item expressions.	`select mean(a.i);`

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.

Defining stream queries

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 131.

Descriptions of built-in aggregate functions

EPL provides the following built-in aggregate functions. In the table, the argument names, for example, *value* and *weight*, are placeholders for expressions. Additional information about some of these functions follows the table.

Aggregate Function	Argument(s)	Returns	Result Description
`avg(value)`	`decimal` or `float`	`decimal` or `float`	The arithmetic mean of the values in the window. The `avg()` and `mean()` functions do exactly the same thing. They are aliases for each other.
`count()`	-	`integer`	The number of items in the window, including any NaN items
`count(predicate)`	`boolean`	`integer`	The number of items for which the argument is true
`count(value)`	`decimal` or `float`	`integer`	The number of items where the `decimal` or `float` value is not NaN
`first(value)`	`decimal` or `float`	`decimal` or `float`	The earliest value in the window being aggregated over
`last(value)`	`decimal` or `float`	`decimal` or `float`	The latest value in the window being aggregated over
`max(value)`	`decimal` or `float`	`decimal` or `float`	The maximum value
`mean(value)`	`decimal` or `float`	`decimal` or `float`	The arithmetic mean of the values in the window. The `mean()` and `avg()` functions do exactly the same thing. They are aliases for each other
`min(value)`	`decimal` or `float`	`decimal` or `float`	The minimum value
`nth(value,index)`	`decimal`, `integer` or `float`, `integer`	`decimal` or `float`	The value of the specified `decimal` or `float` item in the *index* position, starting with the earliest item in the window (item 0) and moving toward the latest item. `nth(value,0)` returns the same item as `first(value)`.
`prior(value,index)` .	`decimal`, `integer` or `float`, `integer`	`decimal` or `float`	The value of the specified `decimal` or `float` item in the *index* position, starting with the most recent item in the window (item 0) and moving toward the earliest item.

Aggregate Function	Argument(s)	Returns	Result Description
			`prior(value,0)` returns the same item as `last(value)`.
`stddev(value)`	`decimal` or `float`	`decimal` or `float`	The standard deviation of the values
`stddev2(value)`	`decimal` or `float`	`decimal` or `float`	The sample standard deviation of the values
`sum(value)`	`decimal`, `float` or `integer`	`decimal`, `float` or `integer`	The sum of the values
`wavg(value,weight)`	`decimal`, `decimal` or `float`, `float`	`decimal` or `float`	The weighted average of the values where each value is weighted by the corresponding weight

Calculations by the built-in aggregate functions might be affected by underflow and overflow. For example, adding a very large number to the collection that the `sum()` function operates on, then adding a very small number, and then removing the very large number will probably result in `0.0`, and not the very small number. Just adding the very small number would result in behavior that you would expect. As with the rest of EPL, the overflow and underflow characteristics are as defined for IEEE 64-bit floating point numbers.

Overloaded functions

The `sum()` function is overloaded. You can specify a `decimal`, `float` or `integer`. The return type matches the argument type.

The `avg()`, `first()`, `last()`, `max()`, `mean()`, `min()`, `nth()`, `prior()`, `stddev()`, and `stddev2()` functions are overloaded. You can specify a `decimal` or a `float`. The return type matches the argument type.

The `count()` function is overloaded. You can specify a `decimal` or a `float`. The return type is an `integer`.

The `wavg()` function is overloaded. You can specify a `decimal, decimal` or a `float, float` combination. The return type will be a `decimal` or a `float`, respectively.

Enabling use of built-in aggregate functions

The built-in aggregate functions reside in the `com.apama.aggregates` package. 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.

Operating on empty windows

Except for the `sum()` and `count()` functions, if the window being aggregated over is empty or insufficiently large then the result is not-a-number (NaN). The `sum()` and `count()` functions return zero if the window is empty.

IEEE special values in aggregate functions

Several of the built-in aggregate functions take `decimal` or `float` arguments. It is possible for a `decimal` or `float` value to be one of the following:

- Positive infinity
- Negative infinity
- Not-a-number (NaN)
- A finite number

The four positional aggregates (`first()`, `last()`, `nth()` and `prior()`) are agnostic to the values in them and return the selected item regardless of its value. If the selected item does not exist (for example, selecting the fifth item from a window of three items), then the aggregate returns `NaN`. The index for nth() and prior() must not be negative. If it is, the correlator terminates the monitor instance.

All the remaining (arithmetic) aggregate functions that take `float` or `decimal` arguments ignore any NaN items that are in the window being aggregated. The result is the aggregate of the window without the NaN items. If you want to count all items including NaN items then use the `count()` aggregate function that takes no arguments.

The behavior of arithmetic aggregate functions over windows that contain positive and negative infinities varies depending on the particular function. The result is either an infinity, NaN or a finite value. The table below shows for a window containing one or more positive infinities and no negative infinities, one or more negative infinities and no positive infinities, or at least one positive and at least one negative infinity, which aggregate function gives which result. In the case of the `wavg()` function the result depends on whether the infinity is the value or the weight.

Input	Outputs +	Outputs -	Outputs NaN	Outputs Finite Value

+Inf	`max()`
	`mean()`
	`sum()`
	`wavg(value)`
			`stddev()`
			`wavg(weight)`
				`min()`

-Inf		`mean()`
		`min()`
		`sum()`
		`wavg(value)`
			`stddev()`
			`wavg(weight)`
				`max()`

Both	`max()`	`min()`	`mean()`
			`stddev()`
			`sum()`
			`wavg()`	

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" in the "Types" section of *Apama EPL Reference*.

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

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 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 rstream 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 110, "Defining size-based windows" on page 111, "Defining cross-joins with two from clauses" on page 119 and "Defining equi-joins with the join clause" on page 120.

When you specify `retain all` you cannot specify `rstream`.

Generating query results

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.

Format

```
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 128.

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 ouput 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` exrpression 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)) : avg_sd {
        if(shouldOutput()) then {
            emit avg_sd;
       }
    }
```

This query computes a rolling average and standard deviation over the last ten minutes of a stock, and emits 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)) : avg_sd {
        emit avg_sd;
    }
```

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.

Generating query results

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 stream queries

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 124. 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 }
```

`bounded | unbounded`	Specify `bounded` when you are defining a custom aggregate function that will work with only a bounded window. That is, the 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, the 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. 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.
`aggregateName`	Specify a name for your aggregate function. This is the name you will specify when you call the function in a `select` clause. For details about the characters you can specify, see "Identifiers" in the "Lexical Elements" section of the *Apama EPL Reference*.
`arglist`	Optionally, specify one or more comma-separated data type/name pairs. Each pair indicates the type and the name of an argument that you are passing to the function. For example, `(float price, integer quantity)`.
`retType`	Specify any EPL data type. This is the type of the value that your function returns.
`aggregateBody`	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 `init()`, `add()`, `remove()` and `value()` actions are special. They define how stream queries interact with custom aggregate functions.

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 "Name precedence" in the "Lexical Elements" section of the *Apama EPL Reference*.

Working with Streams and Stream Queries

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) then {
         s0 := s0 + w;
         s1 := s1 + w*x;
         s2 := s2 + w*x*x;
      }
   }
   action remove( float x, float w ) {
      if (w != 0.0) then {
         s0 := s0 - w;
         s1 := s1 - w*x;
         s2 := s2 - w*x*x;
      }
   }
   action value() returns float {
      if (s0 != 0.0) then { return ((s2 - s1*s1/s0)/s0).sqrt(); }
      else { return float.NAN; }
   }
}
```

Defining custom aggregate functions

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.

Defining custom aggregate functions

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.

Defining custom aggregate functions

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(); }
}
```

Defining custom aggregate functions

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 135 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.

Working with Streams and Stream Queries

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.

Working with lots that contain multiple items

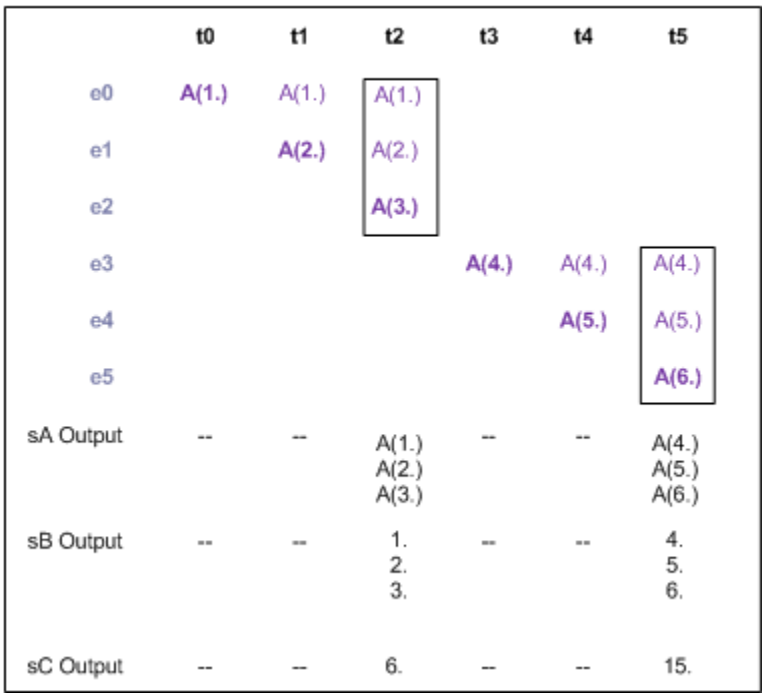
Behavior of stream queries with lots

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 table 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.

Working with lots that contain multiple items

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.

Behavior of stream queries with lots

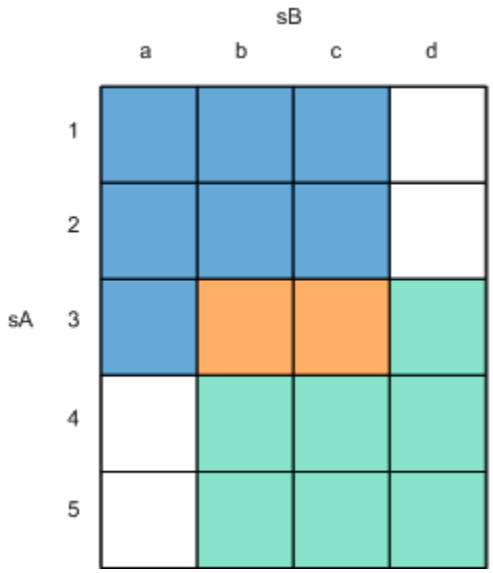
Join operations and lots

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.

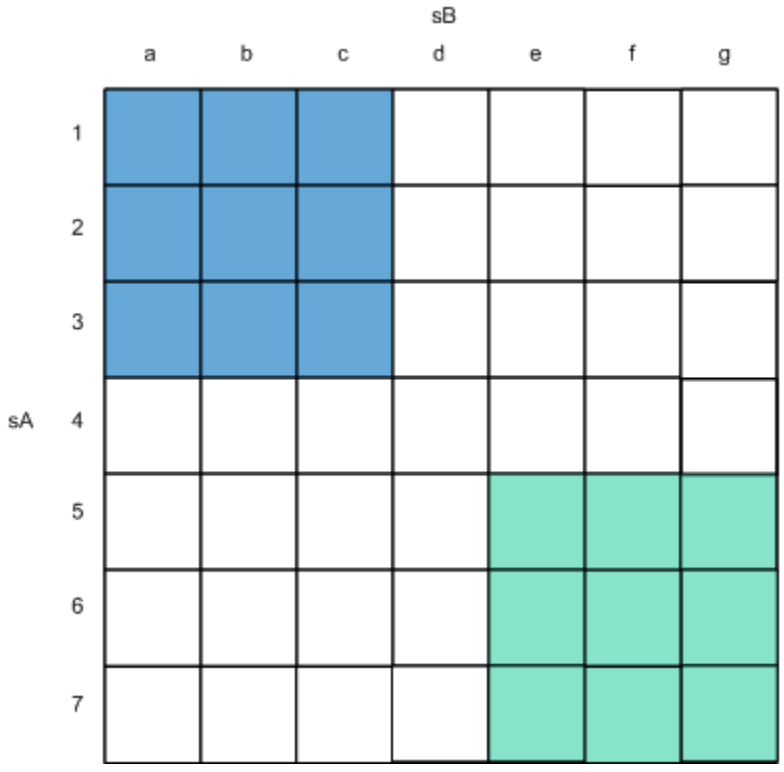
In the diagrams, the items on the left side of the join are represented by the numbered items that come in from the left side and the items on the right side of the join are represented by the lettered items that come in from the top. Each square in the grid can be a joined event. In both diagrams, the 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.



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).



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.

Behavior of stream queries with lots

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.

Behavior of stream queries with lots

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 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 307. Here is an example:

```
action createStreamListener returns listener {
   stream <A> sA := all A();
   return from a in all A() select a.x: x { print x.toString(); }
   // error: meant to use sA in the query above
}
```

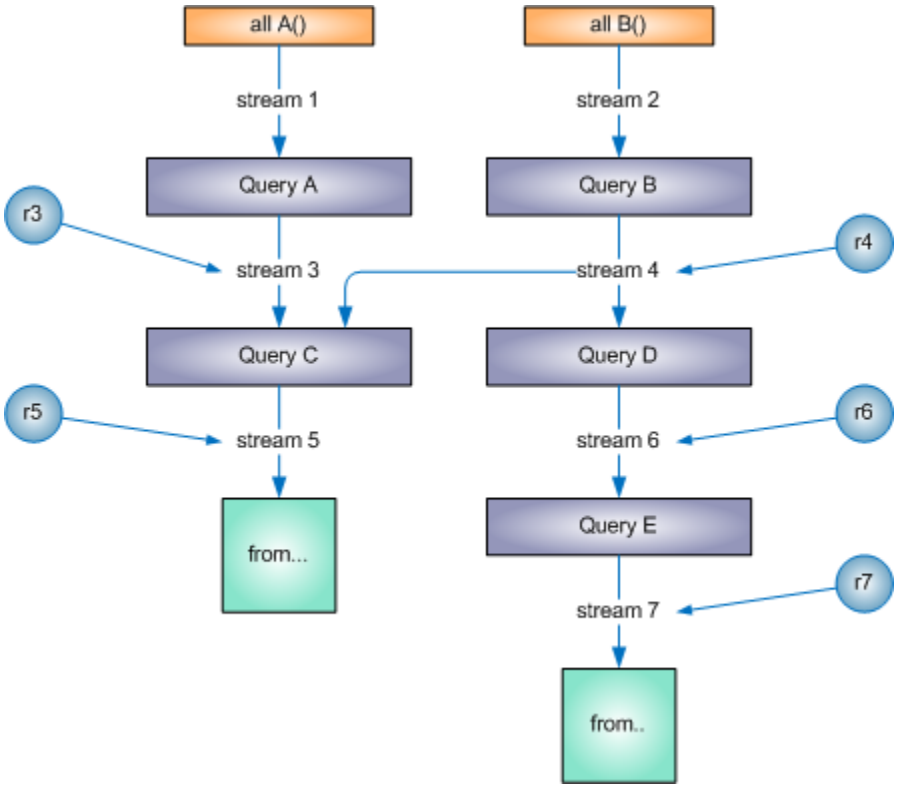
Although executing the code returns a listener variable that refers to the created stream listener, it inadvertently creates an unreferenced stream (the local variable `sA` did refer to this stream but is no longer in scope).

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. An example of how this works is probably beneficial. 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).

Working with Streams and Stream Queries

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.

Stream network lifetime

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.

Stream network lifetime

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.

Working with Streams and Stream Queries

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.

Using dynamic expressions in stream queries

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);
```

Using dynamic expressions in stream queries

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, enqueuing or emitting an event
- Calling another action that has side effects
- Setting up event listeners or new streams

Using dynamic expressions in stream queries

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 stream queries

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
`retain n`	The correlator evaluates `n` every time an item arrives on the stream. The correlator uses the new value of `n` to calculate what should be in the window.
`retain n every m`	The correlator stores incoming items until the current value of `m` is satisfied. When `m` is satisfied, the correlator evaluates both `n` and `m`. The correlator uses the new value of `n` to calculate what should be in the window, including the stored items. Because `m` is evaluated only after it has been satisfied, meeting that condition is always based on the old value of `m`.
`within d`	The correlator evaluates `d` 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 `d` to calculate what should be in the window.
`within d every p`	The correlator stores incoming items until `p` seconds have elapsed. When `p` seconds have elapsed, the correlator evaluates `p` and `d` only if there are any items in the window or stored. The correlator uses the new value of `d` to calculate what should be in the window, including stored events. The correlator uses the new value of `p` to determine the next time the window can change. If there are no items in the window or waiting to enter the window then, for efficiency, the correlator does not evaluate `p`. When the correlator evaluates `p` it is always based on the old value of `p`.
`...retain n`	If a `within` or `within every` window definition also specifies `retain`, the correlator evaluates `n` whenever the window content can change. The correlator uses the new value of `n` to calculate what should be in the window. If the window definition specifies `every`, the window content can change only when `p` is satisfied.

Window Definition	Description
	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.
`partition by k1[, k2]...`	If the window definition specifies a timed `every p` clause, the correlator evaluates each partition expression when `p` 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.
`with unique w`	The correlator evaluates `w` once for each item whenever that item is about to enter the window. If there is an `every` clause, an item can enter the window only when `m` or `p` is satisfied. Otherwise, an item can enter the window when it arrives on the stream.

Understanding when the correlator evaluates particular expressions

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.

Understanding when the correlator evaluates particular expressions

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.

Understanding when the correlator evaluates particular expressions

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.

Understanding when the correlator evaluates particular expressions

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.

Using dynamic expressions in stream queries

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 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() {
      VwapRequest v;
      on all VwapRequest():v spawn monitorVwap(v);
         // Simplified. Assumes no duplicate requests.
   }
   action monitorVwap(VwapRequest v) {
      params := v.params;
      Vwap vwap;
      from t in all Ticks(symbol=v.symbol)
         within params.duration
         every params.period
         select Vwap(t.symbol,wavg(t.price,t.volume)):vwap {
            route vwap;
         }
      VwapRequestUpdate u;
      on all VwapRequestUpdate(symbol=v.symbol) : 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.

Examples of using dynamic expressions in stream queries

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() {
      CountHighValueTicksRequest r;
      on all CountHighValueTicksRequest():r spawn
         monitorHighValueTicks (r);
         // Simplified. Assumes no duplicate requests.
   }
   action monitorHighValueTicks(CountHighValueTicksRequest r) {
      threshold := r.threshold;
      stream<Tick> filtered := from t in all Ticks(symbol=v.symbol)
                               where t.price*t.volume > threshold
                               select t;
      integer c;
      from t in filtered within 60.0 every 60.0 select count(): c {
         print "Count of high value trades in previous minute: " +
            c.toString();
      }
      on all CountHighValueTicksRequestUpdate(symbol=r.symbol) : 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(): c { ... }
```

The parentheses around the embedded query are optional.

Examples of using dynamic expressions in stream queries

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] );
```

Examples of using dynamic expressions in stream queries

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 run-time 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.hasKey(t.symbol) ) then {
      return basketVolume[t.symbol];
   } else {
      return 0.0;
   }
}
```

Examples of using dynamic expressions in stream queries

Troubleshooting and stream query coding guidelines

This section provides high-level guidelines for writing stream query applications that implement best practices.

It is organized as follows:

- "Prefer on statements to from statements" on page 148
- "Know when to spawn and when to partition" on page 148
- "Filter early to minimize resource usage" on page 148
- "Avoid duplication of stream source template expressions" on page 149
- "Avoid using large windows where possible" on page 149
- "In some cases prefer retain all to a timed window" on page 150
- "Prefer equi-joins to cross-joins" on page 150
- "Be aware that time-based windows can empty" on page 150
- "Be aware that fixed-size windows can overflow" on page 150
- "Beware of accidental stream leaks" on page 151
- "IEEE special values in stream query expressions" on page 130

For examples of common stream query coding patterns, see the whitepaper *Apama EPL Streams: A Short Tour* in the `doc\pdf\monitorscript` directory of your Apama installation directory.

Working with Streams and Stream Queries

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.

Troubleshooting and stream query coding guidelines

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.

Troubleshooting and stream query coding guidelines

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.

Troubleshooting and stream query coding guidelines

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:

```
float d;
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 : d {
      print "Difference from mean is " + d.toString();
   }
```

The first fragment behaves differently than this fragment:

```
float d;
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 : 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.

Troubleshooting and stream query coding guidelines

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 the next topic for details.

Troubleshooting and stream query coding guidelines

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 or 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.

Troubleshooting and stream query coding guidelines

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.

Troubleshooting and stream query coding guidelines

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.

Troubleshooting and stream query coding guidelines

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():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.

Troubleshooting and stream query coding guidelines

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.

Troubleshooting and stream query coding guidelines

5 Defining What Happens When Matching Events Are Found

- Using variables 154
- Defining actions 159
- Getting the current time 169
- Generating events 170
- Assigning values 174
- Defining conditional logic 174
- Defining loops 175
- Catching exceptions 176
- Logging and printing 178
- Sample financial application 182

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.

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 following types: `string`, `integer`, `float`, `boolean`, `sequence`, `dictionary`, `location`, `event`, `action`, `context`, `stream` or `listener`. For details about these types, see "Types" in the *Apama EPL Reference*.

Information about variables is presented in the following topics:

- "Using global variables" on page 154
- "Using local variables" on page 156
- "Using variables in listener actions" on page 157
- "Specifying named constant values" on page 158

See also, "Using action type variables" on page 163.

Defining What Happens When Matching Events Are Found

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. (Note that the correlator does not automatically initialize local variables.) The following table shows the values that the correlator assigns to uninitialized global variables.

Table 6. Values the correlator assigns to uninitialized global variables

Global Variable Type	Value Correlator Assigns to Uninitialized Global Variable
`action`	A null value that causes the monitor instance to die if you try to execute the action. In the correlator log file, the error message is `Called uninitialized action value.`
`boolean`	`false`
`chunk`	Contains no state. Each plug-in must define what to do upon receiving a default-initialized chunk as an argument.
`context`	A null 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`	`0.0d`
`dictionary`	Empty dictionary
`event`	Instance of the event where each of its fields has the standard default values as per this table.
`float`	`0.0`
`integer`	`0`
`listener`	A null 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)
`sequence`	Empty sequence
`stream`	A null stream that cannot be used in any meaningful way. To use the variable you must assign a non-null stream to it.
`string`	"" (empty string)

Using variables

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 intialize 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() {
   Event e;
   on all Event():e {
      log e.toString();
   }
}
```

You can also initialize a local variable by coassigning to it from a stream. For example:

```
action onload() {
   float f;
   from x in all X() select x.f : f {
      log f.toString();
   }
}
```

Using variables

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 (as described in "Reference data types" in the "Types" section of the *Apama EPL Reference*), 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.

Using variables

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;
```

type	Specify `boolean`, `decimal`, `float`, `integer`, or `string`. This is the type of the constant value.
name	Specify an identifier for the constant. This name must be unique within its scope — monitor, event, or action.
literal	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.

Using variables

Defining actions

A monitor can define any 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. 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 following topics provide information about defining actions:

- "Format for defining actions" on page 159
- "Invoking an action from another action" on page 160
- "Specifying actions in event definitions" on page 161
- "Using action type variables" on page 163

Defining What Happens When Matching Events Are Found

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 "Types" section of the *Apama EPL Reference*.

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 38. 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.

Before Apama Release 4.1, actions and variables were allowed to have the same names. For example, you were allowed to coassign an event to a variable that had the same name as the action that handled the event:

```
on all Update():update update();
```

With Apama 4.1, this is no longer allowed since you can now declare `action` type variables. See "Using action type variables" on page 163. If you have any code that uses the same identifier for an action and a variable, you must change it. For example:

```
on all Update():update handleUpdate();
```

Defining actions

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();
```

Defining actions

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. For example, an action in an event can

- Set up event or stream listeners
- Call other actions within that event
- Access members of that event

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 instance in which the call was made. 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 plugins from within event actions. In the event definition, specify the `import` statement to to give the plugin an alias within the event. Specify the `import` statement in the same way that you specify it for a monitor. You use the plugin alias to call functions on the plugin in the same way as you use it for a monitor.

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.

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 53.

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.
- The following event listener call syntax is not valid within event actions:

  ```
  on A() foo;
  ```

  Instead, specify this:

  ```
  on A() foo();
  ```

Defining actions

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, or enqueue 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 167.

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 peform 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 unitialized, 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 161.) You must include this implicit argument when determining whether an action definition conforms to an `action` variable declaration. For example, the following is illegal:

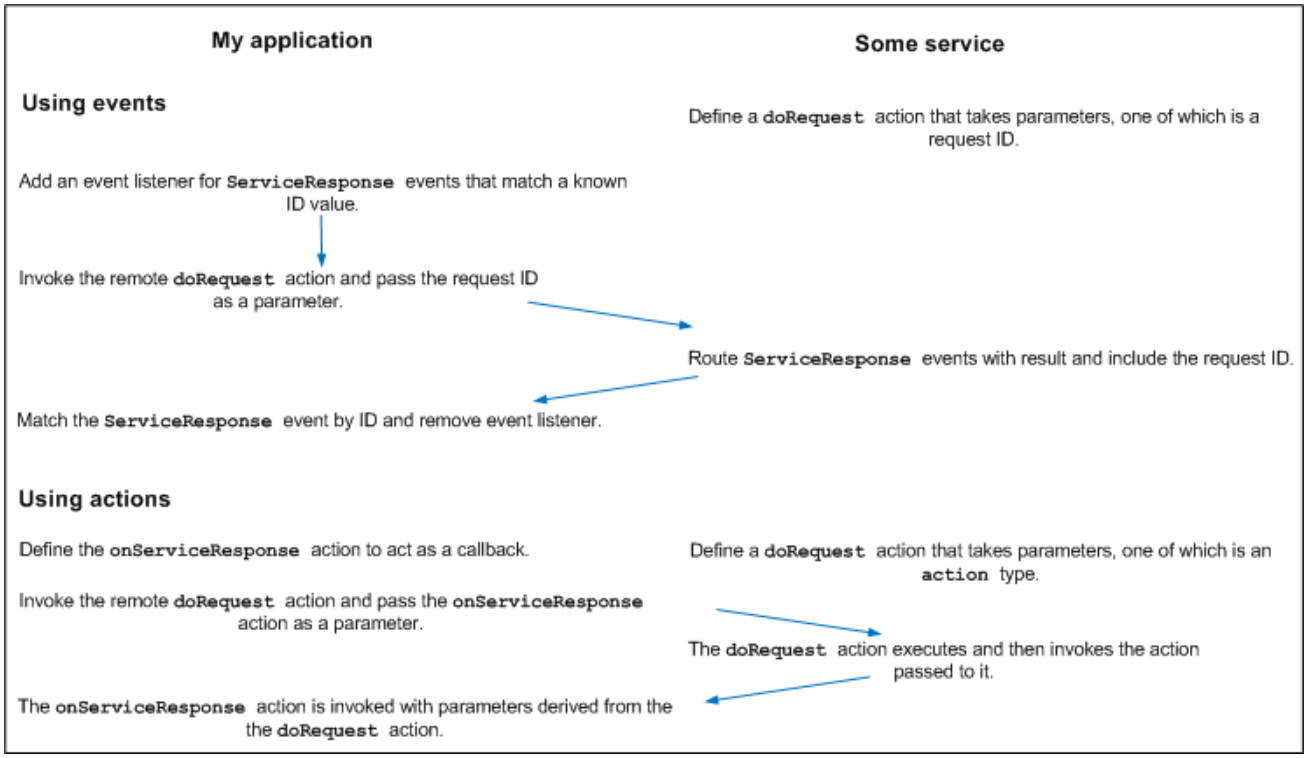
```
event A {
   action foo(float) returns string {
      return "Hello";
   }
   action bar() {
      action<float> returns string f := A.foo;
   }
}
```

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) 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 := ...;
      ServiceResponse r;
      on Response( requestId=id ): 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, or emit an event that contains an `action` variable field. It is okay to route, enqueue, or emit 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.

Java

In an Apama Java application, you cannot declare event types that have `action` type members. Consequently, events that contain `action` type fields are invisible to Apama Java applications.

Defining actions

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 "Generating events that keep time" on page 91.
- 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 88.

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.

Defining What Happens When Matching Events Are Found

Generating events

As discussed previously, listener actions can perform calculations and log messages. In addition, listener actions can dynamically generate events. Specify the `route`, `enqueue`, or `emit` statement to do this.

The following topics discuss this:

- "Generating events with the route command" on page 171
- "Generating events with the enqueue command" on page 171
- "Generating events to emit to outside receivers" on page 173

Defining What Happens When Matching Events Are Found

Generating events with the route command

The `route` command 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` command. 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" on page 35.

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.

You cannot route the following types:

- `action`, `chunk`, `listener`, `stream`
- A `sequence` that contains a type that is unroutable
- A `dictionary` whose key or value is a type that is unroutable
- An `event` that contains a type that is unroutable

Note that you can route an event whose type is defined in a monitor.

Generating events

Generating events with the enqueue command

The `enqueue` command generates an event and places the event on a special queue just for events generated by the `enqueue` command. A separate thread moves these events to the input queue of each public context. This arrangement ensures that if the input queue of a public context is full, the event generated by `enqueue` still arrives on its special queue, and is moved to each public context's input queue as soon as that queue has room. Active listeners will eventually receive events that are `enqueue`'d, once those events make their way to the head of the input queue alongside normal events.

There are two formats available for using `enqueue`. You can directly enqueue an event, as the example below does first, or else place the event in a string and enqueue that. If you use this latter format, you must ensure that you define the string to represent a valid event.

Use the `enqueue` statement when you want to ensure that the correlator processes the generated event after it has processed all routed events. Note that other external or enqueued events may be processed prior to processing this enqueued event. To defer processing an event until after

processing of all routed events, enqueueing to `context.current()` might be preferable. The `enqueue` statement is also useful when you want to send events into all public contexts.

For example, consider a further revised version of the earlier example:

```
event StockTickPriceChange {
   string owner;
   string name;
   float price;
}

// A new processTicks action that dispatches an event to
// the input queue instead of logging
action processTicks() {

// The following enqueue format sends the event itself.
   enqueue StockTickPriceChange(currentStock.owner,
      newTick.name, newTick.price);

// Or, use the following enqueue format, which sends a string that
// contains the event.
   enqueue "StockTickPriceChange(\""+currentStock.owner+
      "\",\""+newTick.name+"\", "+newTick.price.toString()+")";
}
```

If the string does not represent an event that fully complies with an event type that has been defined elsewhere in EPL then it will be thrown away before being placed on the input queue. This is the same behavior as for any normal event received by the correlator. Unless the correlator understands its event type (by having had it defined in EPL) it ignores it.

You cannot enqueue the following events:

- An event whose type is defined inside a monitor.
- An unroutable event type, that is, an event type that contains a field whose type is something other than a primitive type, a `location` type, or a `context` type.

Generating events

Enqueuing to contexts

To enqueue an event to a particular context, use the following form of the `enqueue` statement:

`enqueue` *`event_expression`* `to` *`context_expression`*`;`

The result type of *`event_expression`* must be `event`. It cannot be a string representation of an event.

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 "Enqueueing an event to a particular context" in *Developing Apama Applications in EPL*.

Enqueueing an event to a sequence of contexts is non-deterministic. For details, see "Enqueuing an event to a sequence of contexts" in *Developing Apama Applications in EPL*.

All routable event types can be enqueued to contexts, including event types defined in monitors.

Generating events

Generating events to emit to outside receivers

The `emit` command dispatches events to external registered event receivers, which means that the events leave the correlator. Active listeners do not receive emitted events.

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.

You cannot emit the following events:

- An event whose type is defined inside a monitor

- An unroutable event type

Generating events

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.

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.

Defining What Happens When Matching Events Are Found

Defining conditional logic

EPL supports conditional `if-then` and `if-then-else` statements.

Syntactically an `if-then` statement consists of an `if` keyword followed by a `boolean` expression followed by a `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. If the expression is false, the `if-then` statement exits.

The `boolean` expression must evaluate to the `boolean` values `true` or `false`.

An `if-then-else` consists of `if` followed by a `boolean` expression followed by `then` followed by a 'then' block followed by an `else` keyword followed by an 'else' block. If the `boolean` expression is true, the first block is executed, otherwise the second block is executed.

There is a special variant of the `if-then-else` allowed where a second nested `if-then` or `if-then-else` statement can replace the second block. This is only of relevance in that no curly braces are required in this special case.

In standard BNF notation this syntactic definition looks as follows:

```
ifStatement ::= if booleanExpression then block
| if booleanExpression then block1 else block2
| if booleanExpression then block3 else ifStatement block ::= {statementList }
```

Note: BNF is an acronym for "Backus Naur Form". John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language in 1960, and since then BNF notation is the standard notation used to specify the syntax rules of programming languages.

An EPL example follows:

```
if floatVariable > 5.0 then {
   integerVariable := 1;
}  else if floatVariable < -5.0 then {
      integerVariable := -1;
}  else {
      integerVariable := 0;
}
```

Note that `if-then-else` statements can be nested. In other words, the body of a `then` or an `else` can contain another `if-then-else`, in addition to the explicit `else if` combination.

Defining What Happens When Matching Events Are Found

Defining loops

EPL supports two loop structures, `while` and `for`.

The `while` statement's BNF definition is:

whileStatement ::= while *booleanExpression block*

An EPL example 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`. In BNF its definition is:

forStatement ::= for *counter* in *sequence block*

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.

Defining What Happens When Matching Events Are Found

Catching exceptions

EPL supports the try-catch exception handling structure. The try-catch statement's BNF definition is:

`tryCatchStatement` `::= try` *`block1`* `catch(Exception` *`variable`*`)` *`block2`*

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 enqueueing an event to a context and specifying a variable that has not been assigned a valid context object

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 data type. The `Exception` and `StackTraceElement` data 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 *EPL Reference*, *Types*, *Exception*.
- 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.getStackTace()`
- `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.

Defining What Happens When Matching Events Are Found

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 182.

The `log` statement sends the specified string to a particular log file depending on the applicable log level. For details, see *Deploying and Managing Apama Applications*, "Event Correlator Utilities Reference", "Shutting down and managing components", "Setting logging attributes for packages, monitors, and events".

The following topics provide information for using the `log` statement:

- "Specifying log statements" on page 179
- "Log levels determine results of log statements" on page 179

- "Where do log entries go?" on page 181
- "Examples of using log statements" on page 181
- "Strings in print and log statements" on page 182

Defining What Happens When Matching Events Are Found

Specifying log statements

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

```
log string [at identifier]
```

Syntax description

string	Specify an expression that evaluates to a string.
identifier	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 `CRIT`.

For each encountered `log` statement, the correlator compares the specified identifer 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 logging attributes for packages, monitors, and events" in the "Event Correlator Utilities Reference", section of *Deploying and Managing Apama Applications*.

Logging and printing

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 evaluated even
   // though the log level is too low to send output to the log file.

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 emit an event.
```

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 logging attributes for packages, monitors, and events" in *Deploying and Managing Apama Applications*, Event Correlator Utilities Reference, "Shutting down and managing components". 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

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
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.

Logging and printing

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`.

Logging and printing

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.

Logging and printing

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.

Logging and printing

Sample financial application

This section describes a complete financial example, using the techniques discussed earlier in this chapter.

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` command 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` command (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` command exploits data from variables used before and after the `spawn` command (that is, `limit` and `latestPrice`, respectively).

This example also demonstrates how mathematical operations may be used within event expressions. Here, two `on` commands 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.

Defining What Happens When Matching Events Are Found

6 Implementing Parallel Processing

- Introduction to contexts 186
- Creating contexts 188
- Calling context methods 189
- Obtaining context references 189
- Spawning to contexts 190
- Enqueuing an event to a particular context 191
- Enqueuing an event to a sequence of contexts 193
- Common use cases for contexts 194
- Samples for implementing contexts 194
- Contexts and correlator determinism 201
- How contexts affect other parts of your Apama application 201

By default, the correlator operates in a serial manner. If you want, you can implement contexts for parallel processing.

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.

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 252. 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, the type `context` is a reference type. When you create a value 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 enqueue an event to the context. When you spawn to a context, the correlator creates the context if it does not already exist.

The following topics introduce contexts:

- "What is inside/outside a context?" on page 186
- "About context properties" on page 187
- "Context lifecycle" on page 187
- "Comparison of a correlator and a context" on page 187

Implementing Parallel Processing

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.

Introduction to contexts

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. A value of `true` lets the context receive external events; this is a public context. A value of `false` indicates a private context that cannot receive external events. This is the default. Note that the main context is public.

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 enqueue 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 "Enqueuing an event to a particular context" on page 191.

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.

Introduction to contexts

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.

Introduction to contexts

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, however, a correlator has a few special properties:

- It is always public.
- The name is always `main`.
- It always exists.
- It is the context where `onload()` is run.
- It is the only context where Java applications can run.

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.

There is one queue of enqueued events for all contexts. When you specify the `enqueue` command (not the `enqueue...to` command), the enqueued event goes to the special queue for enqueued events. The correlator then places the event on the input queue of each public context. The correlator ensures that an enqueued event always arrives on the appropriate input queue(s). An enqueue operation never blocks. However, if the input queue of a context is full and the enqueued events queue gets very large, the result can be an unbounded memory usage error.

The `engine_receive` utility receives events from all contexts.

The `engine_send` utility sends events to all contexts that have been set to receive external events, that is, all public contexts.

Introduction to contexts

Creating contexts

In EPL, you refer to a context by means of an object of type `context`. The `context` type is a reference type. Context objects are lightweight and creating one only creates a stub object and allocates an ID. In other words, when you create a context, you are actually creating a context reference.

Use one of the following constructors to create a context:

```
context(string name)
context(string name, boolean receivesInput)
```

The optional `receivesInput` Boolean flag controls whether the context is public or private:

- A public context can receive external, enqueued, and routed events.
- A private context cannot receive events from external components sent to the correlator. A private context can receive enqueued events from other contexts, and enqueued events and routed events from monitor instances in that context. The default is that a context is private.

The following example creates a reference, `c`, to a private context whose name is `test`:

```
context c:=context("test");
```

The following topics provide additional information:

- "Calling context methods" on page 189
- "How many contexts can you create?" on page 189

Implementing Parallel Processing

Calling context methods

After you create a context, you can call the following methods on that context:

Table 7. Calling context methods

Method	Returns	Description
`object.getId()`	`integer`	Instance method that returns the ID of the context.
`object.getName()`	`string`	Instance method that returns the name of the context.
`object.isPublic()`	`boolean`	Instance method that returns true if the context is public.
`object.toString()`	`string`	Instance method that returns a string that contains the properties of the context. For example, for a public context whose name is `test`, the content of the returned string would be something like this: `context(1, "test", true)`
`context.current()`	`context`	Static method that returns a reference to the current context. The current context is the context that contains the monitor instance or event instance that is calling this method.

Implementing Parallel Processing

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.

Calling context methods

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 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 enqueued event, or a spawn operation.

For example:

```
Calculate calc;
on all Calculate():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
   enqueue CalculationResponse(id, value) to caller;
}
```

If a monitor instance that creates a context does not send a context reference outside itself, no other context can send events to that context, except by means of correlator 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 190.

- `enqueue EventExpression to ContextExpression;`

  See "Enqueuing an event to a particular context" on page 191.

Implementing Parallel Processing

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.

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 203.

Like the regular `spawn` 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 42.

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 enqueued events in the order in which they arrive. For example, suppose a monitor contains the following statements:

```
spawn action1() to ctx;
enqueue e1 to ctx;
spawn action2() to ctx;
enqueue e2 to ctx;
```

The `ctx` context processes this in the following order: `action1()`, `e1`, `action2()`, `e2`.

Implementing Parallel Processing

Enqueuing an event to a particular context

In a monitor, you can enqueue an event to a particular context, as described here, or you can enqueue an event to a `sequence` of contexts, described in the next topic. The format for enqueuing an event to a particular context is as follows:

`enqueue` *`EventExpression`* `to` *`ContextExpression`*`;`

- Replace *`EventExpression`* with any valid EPL expression that is of an `event` type. Unlike the general `enqueue` statement, you cannot specify a `string` representation of an event. For example, you cannot enqueue `&TIME` pseudo-ticks.
- Replace *`ContextExpression`* with any valid EPL expression that is of the `context` type. Typically, this is the name of a `context` variable.

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 enqueued event. The correlator does not need to finish executing the action that enqueued the event before it processes the enqueued event in the target context. The correlator might process the enqueued event before it finishes executing the action that sent the event. Or, the correlator might process the enqueued 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 enqueued event is also unpredictable. For example:

```
action analyse(string symbol) {
   context c:=context(symbol);
   spawn submon(symbol) to c;
   com.apama.marketdata.Tick tick;
   log "Listening for "+symbol;
   on all com.apama.marketdata.Tick(symbol=symbol):tick {
      enqueue tick to c;
   }
   on com.apama.marketdata.Finished() {
      enqueue com.apama.marketdata.Finished() to c;
   }
}action submon(string symbol) {
   ...
}
```

The `enqueue...to` statement does not place the event on the special enqueued events queue. Instead, it puts the event on the end of the target context's input queue. Consequently, it is possible for an `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 enqueue 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 enqueue an event to a context that does not contain any monitor instances, the correlator discards the event because there are no listeners for it.

In a parallel application, when you specify `enqueue` and you do not specify a context, the correlator sends the event to all public contexts. To enqueue an event to only the main context you can simply use the `enqueue` statement if you are certain there are never other public contexts. If there are or there might be one or more other public contexts, then you must enqueue the event to a context reference for the main context. For example, you can obtain a reference to the main context by calling `context.current()` in a monitor instance in the main context. You can then pass that reference by spawning from that monitor instance to a monitor instance in another context. The format for doing this is as follows:

`enqueue` *`eventExpression`* `to` *`context_reference_to_main_context`*`;`

In some situations, for example when you change a single-context application to use parallel processing, you might want to explicitly enqueue an event to only the context that contains the monitor instance that contains the `enqueue` statement. To enqueue an event to only this context specify:

`enqueue` *`eventExpression`* `to context.current()`

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;
   }
}
```

See also "Generating events with the enqueue command" on page 171.

Implementing Parallel Processing

Enqueuing an event to a sequence of contexts

In a monitor, you can enqueue an event to a sequence of contexts. The format for doing this is as follows:

```
enqueue EventExpression to ContextSequenceExpression;
```

- Replace *EventExpression* with any valid EPL expression that is of an `event` type. Unlike the general `enqueue` statement, you cannot specify a `string` representation of an event. For example, you cannot enqueue `&TIME` pseudo-ticks.
- Replace *ContextSequenceExpression* with any valid EPL expression that resolves to `sequence<context>`.

This 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. You cannot enqueue an event whose type is defined inside a monitor.

In each target context, the correlator can immediately process the enqueued event. The correlator does not need to finish executing the action that enqueued the event before it processes the enqueued events in the target contexts. The correlator might process an enqueued event before it finishes executing the action that sent the event. Or, the correlator might process an enqueued event some time after it completes executing the action that sent the event. The order is unpredictable.

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 ];

   com.apama.marketdata.Tick tick;
   log "Listening for "+symbol;
   on all com.apama.marketdata.Tick(symbol=symbol):tick {
      enqueue tick to ctxs;
   }
   on com.apama.marketdata.Finished() {
      enqueue 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 ];
```

```
   com.apama.marketdata.Tick tick;
   log "Listening for "+symbol;
   on all com.apama.marketdata.Tick(symbol=symbol):tick {
      enqueue tick to ctxs.values();
   }
   on com.apama.marketdata.Finished() {
      enqueue com.apama.marketdata.Finished() to ctxs.values();
   }
}
action submon(string symbol) {
...
}
```

The `enqueue...to` statement does not place the event on the special enqueued events queue. Instead, it puts the event on the end of the input queue of each target context. Consequently, it is possible for an `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 an `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 enqueued event in that sequence because there are no listeners for it.

If one of the contexts in the sequence has not been created before you enqueue an event to it then the correlator terminates the monitor instance.

In a parallel application, when you specify `enqueue` and you do not specify a context, the correlator sends the event to all public contexts, including the main context.

Consider the following two code fragments:

```
for c in mySequence {
   enqueue myEvent to c;
}

enqueue 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.

Implementing Parallel Processing

Common use cases for contexts

See "Tuning contexts" on page 309.

Implementing Parallel Processing

Samples for implementing contexts

Apama provides a number of applications that illustrate the use of contexts. These examples are in the `samples\monitorscript\contexts` directory and in the `samples\monitorscript\concurrency-theory` directory.

Information for using these examples is in the following topics:

- "Simple sample implementation of contexts" on page 195
- "Running samples of common concurrency problems" on page 195
- "About the samples of concurrency problems" on page 196
- "About the race sample" on page 197
- "About the deadlock sample" on page 198
- "About the compareswap sample" on page 199

Implementing Parallel Processing

Simple sample implementation of contexts

In your Apama installation directory, in the `samples\monitorscript\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 in Apama Studio to compare the implementations.

To run the applications, execute `run-sample.bat` on Windows or `run_sample.sh` on UNIX. 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 emitted 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.

Samples for implementing contexts

Running samples of common concurrency problems

Sample applications in the `samples\monitorscript\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.
   - On Windows, select Start > Programs > Software AG > Apama 5.1 > Apama Command Prompt .

- On UNIX, source the `apama_env` file to set environment variables.

2. Change to the `$APAMA_INSTALL_DIR/samples/monitorscript/concurrency-theory` directory.
3. Invoke `run_sample.bat` (Windows) or `run_sample.sh` (UNIX) with an argument of `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`.

Samples for implementing contexts

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 emit 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.

Samples for implementing contexts

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). Teller1 based its calculation on values that were out of date at the point they were sent to the database.

Samples for implementing contexts

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, teller2 would wait to lock B's account until teller1 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 teller1 locks account A and teller2 locks account B, and teller1 tries to lock account B and teller2 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)`)	

Time	Teller 1	Teller 2	Bank Database
	Lock B		A: locked by t1, t2 waitingB: locked by t2, t1 waiting
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.

Samples for implementing contexts

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 retrys 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

Time	Teller 1	Teller 2	Bank database
	Get A		
	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

Time	Teller 1	Teller 2	Bank database
	Get B		
	B = 75		
	newB = 125		
	Set B (old=75)		
			A: 50, B: 125, C: 125
			Set B success

Samples for implementing contexts

Contexts and correlator determinism

Creating one or more contexts makes the correlator non-deterministic. In other words, injecting the same monitor can product 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 emit events
- The order in which contexts enqueue events to other contexts
- The order in which contexts spawn to other contexts

See also "About replay logs and parallel processing" on page 202 in *Deploying and Managing Apama Applications*.

Implementing Parallel Processing

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 following topics provide information to help you understand the behavior.

- "About replay logs and parallel processing" on page 202
- "Deadlock avoidance when parallel processing" on page 203
- "Clock ticks when parallel processing" on page 203

- "Using correlator plug-ins in parallel processing applications" on page 203

Implementing Parallel Processing

About replay logs and parallel processing

As mentioned before, applications that implement parallel processing might have non-deterministic behavior. However, you can still inject a parallel application into a correlator that you started with the `--replayLog` option. In this situation, the correlator records special pseudo-events that capture non-deterministic behavior in the replay log. This is in addition to recording all external events as usual.

To replay correlator behavior, including all non-deterministic decisions, specify the `--replayMode` argument when you start the correlator that will be reproducing the behavior captured in the replay log. Specify `--replayMode` in addition to the other options described in the correlator utilities section of the deployment guide.

See "Reprocucing correlator behavior from a replay log" in the correlator utilities section of *Deploying and Managing Apama Applications*.

There are some limitations on how you can modify the EPL that you are injecting when reproducing parallel processing behavior from a replay log. In the code that you replay, the following must all be the same as they were when the behavior was captured in the replay log:

- `emit`, `enqueue...to`, and `spawn...to` operations
- Calls to `context` constructors
- Calls to the `integer.getUnique()` method

For applications that use multiple contexts or that enqueue 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 `emit`, `enqueue...to` or other operation is important. Operations that can affect the state of other contexts or the emitted events are non-deterministic when run in parallel.

If a replay log is incomplete, it might not have sufficient information to process all events sent to it. For example, the non-deterministic behaviour that occurred when generating the replay log could be unknown. Incomplete replay logs can occur in the unlikely event that the correlator stopped while processing events, for example if it ran out of memory, was killed, or crashed. There is no simple way of determining if the replay log is incomplete other than attempting to run it and seeing if the correlator does process all events on the contexts' input queues.

It is possible to run the replay log extract script with the `--noReplayMode` option and then run the correlator with the `--replayMode` option. Running the extract script with the `--noReplayMode` option does not extract the extra information needed to reproduce non-deterministic behavior in the original order. The correlator would then process all events, but the behavior (in terms of what events are emitted and what state monitors hold) might be significantly different if your EPL makes use of multiple contexts.

How contexts affect other parts of your Apama application

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 enqueuing an event to context `2`.
- Context `2` is enqueuing 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 enqueuing 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`.

How contexts affect other parts of your Apama application

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 distributer 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.

How contexts affect other parts of your Apama application

Using correlator plug-ins in parallel processing applications

The standard MemoryStore, Time Format, Log File Manager, and Payload Extraction 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.

For information about writing correlator plug-ins for use with parallel applications, see "The Plug-in Development Kit" in *Writing Correlator Plug-ins*.

Note: The C class `AP_Context`, and the C++ class `Context`, which you use for correlator plug-in development, are completely different and unrelated to contexts that you define for parallel processing.

How contexts affect other parts of your Apama application

7 Using Correlator Persistence

- Description of state that can be persistent 206
- When persistence is useful 206
- When non-persistent monitors are useful 207
- How the correlator persists state 207
- Enabling correlator persistence 208
- How the correlator recovers state 210
- Designing applications for persistence-enabled correlators 213
- Upgrading monitors in a persistence-enabled correlator 213
- Sample code for persistence applications 214
- Requesting snapshots 217
- Developing persistence applications 217
- Using correlator plug-ins when persistence is enabled 217
- Using the MemoryStore when persistence is enabled 218
- Comparison of correlator persistence with other persistence mechanisms 219
- Restrictions on correlator persistence 220

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 Java monitors cannot be persistent. State in chunks, with a few exceptions, also cannot be persistent.

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 Java monitors. EPL files that were injected from a Correlator Deployment Package are not stored in plain text.

  Code that is not injected includes the following:

  - Correlator 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()` IDs and context IDs.

Note: In general, chunks cannot be persistent. However, chunks used by the Apama Time Format correlator plug-in and the Apama MemoryStore plug-in can be persistent.

Using Correlator Persistence

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.

Using Correlator Persistence

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 213.
- Interacts with user-defined correlator plug-ins or Apama correlator 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 Java monitors are non-persistent monitors.

Using Correlator Persistence

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, when you enable persistence the correlator does the following:

- Takes a snapshot of state changes every 200 milliseconds. This is the snapshot initerval. 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 replay/input log if one was specified when the correlator was started. This happens only upon restarting the correlator. The replay/input log can then use this initial state, plus its own data to replay correlator execution.
- 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.

Using Correlator Persistence

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 213.

To enable correlator persistence, you must

- 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 213, "Defining recovery actions" on page 212 and "Sample code for persistence applications" on page 214.
- Specify one or more persistence options when you start the correlator. You must always specify the `-P` option to enable correlator persistence.

  Specify only the `-P` option to implement default behavior for correlator persistence. To change 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`, `-PsnapshotInterval` and `-PstoreLocation`, the correlator uses the values you specify for the snapshot interval and the recovery datastore location and uses the default setttings for all other persistence behavior.

  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.

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	`-PsnapshotInterval=`*`interval`* Specify an integer that indicates the number of milliseconds to wait.
The correlator can automatically adjust the snapshot interval according to application behavior. It can be useful to set this to false to diagnose a problem or test a new feature.	True — the correlator automatically adjusts the snapshot interval.	`-PadjustSnapshot=`*`boolean`*
The correlator puts the recovery datastore in a specified directory.	The directory in which the correlator was started. That is, the current directory.	`-PstoreLocation=`*`path`* You can specify an absolute or relative path. The directory must exist.

Correlator Persistence Behavior	Default	Option for Changing
The correlator copies the snapshot into a specified file. This is the recovery datastore.	`persistence.db`	`-PstoreName=`*`filename`* Specify a filename without a path.
For correlators that use an external clock, the correlator uses a specified time of day as its starting time when it restarts. This behavior is useful only for replaying replay logs that contain recovery information.	The time of day captured in the last committed snapshot.	`-XrecoveryTime` *`num`* To change the default, specify an integer that indicates seconds since the epoch.
The correlator can automatically copy the recovery datastore to the replay log when a persistence-enabled correlator restarts.	The correlator copies the recovery datastore to the replay/input log.	`-noDatabaseInReplayLog` 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 replay log.

Using Correlator Persistence

How the correlator recovers state

When you restart a correlator for which persistence has been enabled the correlator

- Detects, recompiles, and reinjects 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 Java monitors at their `onload()` action
- Executes any `onBeginRecovery()` and `onConcludeRecovery()` actions. See "Defining recovery actions" on page 212 .
- Recovers persistent connections (connections created with `engine_connect -p`) and resumes them at the first opportunity

Code is reinjected 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 explicity with the `engine_delete` utility or implicity 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.

Using Correlator Persistence

Recovery order

When the correlator recovers state from a recovery datastore it does the following in the following order:

1. Recompile and reinject all source 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 212.
5. Quiesce — The correlator waits for all events that have been enqueued to a context to be processed, and also waits for any events that are enqueued to a context as a result of those events to be processsed, and so on, until no more events are generated and enqueued 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 `enqueue...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 212.
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`.

How the correlator recovers state

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 reinjects 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 213 and "Sample code for persistence applications" on page 214.

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 211 for more details about when `onBeginRecovery()` and `onConcludeRecovery()` are executed.

How the correlator recovers state

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 Java and 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.

How the correlator recovers state

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.

Using Correlator Persistence

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. To upgrade a monitor without first terminating all old instances of the monitor:

1. Initially deploy a monitor that contains code that enables that monitor to give its state to a new version of the monitor and to terminate upon request. If a deployed monitor does not contain such code it is not possible to upgrade it without terminating all instances.
2. Modify your monitor code to the new behavior you want and be sure to change the name of the monitor. For example, if the old monitor is `RequestLoan`, you might name the new monitor `RequestLoan2`.
3. Add code to your upgraded monitor so it atomically routes events that do the following:
   a. Retrieves the current state of the old monitor.
   b. Checks that the new monitor can upgrade from the old monitor.
   c. Requests the old monitor version(s) to terminate.
   d. Sets up its own listeners.
4. Inject the new version of your monitor.

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.

Using Correlator Persistence

Sample code for persistence applications

Several topics provide sample code for persistence applications.

- "Sample code for discarding stale state during recovery" on page 215

- "Sample code for recovery behavior based on downtime duration" on page 215
- "Sample code that recovers subscription to non-persistent monitor" on page 216

Using Correlator Persistence

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();
      ControlEvent c;
      on all ControlEvent():c { handleControl(c); }
   }
   action initiateListeners;
      Data d;
      l:=on all Data():d { process(d); } // Process is moderately expensive
      lt:=on all wait(0.1) { emit Average(state); }
   }
   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 persistence applications

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
   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)
```

```
         then {
            // 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 then {
         longDowntime:=false;
         initiateListener(); // Go back to normal.
      }
   }
}
```

Sample code for persistence applications

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 {
   action onload {
      Subscribe s;
      on all Subscribe():s {
         if not connected then {
            pendingSubscribes.append(s);
         } else {
            if(incrRefCount(s.subkey) then {
               emit Adapter_Subscribe(s.subkey);
            }
         }
      }
      on all wait(1.0) {
         emit IsAdapterUp();
      }
      on all AdapterUp() {
         connected:=true;
         for s in pendingSubsscribes {
            route s;
         }
         pendingSubscribes.clear();
      }
   }
}
persistent monitor eg2 {
   listener l;
   Instance i;
   context svcCtx;
   action spawnedInstance(context c) {
      svcCtx:=c; // Contains anything required to recover subscription.
      enqueue Subscribe(i.subkey) to svcCtx;
      Data d;
      l:=on all Data():d { process(d); }
   }
   action onConcludeRecovery() {
        // Non-persistent service monitor is now reset to its onload state.
        // Re-subscribe.
      enqueue Subscribe(i.subkey) to svcCtx;
   }
}
```

Sample code for persistence applications

Requesting snapshots

A persistent or non-persistent monitor can request a snapshot to occur as soon as possible, and be notified of when that snapshot has been committed to disk. You use Apama's Management interface to do this. The Management interface lets you create instances of `Persistence` events and then call the `persist()` action on those events. When the correlator processes a `Persistence` event it takes and commits a snapshot and executes the specified callback action after the snapshot is committed.

To use the Management interface, you add the `Correlator Management` bundle to your Apama project. For details, see "Using the Management interface" on page 283.

Using Correlator Persistence

Developing persistence applications

While you are writing the EPL code for your persistence application, use Apama Studio 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 checkboxes so that nothing is reinjected.
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 checkboxes 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.

Using Correlator Persistence

Using correlator plug-ins when persistence is enabled

A persistent monitor can import a correlator plug-in only when one of the following conditions is met:

- None of the plug-in's functions/actions, including unused functions/actions, refer to a `chunk` data 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 correlator plug-ins and other Apama-provided plug-ins cannot persist chunks.

Using Correlator Persistence

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 a correlator-persistent store from a correlator that does not have persistence enabled the correlator terminates the monitor instance that is trying to prepare the store. Likewise, if you try to prepare an in-memory or on-disk 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		

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 corelator-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 283.

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 252.

Using Correlator Persistence

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

Persistence characteristic	Correlator persistence	MemoryStore	Apama Database Connector Adapter (ADBC)
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

Using Correlator Persistence

Restrictions on correlator persistence

Java monitors cannot be persistent.

A persistent monitor can use the Apama Time Format and MemoryStore correlator plug-ins and the `chunk` data types contained by the events defined by those plug-ins. A persistent monitor cannot use any other `chunk` data types. This means that a persistent monitor cannot use an event or plug-in that references a `chunk` data type even if the application does not use those chunks.

Using Correlator Persistence

8 Common EPL Patterns

- Contrasting using a dictionary with spawning 222
- Factory pattern 223
- Using quit() to terminate event listeners 224
- Combining the dictionary and factory patterns 225
- Testing uniqueness 225
- Reference counting 226
- Inline request-response pattern 228
- Writing echo monitors for debugging 229

When developing EPL 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.

Common EPL Patterns

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() {
      Translation t;
      on all Translation():t addTranslation(t);
      Input i;
      on all Input():i translate(i);
   }
   action addTranslation(Translation t) {
      translations[t.raw] := t.converted ;
   }
   action translate(Input i) {
      if translations.hasKey(i.value) then {
         emit Output( translations[i.value] );
      }
      else { fail(i); }
   }
   action fail(Input i ) {
      print "Cannot translate: " + i.value;
   }
}
```

Contrasting using a dictionary with spawning

Translation using spawning

Same events as translation using dictionary.

The monitor:

```
monitor Translator {
   action onload() {
      Translation t;
      on all Translation():t addTranslation(t);
      Input i;
      on all unmatched Input():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) {
      emit Output(converted);
   }
   action fail(Input i) {
      print "Cannot translate: " + i.value;
   }
}
```

Contrasting using a dictionary with spawning

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 222, to use the alternate factory form.

Common EPL Patterns

Canonical factory pattern

The event:

```
event NewOrder {...}
```

The monitor:

```
monitor OrderProcessor {
   ...
   action onload() {
      NewOrder order;
```

```
      on all NewOrder():order spawn processNewOrder(order);
   }
   action processNewOrder(NewOrder order) {
   ...
   }
}
```

Factory pattern

Alternate factory pattern

The event:

```
event NewOrder {...}
```

The monitor:

```
monitor OrderProcessor {
   NewOrder order;
   action onload() {
      on all NewOrder():order spawn processOrder();
   }
   action processOrder() {
   ...
   }
}
```

Factory pattern

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;
   OrderIn theOrder;
   action onload() {
      on all OrderIn():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.getUnique();
      Trade t; r.listener := on all Trade(orderOutId=id):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 then {
         r.listener.quit();
         ...
      }
      ...
   }
}
```

As stated earlier, for real-world solutions there is generally a better option that using `quit()`. For example, the exchange(s) probably also emit `OrderComplete` events. In this case you can change the `on` statement as follows:

```
on all Trade(orderOutId=id):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.

Common EPL Patterns

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 ensure 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.

Common EPL Patterns

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() {
      NewOrder order;
      on all NewOrder():order validateOrder(order);
   }
   action validateOrder(NewOrder order){
      if orders.hasKey(order.key) then{
         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() {
      ValidNewOrder valid;
      on all ValidNewOrder():valid spawn processOrder(valid.order);
   }
   action processOrder( NewOrder order ) {
   ...
   }
}
```

Common EPL Patterns

Reference counting

The following pattern is another example that you can use to 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() {
      Subscribe s;
      Unsubscribe u;
      on all Subscribe():s subscribe(s);
      on all Unsubscribe():u unsubscribe(u);
   }

   action subscribe(Subscribe s){
      if items.hasKey(s.toWhat) then {
         dictionary < string, integer > subscriptions :=
            items[s.toWhat];
         if subscriptions.hasKey(s.originator) then {
            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.hasKey(u.fromWhat) then {
         dictionary < string, integer > subscriptions :=
            items[u.fromWhat];
         if subscriptions.hasKey(u.originator) then {
            if subscriptions[u.originator] <= 1 then {
               subscriptions.remove(u.originator);
               if subscriptions.size() = 0 then {
                  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() {
      CreateServiceItem c;
      on all CreateServiceItem():c spawn createServiceItem(c);
   }

   action createServiceItem(CreateServiceItem c) {
   //...
      DestroyServiceItem d;
      on all DestroyServiceItem():d destroyServiceItem(d);
   }
```

```
   action destroyServiceItem(DestroyServiceItem d) {
   //...die;
   }
}
```

Common EPL Patterns

Inline request-response pattern

You can use the `route` command 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.

Common EPL Patterns

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, ... );
      Response r;
      on Response(requestId=id):r {
      // continue processing
      ...
      // Beware! Any code here will execute immediately
      // (before processing the response)
   }
}

monitor Server {
   action processRequests() {
      Request r;
      on all Request():r {
         // evaluate response
         route Response(r.id,...);
      }
   }
}
```

Inline request-response pattern

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,...);
      ResponseA ra;
      on ResponseA(id=requestId):ra doWork2(ra);
   }
   action doWork2(ResponseA ra) {
      //do some more processing
      integer requestId := integer.getUnique();
      route RequestB(requestId,...);
      Response rb;
      on ResponseB(id=requestId):rb doWork3(rb);
   }
   action doWork3(ResponseB rb) {
      //do yet more processing
   }
}
```

Inline request-response pattern

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():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():subDepth {
   doSomethingElse();
}
```

The `completed` event listener in the echo monitor triggers after the correlator finishes processing the `unmatched` event listener in the production monitor.

Common EPL Patterns

9 Using Correlator Plug-ins in EPL

- Overhead of using plug-ins 232
- When to use plug-ins 232
- When not to use plug-ins 232
- Using the Payload Extraction plug-in 233
- Using the Time Format plug-in 238
- Using the Log File Manager plug-in 248
- Using the MemoryStore 252
- Using the Management interface 283
- Interfacing with user-defined correlator plug-ins 285
- About the chunk data type 286

In EPL programs, you can use standard correlator plug-ins provided with Apama and you can also use correlator plug-ins that you define yourself. A correlator plug-in consists of an appropriately formatted library of C or C++ functions that can be called from within EPL. The event 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 correlator plug-ins, see *Writing Correlator Plug-ins*.

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 correlator 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 and use the Apama client SDK to write a separate process that does the computations. For example, the correlator might communicate with this external process by emitting `ComputeRequest` events on a particular channel and the process would respond by sending `ComputeResult` events.

Using Correlator Plug-ins in EPL

When to use plug-ins

A custom plug-in is a suitable solution in the following situations:

- You have an in-house or third-party library of (possibly complex) C/C++ functions that you want to re-use.
- The operations you need to perform are more easily/efficiently performed using the C/C++ language than using EPL. For example, you need to use data structures that are not easily represented in EPL.

Using Correlator Plug-ins in EPL

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 Correlator Plug-ins in EPL

Using the Payload Extraction plug-in

Apama events are rigidly structured and need to comply with an event type definition. This describes the structure of a particular event, in particular its name, as well as the order, name, type and number of its constituent fields.

However, external events, even when they are of the same 'type' or nature (for example, all `Trade` events or `News` headlines) might vary in format and structure, even when originating from the same source or feed.

This issue is of particular relevance in the IAF and its semantic mapper module, where rules define how to translate and map from external events to Apama events and vice-versa.

In order to accommodate this, Apama provides for the concept of a payload field within an Apama event. The payload field, typically the last field in an event type definition, can embed any number of additional optional fields in addition to the always-present primary fields.

For example, consider an external message that can appear in several guises, but where each always consists of a particular subset of critical fields together with a variable number of additional optional fields.

If it is desired that these varying guises are mapped to a single Apama event type, then this needs to be defined so that its fields correspond to the subset of critical (and always present) fields, followed by a payload field into which the additional (and optional) fields are embedded.

Note: The Payload Extraction correlator plug-in is deprecated as of Apama 5.0. All Apama adapter products now use dictionary-format payloads rather than string-format payloads. The dictionary-format payload is considerably more efficient in almost all cases. While string-format payloads are still accepted in Apama 5.1, this support, including the Payload Extraction correlator plug-in, is deprecated and will be removed in a future release of the Apama platform. It is recommended that all third-party adapters are migrated to dictionary-format payloads as soon as possible.

Using Correlator Plug-ins in EPL

Accessing the payload in the correlator

The contents of the payload field are structured as a list of field name and value pairs. Although this may be directly accessed as a string within EPL, Apama provides the Payload Extraction plug-in to make extraction and manipulation of specific fields more straightforward.

The Payload Extraction plug-in is available as `libPayloadPlugin.so` in the `lib` directory on UNIX. On Microsoft Windows it is available as `PayloadPlugin.dll` in the `bin` directory.

The following EPL, which should be located within the monitor that requires the plug-in's functions, loads the Payload Extraction plug-in and assigns it the alias `payloadMgr`:

```
import "PayloadPlugin" as payloadMgr;
```

The plug-in provides a suite of methods that allow extraction of named fields from the payload field.

There are two groups of these functions, the direct access functions, and the indirect access functions.

Using the Payload Extraction plug-in

The direct-access functions

The primary direct-access functions are described in the following sections. For information about their parameters, see "Parameters for primary direct-access functions" on page 234.

parseString

```
string parseString(
   string payload_string,
   string field_name,
   string default_value)
```

Access the payload specified in *payload_string*, locate the field called *field_name*, and return its contents as a `string`. If the field does not exist, is empty, or is invalid, return the string literal specified in *default_value*.

parseFloat

```
float parseFloat(
   string payload_string,
   string field_name,
   float default_value)
```

Access the payload specified in *payload_string*, locate the field called *field_name*, and return its contents as a `float`. If the field does not exist, is empty, is invalid, or cannot be represented as a `float`, return the float value specified in *default_value*.

parseInteger

```
integer parseInteger(
   string payload_string,
   string field_name,
   integer default_value)
```

Access the payload specified in *payload_string*, locate the field called *field_name*, and return its contents as an `integer`. If the field does not exist, is empty, is invalid, or cannot be represented as an `integer`, return the integer value specified in *default_value*>.

parseBoolean

```
boolean parseBoolean(
   string payload_string,
   string field_name,
   boolean default_value)
```

Access the payload specified in *payload_string*>, locate the field called *field_name*, and return its contents as a `boolean`. If the field does not exist, is empty, is invalid, or cannot be represented as a `boolean`, return the boolean value specified in *default_value*.

Using the Payload Extraction plug-in

Parameters for primary direct-access functions

The following table describes the parameters used by the direct-access functions:

Parameter	Description
payload_field	The payload field. For example, if the event to be analyzed has been placed in the event variable `ev`, then its payload field would conventionally be `ev.__payload`.
field_name	The name of the field to extract from the payload. This parameter takes a string literal.
default_value	The value to return if the field specified in *field_name* does not exist, has no value (is empty or null), is invalid (e.g. contains binary characters) or cannot be cast to the type requested. The type of this parameter depends on the particular method; it is a `string` for `parseString`, a `float` for `parseFloat`, an `integer` for `parseInteger`, and a `boolean` for `parseBoolean`. Note that all types can be cast to a `string`, however, the opposite might not be possible. For example a `string` can be only be successfully cast to a `float` if it contains a valid float number, for example, 1.0.

The direct-access functions

Secondary direct-access functions

In addition to the primary direct-access functions, the plug-in also provides a secondary set of direct access functions. These differ from the primary set in that instead of taking one default value that is returned in case of each of the three error conditions, they take a specific default value for each condition. The secondary functions behave exactly as the primary set but vary with respect to the parameters, as described in "Parameters for secondary direct-access functions" on page 236.

parseStringD

```
string parseStringD(
   string payload_field,
   string field_name,
   string default_nonExistent,
   string default_empty,
   string default_invalid
```

parseFloatD

```
float parseFloatD(
   string payload_field,
   string field_name,
   float default_nonExistent,
   float default_empty,
   float default_invalid)
```

parseIntergerD

```
integer parseIntegerD(
   string payload_field,
   string field_name,
   integer default_nonExistent,
   integer default_empty,
   integer default_invalid)
```

parseBooleanD

```
boolean parseBooleanD(
   string payload_field,
   string field_name,
   boolean default_nonExistent,
```

```
   boolean default_empty,
   boolean default_invalid)
```

The direct-access functions

Parameters for secondary direct-access functions

The parameters used by the sencondary direct-access function differ from those used by the primary functions as described in the following table:

Parameter	Description
default_nonExistent	The value to return if the field specified in *field_name* does not exist. The type of this parameter depends on the particular method.
default_empty	The value to return if the field specified in *field_name* has no value (is empty or null). The type of this parameter depends on the particular method.
default_invalid	The value to return if the field specified in *field_name* cannot be cast to the type requested or is otherwise invalid (e.g. contains binary escape codes). The type of this parameter depends on the particular method. Note that all types can be cast to a `string`, however, the opposite might not be possible. For example a `string` can be only be successfully cast to a float if it contains a valid float number, e.g. 1.0.

The direct-access functions

The indirect-access functions

The advantage of the direct-access functions is that they are straightforward to use. They parse through the payload on every call, so they are also the most appropriate functions to utilize if you only need to access a single field from within the payload.

However, if it is required to access several fields from within the payload or even the same fields repeatedly, it is more appropriate to use the indirect-access functions. In this case the payload is parsed only once, and a highly optimized representation of its contents is generated. This makes subsequent field access very fast, and the difference in performance over the direct access functions can be considerable, particularly if the payload contains a very large number of fields.

To use the indirect access functions you must first call `parsePayload(>payload_field)`, which returns a `chunk` object.

The parameters for the indirect-access functions have the same meaning as for the direct-access function parameters; see "Parameters for primary direct-access functions" on page 234.

The primary indirect-access functions are described in the following sections.

getString

```
string getString(
   chunk payload_chunk,
   string field_name,
   string default_value)
```

Access the `chunk` specified in *`payload_chunk`*, retrieve the field called *`field_name`*, and return its contents as a `string`. If the field does not exist, is empty, or is invalid, return the string literal specified in *`default_value`*.

getFloat

```
float getFloat(
   chunk payload_chunk,
   string field_name,
   float default_value)
```

Access the `chunk` specified in *`payload_chunk`*, retrieve the field called *`field_name`*, and return its contents as a `float`. If the field does not exist, is empty, is invalid, or cannot be cast to a valid `float`, return the float value specified in *`default_value`*.

getInteger

```
integer getInteger(
   chunk payload_chunk,
   string field_name,
   integer default_value)
```

Access the `chunk` specified in *`payload_chunk`*, retrieve the field called *`field_name`*, and return its contents as a `integer`. If the field does not exist, is empty, is invalid, or cannot be cast to a valid `integer`, return the integer value specified in *`default_value`*.

getBoolean

```
boolean getBoolean(
   chunk payload_chunk,
   string field_name,
   boolean default_value)
```

Access the `chunk` specified in *`payload_chunk`*, retrieve the field called *`field_name`*, and return its contents as a `boolean`. If the field does not exist, is empty, is invalid, or cannot be cast to a valid `boolean`, return the boolean value specified in *`default_value`*.

Using the Payload Extraction plug-in

Secondary indirect-access functions

The secondary indirect-access functions are as follows:

getStringD

```
string getStringD(
   chunk payload_chunk,
   string field_name,
   string default_nonExistent,
   string default_empty,
   string default_invalid)
```

getFloatD

```
float getFloatD(
   chunk payload_chunk,
   string field_name,
   float default_nonExistent,
   float default_empty,
   float default_invalid)
```

getIntegerD

```
integer getIntegerD(
   chunk payload_chunk,
   string field_name,
   integer default_nonExistent,
   integer default_empty,
   integer default_invalid)
```

getBooleanD

```
boolean getBooleanD(
   chunk payload_chunk,
   string field_name,
   boolean default_nonExistent,
   boolean default_empty,
   boolean default_invalid)
```

The indirect-access functions

Using the Time Format plug-in

The Time Format plug-in provides a set of functions to assist with date and time formatting. For a list of time zones, see *Using Dashboard Viewer.* , "Timezone ID Values". Specify an `import` statement in each monitor that requires the Time Format plug-in functions.

```
import "TimeFormatPlugin" as timeMgr;
```

This loads the Time Format plug-in and assigns the *timeMgr* alias to it.

Description

Windows	The plug-in is available as `TimeFormatPlugin.dll` in the `bin` directory.
UNIX	The plug-in is available as `libTimeFormatPlugin.so` in the `lib` directory. When you use the Time Format plug-in on any UNIX-based system, you must correctly configure the `TZ` environment variable so the plug-in can identify the correct locale. 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, the plug-in does not recognize `TZ=GB`. If you specify something like this, the plug-in uses Coordinated Universal Time (UTC).

Functions

The Time Format plug-in provides the following functions:

- "Time Format plug-in format functions" on page 239
- "getTime()" on page 239
- "Time Format plug-in parse functions" on page 240
- "parseTimeFromPattern()" on page 241

- "Time Format plug-in compile pattern functions" on page 242
- "getMicroTime()" on page 243

See also "Format specification for the Time Format plug-in functions" on page 243

Using Correlator Plug-ins in EPL

Time Format plug-in format functions

The format functions convert the `time` parameter to the local time and return that time in the format you specify.

Usage

```
string format(float time, string format)

string formatUTC(float time, string format)

string formatWithTimeZone(float time, string format, string tzName)
```

Usage description

time	Float that indicates the time you want to format. This value is the number of seconds since the epoch in UTC. This is the same format used by the `currentTime` variable. For information about the `currentTime` variable, see "Getting the current time" on page 169.
format	String that specifies the format that you want the returned time to have. For details about what you can specify for the `format` string, see "Format specification for the Time Format plug-in functions" on page 243.
name	String that specifies the name of a time zone.

The `format()` function converts the `time` parameter to the local time and returns that time in the format you specify.

The `formatUTC()` function returns the time specified in the `time` parameter in the format you specify. The `formatUTC()` function always returns UTC (GMT no matter what the local time is). This is true of all `*UTC` functions.

The `formatWithTimeZone()` function converts the `time` parameter, which is UTC, to the time in the time zone you specify and returns that time in the format you specify.

Using the Time Format plug-in

getTime()

The `getTime()` function returns the local time as a float of seconds since the epoch (UTC).

Syntax

```
float getTime()
```

This value has the same format as the `currentTime` variable. However, the `getTime()` function returns the actual local time whereas the `currentTime` variable contains the time that the event being

processed was received by the correlator. The time returned by the `getTime()` function is accurate to the millisecond.

The Windows and UNIX versions of the Time Format plug-in are not guaranteed to return the same time all the time. This is because the underlying system libraries that the plug-in relies on have different interpretations of what constitutes local time in certain country locales, in particular during summer time. This discrepancy is caused by the fact that at the epoch, January 1st 1970, Great Britain was temporarily one hour ahead of UTC. Some UNIX system libraries, like those on Solaris, account for this, others, like that on Windows, do not.

Using the Time Format plug-in

Time Format plug-in parse functions

The Time Format plug-in parse functions parse the value contained by the `timeDate` parameter according to the format you specify in the `format` parameter. Each function returns the resulting value as a float of seconds since the epoch.

Usage

```
float parse(string format, string timeDate)
float parseTime(string format, string timeDate)

float parseUTC(string format, string timeDate)
float parseTimeUTC(string format, string timeDate)

float parseWithTimeZone(string format, string timeDate, string tzName)
float parseTimeWithTimeZone(string format, string timeDate, string tzName)
```

Usage description

format	String that specifies the format of the value in the *timeDate* parameter. For details about what you can specify here, see "Format specification for the Time Format plug-in functions" on page 243.
timeDate	String that contains the time you want to parse.
name	String that specifies the name of a time zone.

Note: Release 4.1 added the `parse()`, `parseUTC()`, and `parseWithTimeZone()` functions to be consistent with the `format()`, `formatUTC()`, and `formatWithTimeZone()` functions. Previously, the Time Format plug-in provided only `parseTime()`, `parseTimeUTC()`, and `parseTimeWithTimeZone()`. The corresponding parse functions (for example, `parse()` and `parseTime()`) return the same results. There are no plans to deprecate the older functions.

Purpose

The `parse()` and `parseTime()` functions parse the value contained by the *timeDate* parameter according to the format you specify in the *format* parameter, interpreting it as a date and time in the local timezone. Each function returns the resulting value as a `float` of seconds since the epoch.

The `parseUTC()` and `parseTimeUTC()` functions parse the value contained by the *timeDate* parameter according to the format you specify in the *format* parameter. Each function interprets the *timeDate* as a UTC date and time. The returned value is a `float` of seconds since the epoch.

The `parseWithTimeZone()` and `parseTimeWithTimeZone()` functions parse the value contained by the *`timeDate`* parameter according to the format you specify in the *`format`* parameter, interpreting it as a date and time in the local timezone. The returned value is a `float` of seconds since the epoch.

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.
- Each function returns `NaN` if it cannot parse the specified string.
- If *`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 `parseWithTimeZone()` 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 `parseWithTimeZone()` 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`:

```
timeMgr.parseTime("yyyy.MM.dd G 'at' HH:mm:ss", "1996.07.10 AD at 15:08:56")
```

See also "Midnight and noon" on page 247.

The following examples both parse the *`timeDate`* string as having a time zone of `UTC+0900`.

```
timeFormat.parseWithTimeZone("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.

Using the Time Format plug-in

parseTimeFromPattern()

The `parseTimeFromPattern()` function parses the *`timeDate`* string according to the time/date parser object in the *`compiledFormat`* parameter. This function returns the result as a float of seconds since the epoch.

Usage

```
float parseTimeFromPattern(chunk compiledFormat, string timeDate)
```

Usage description

compiledFormat	Chunk that contains a time/date parser object. This chunk is the result of one of the `compilePattern()` functions. See "Time Format plug-in compile pattern functions" on page 242.
timeDate	String that represents a time and/or a date.

For more information about the return value, see the description of "Time Format plug-in parse functions" on page 240. For an example of how to use `parseTimeFromPattern()`, see "Time Format plug-in compile pattern functions" on page 242.

Note that the following are equivalent:

```
parse(format, timeDate)
parseTime (format, timeDate)
parseTimeFromPattern(compilePattern(format), timeDate)
```

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`.

Using the Time Format plug-in

Time Format plug-in compile pattern functions

The compile pattern functions return a time-parser object of type `chunk` that you can specify in multiple calls to the `parseTimeFromPattern()` function. Reusing a time-parser object is significantly more efficient than providing the same time pattern (`format`) in multiple calls to parse functions.

Usage

```
chunk compilePattern(string format)
chunk compilePatternUTC(string format)
chunk compilePatternWithTimeZone(string format, string name)
```

Usage description

format	String that specifies a set of instructions for parsing a time/date object.
name	String that specifies the name of a time zone.

Purpose

The `compilePattern()` function returns a time-parser object that can generate the time as a `float` value that corresponds to the local date/time string to be parsed.

The `compilePatternUTC()` function returns a time-parser object that can generate the time as a `float` value that corresponds to the UTC date/time string to be parsed.

The `compilePatternWithTimeZone()` function returns a time-parser object that can generate the time as a `float` value that corresponds to the date/time string to be parsed, in the specified timezone.

Examples

The following example shows how you can specify the `chunk` returned by a `compilePattern()` function in multiple calls to `parseTimeFromPattern()`:

```
chunk compiled:=timeMgr.compilePattern(pattern);
t1:=parseTimeFromPattern(compiled, time1);
t2:=parseTimeFromPattern(compiled, time2);
t3:=parseTimeFromPattern(compiled, time3);
t4:=parseTimeFromPattern(compiled, time4);
```

When you call a `compilePattern()` function and then the `parseTimeFromPattern()` function, the result is the same as calling one of the parse functions. The advantage of calling a `compilePattern()` function and then `parseTimeFromPattern()` is that it is faster. For example:

```
timeMgr.parse("yyyy.MM.dd G 'at' HH:mm:ss", "1996.07.10 AD at 15:08:56")
```

See also "Midnight and noon" on page 247.

If you use this pattern many times, it is faster to call `compilePattern()`, as follows:

```
Chunk c := timeMgr.compilePattern("yyyy.MM.dd G 'at' HH:mm:ss")
timeMgr.parseTimeFromPattern(c,"1996.07.10 AD at 15:08:56")
```

Using the Time Format plug-in

getMicroTime()

The `getMicroTime()` function returns a floating point (`float`) timestamp that represents the number of seconds since an unspecified epoch (a zero point).

The timestamp is accurate to one microsecond or better. The return value has no necessary relationship to wall time or correlator time and should be used only to compare similar timestamps.

Because the epoch is unspecified, you should use these timestamps in calculations only with other timestamps generated by the `getMicroTime()` function or with equivalent high-resolution timestamps generated by an Apama adapter.

For example, you might want to use high-resolution timestamps for calculating event processing latency in an Apama application. You can compare the timestamps across processes on the same machine. The timestamps are not comparable between different machines.

Using the Time Format plug-in

Format specification for the Time Format plug-in 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 Time Format plug-in 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 `SimpleDataFormat` class transforms the full form. For example, `EEEE` formats/parses `Monday`.
  - If you specify fewer than four letters, the `SimpleDataFormat` 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 `SimpleDataFormat` class transforms text. For example, `MMM` formats/parses `Jan`, while `MMMM` formats/parses `January`.
  - If you specify one or two letters, the `SimpleDataFormat` class transforms a number. For example, `M` formats/parses `1`, and `MM` formats/parses for `01`.

The following tables provides the meaning of each letter you can specify in a pattern. After the table, there are a number of combined examples.

Table 8. Descriptions of pattern letters in format strings

Symbol	Meaning	Presentation	Example	Sample Result
`G`	Era designator	Text	`G` `G`	`AD` `BC`
`y` (lower-case)	Year	Number	`yy` `yyyy`	`96` `1996`
`Y` (upper-case)	Year for indicating which week of the year. Use with the `w` 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`
			`MM`
			`MMM`
			`MMMM`
				`9`
				`09`
				`Sep`
				`September`

`d`	Day in month	Number	`d` `dd`	`7` `07`

Symbol	Meaning	Presentation	Example	Sample Result
			dd	25
h	Hour in AM or PM (1—12)	Number	hh	05
H	Hour in day (0—23) See also "Midnight and noon" on page 247.	Number	H HH HH	0 05 14
m	Minute in hour See also "Midnight and noon" on page 247.	Number	m mm mm	3 03 55
s	Second in minute	Number	s ss ss	5 05 59
S	Fractional second	Number	S SS SSS	2 20 200
E	Day of week	Text	E EE EEE EEEE	Fri Fri Fri Friday
e	Day of week (1—7) This is locale dependent. Typically, Monday is 1.	Number	e	4
D	Day in year	Number	D DD DDD DDD	7 07 007 123
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 (lower-case)	Week in year. Use with uppercase Y. The week that contains January 1st is week 1. For example, if a week starts on Monday and ends on Sunday, and if January 1st is a Sunday, then week 1	Number	The first example below uses uppercase Y. The second example shows the difference when you use lowercase y. "'Week' w YYYY" "'Week' w yyyy"	Suppose you are transforming December 31st, 2008, which is a Wednesday. "Week 1 2009" "Week 1 2008"

Symbol	Meaning	Presentation	Example	Sample Result
	contains December 26 - 31 plus January 1.			
W (upper-case)	Week in month. The week that contains the 1st of the month is week 1. For example, if a week starts on Monday and ends on Sunday, and if July 1 is a Friday (5), then week 1 of July contains June 27 - 30 and July 1 - 3.	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	Time zone	Text	z	Pacific Standard Time
Z	Time zone (RFC 822)	Number	Z	-0800
v	Generic time zone	Text	v	Pacific Time
V	Time zone abbreviation	Text	V	PT
VVVV	Time zone location	Text	VVVV	United States (Los Angeles)
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 timeDate string
`yyyy.MM.dd G 'at' HH:mm:ss z`	`1996.07.10 AD at 15:08:56 PDT`
`EEE, MMM d, ''yy`	`Wed, July 10, '96`
`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, PST`
`yyyyy.MMMMM.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 formal `"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 Time Format plug-in

Using the Log File Manager plug-in

Apama includes the Log File Manager plug-in to assist in logging status and diagnostic information from EPL and Java applications. However, rather than using the Log File Manager plug-in directly, you typically route events to the `LoggingManager` monitor and the `LoggingManager` monitor calls the functions in the Log File Manager plug-in based on the routed events.

It can be convenient to use the Log File Manager because of its comma-separated value (CSV) output and the ability to query the logging level.

Note: An alternative to the Log File Manager plug-in is to use the `engine_management` utility to set logging levels and log files, and then insert `log` statements in your EPL code. You can expect better performance from this alternative. See "Setting logging attributes for packages, monitors, and events", in "Shutting down and managing components" in the "Event Correlator Utilities Referenece" section of *Deploying and Managing Apama Applications*, and "Specifying log statements" on page 179.

The `LoggingManager.mon` file is in the `monitors` directory of your Apama installation dirctory. This file defines events and actions that help you log information. The EPL in `LoggingManager.mon` uses both the Log File Manager plug-in and the Time Format plug-in. As this is quite comprehensive you are unlikely to need to use the Log File Manager plug-in directly, and the latter is only documented here for completeness.

If you find that you do need to directly use the Log File Manager plug-in, specify something like the following right after the `monitor` statement for the monitor that needs to use the plug-in. This loads the Log File Manager plug-in and assigns it the alias `logMgr`:

```
import "Loggingplugin" as logMgr;
```

The Log File Manager plug-in is available as `libLoggingPlugin.so` in the `lib` directory on Unix. On Microsoft Windows it is available as `LoggingPlugin.dll` in the `bin` directory.

Using Correlator Plug-ins in EPL

Plug-in functions

This section describes the functions provided by the Log File Manager plug-in. The next section describes the events that you route to the `LoggingManager` monitor in order to call these functions.

Before using any of the functions you should define a variable of type `chunk` and then pass it to each method. If the variable is local it should be initialized with `new chunk`, though this is not necessary for globals or event members.Since plug-ins are by definition stateless, this mechanism is required to preserve state (such as the log file's name and location) across function calls.

init

```
boolean init (
   string filename,
   boolean append,
   chunk state)
```

Initialize the Log File Manager plug-in. Set the `filename` parameter to indicate the path and filename of the file to use, set `append` to `true` if you wish to continue using an existing log file or set to `false` to recreate/overwrite it. You can set `filename` to the special keyword `"stdout"` to log to standard output, and `"stderr"` to log to standard error. What this method does is encode these parameters into the

`state` chunk, so that you can then pass it to all the other functions. Returns `false` if successful, `true` otherwise.

setLevel

```
boolean setLevel (
   chunk state,
   string level)
```

Set the current logging level. You can set *`level`* to one of `"trace"`, `"debug"`, `"info"`, `"warn"`, `"error"`, or `"fatal"`— in order of increasing priority. Only log messages at the current priority level or higher will be output. This property is encoded within the `state` chunk. Returns `false` if successful, `t rue` otherwise.

setFormat

```
boolean setFormat(
   chunk state,
   string format)
```

Set the current log output format. You can set *`format`* to either `"normal"` or `"csv"`. The former will print out regular logging messages, while the latter will apply special comma-separated format to the log output. The comma-separated format is particularly useful for loading the logging output into a spreadsheet or database for further analysis. Returns `false` if successful, `true` otherwise.

trace

```
trace(
   chunk state,
   string message)
```

Log *`message`* at the `trace` level.

debug

```
debug(
   chunk state,
   string message)
```

Log *`message`* at the `debug` level.

info

```
info(
   chunk state,
   string message)
```

Log *`message`* at the `info` level.

warn

```
warn(
   chunk state,
   string message)
```

Log *`message`* at the `warn` level.

error

```
error(
   chunk state,
   string message)
```

Log *message* at the `error` level.

fatal

```
fatal(
   chunk state,
   string message)
```

Log *message* at the `fatal` level.

logAt

```
logAt(
   chunk state,
   string levelAt,
   string message)
```

Log *message* at the level specified by the *levelAt* parameter. This can be one of `"debug"`, `"info"`, `"warn"`, `"error"`, or `"fatal"`.

Using the Log File Manager plug-in

Events that can be routed to the LoggingManager monitor

The following events can be routed to the `LoggingManager` monitor. Note that the default settings are to use `"INFO"` for the log level, `"normal"` for the logging format, and `"stdout"` for the file:

LogEvent

```
event LogEvent {
   string level;
   string message;
}
```

Informs the monitor to log *message* at a set *level*, which should be set to one of “`FATAL`”, “`ERROR`”, “`WARN`”, “`INFO`”, “`DEBUG`”, or “`TRACE`”.

SetLogFile

```
event SetLogFile {
   string filename;
   boolean append;
}
```

Sets the log file *filename* should be set to identify the file to use, while *append* specifies whether to append to the file if it already exists. On receipt of this event log file rotation is de-activated; see the `SetLogRotating` event, below.

SetLogRotating

```
event SetLogRotating {
   string filenameTemplate;
   boolean useInternalTime;
   boolean append;
   integer rotatePeriod;
   integer rotateHour;
   integer rotateMinute;
}
```

Sets a log file to write all messages, and activates log file rotation. The parameters have the following meanings:

- `filenameTemplate` defines a template for naming the log file. The template can contain Time Format plug-in format specifiers, which should be surrounded by quotation marks, for example `"correlator_'ddMMyyyy'.log"`, see "Format specification for the Time Format plug-in functions" on page 243.
- `append` indicates if the log message should be appended to the file if it already exists.
- `useInternalTime` should be set to `true` to use the correlator's internal time, not the external system time, when creating the log file's name.
- `rotatePeriod` indicates the periodicity of the rotation; `0`=every minute, `1`=hourly, `2`=daily, `3`=weekly on Sundays, `4`=monthly on the first of the month.
- `rotateHour` specifies the hour at which to perform a rotation.
- `rotateMinute` specifies the minute in the hour at which to perform a rotation.

SetLogRotatingDaily

```
event SetLogRotatingDaily {
}
```

Sets log file rotation to daily (00:00) using the default filename template (`"correlator-'dd'-'MM'-'yyyy'.log"`).

SetLogLevel

```
event SetLogLevel {
   string level;
}
```

Sets the log level of the logger. Only messages with a level equal or higher to the level are logged. The `level` field should be set to one of "`FATAL`", "`ERROR`", "`WARN`", "`INFO`", or "`DEBUG`".

SetLogFormat

```
event SetLogFormat {
   string format;
}
```

Sets the text format to log messages. The *`format`* parameter can be either "`normal`" or "`csv`". The former specifies the regular format of logging messages, while the latter specifies a special comma-separated format. The comma-separated format is particularly useful for loading the logging output into a spreadsheet or database for further analysis.

SetDefaultLogFile

```
event SetDefaultLogFile {
}
```

Closes any currently open log file, deactivates log file rotation and directs subsequent log messages to the default log location — currently, "`stdout`". The logging level is not changed by this event.

Using the Log File Manager plug-in

Including the Log File Manager in a project

To use the Log File Manager plug-in, you need only add the Logging Manager bundle to your project:

1. In Apama Studio, to add a bundle to your project, open the project in the Apama Developer perspective.

2. In the Developer Project View, right-click the project name and select Properties from the menu that appears.
3. In the Properties dialog, select MonitorScript Build Path (under Apama) and then select the Bundles tab.
4. Click Add to display a list of Apama Studio bundles.
5. Select the Logging Manager bundle and click OK twice.

Using the Log File Manager plug-in

Loading the Log File Manager when deploying with the Management & Monitoring console

When you use the Management & Monitoring console to deploy an Apama project, the Management & Monitoring component automatically loads the Log File Manager if it processes a monitor that contains `import Loggingplugin`.

Using the Log File Manager plug-in

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 265.

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 218.

See "Using the Apama Database Connector" in *Deploying and Managing Apama Applications*.

Information for using the MemoryStore is organized into the topics listed below. For details about the event types that provide the MemoryStore interface, see the MemoryStore ApamaDoc: *apama_install_dir*`\doc\ApamaDoc\index.html`.

- "Introduction to using the MemoryStore" on page 253
- "Overview of MemoryStore events" on page 254

- "Overview of steps for using the MemoryStore" on page 254
- "Adding the MemoryStore bundle to your project" on page 255
- "Preparing and opening stores" on page 255
- "Description of row structures" on page 257
- "Preparing and opening tables" on page 259
- "Using transactions to manipulate rows" on page 260
- "Creating and removing rows" on page 262
- "Iterating over the rows in a table" on page 262
- "Requesting persistence" on page 263
- "Exposing in-memory or persistent data to dashboards" on page 264
- "Restrictions affecting MemoryStore disk files" on page 265
- "Using the distributed MemoryStore" on page 265

Using Correlator Plug-ins in EPL

Introduction to using the MemoryStore

Data that the MemoryStore stores must be one of the following types: `boolean`, `float`, `integer` or `string`.

To use the MemoryStore, you add the MemoryStore Plugin bundle to your Apama project. 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 data views 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 218.

For information on using the MemoryStore in a distributed fashion, see "Using the distributed MemoryStore" on page 265.

Using the MemoryStore

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.

Using the MemoryStore

Overview of steps for using the MemoryStore

To use the MemoryStore, you must first add the MemoryStore bundle to your project, unless you are using the distributed MemoryStore. (If you are using the distributed MemoryStore, instead of adding 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 269.)
After you add 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 see further information, "Notifications" on page 273.
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.

Using the MemoryStore

Adding the MemoryStore bundle to your project

To use the MemoryStore, you need only add the MemoryStore bundle to your project as follows. (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 269.)

1. In Apama Studio, to add a bundle to your project, open the project in the Apama Developer perspective.
2. In the Project Explorer, right-click the project name and select Apama > Add Bundle from the pop-up context menu. The **Add Bundle** dialog is displayed.
3. In the **Add Bundle** dialog, select The MemoryStore bundle and click OK.

Adding the `MemoryStore` bundle to your project makes the `MemoryStore.mon` file available to the monitors in your project. When you run your project, Apama Studio 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` utility to manually inject your EPL, instead of using Apama Studio, and you want to expose MemoryStore tables to dashboards, you need to inject the `MemoryStoreScenarioImpl.mon` monitor, which is in the same directory as the `MemoryStore.mon` file.

Using the MemoryStore

Preparing and opening stores

The first step for storing data in memory is to create an instance of a `Storage` event. You use the `Storage` event 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.

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 {
   Storage storage;
   Store store;

   action onload() {
      integer id := storage.prepareOrCreate("storename", "/tmp/example.dat");
      Finished f;
      on Finished(id,*,*):f
      onStorePrepared(f);
      ...
  }
}
```

After a store has been successfully prepared, you can open it:

```
action onStorePrepared(Finished f) {
   if not f.success then { 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" on page 270.

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.

Using the MemoryStore

Description of row structures

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"];
```

The `Schema` event has additional members that indicate how to publish the table. See "Exposing in-memory or persistent data to dashboards" on page 264.

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.

`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

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
- `float` types are `0.0`
- `integer` types are `0`
- `string` types are empty (`""`)

`Row` events define actions that do the following:

- Get and set `boolean`, `float`, `integer`, and `string` fields by name. 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.
- 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.

Using the MemoryStore

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. 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,*,*):f onTablePrepared(f);

action onTablePrepared(Finished f) {
   if not f.success then { 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 the MemoryStore

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 in-memory rows. After you commit the changes you have made to your `Row` events, the updated in-memory 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 in-memory table. If nothing else modified the in-memory 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 in-memory row in the table has been modified, the monitor instance that called this action dies, leaving the in-memory table unchanged.

- `Row.tryCommit()` — Behaves like `commit()` except that it does not kill the monitor instance upon failure. If the in-memory row in the table has been modified, this action returns false and leaves the in-memory 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 in-memory row. In other words, if the in-memory row has been modified, this action does the following:
  - Leaves the in-memory row 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.

Using the MemoryStore

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 optimising 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 then { row.update(); }
}
```

The `row.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);
```

This example is equivalent to the previous one, both in behavior and performance. Which to use is a matter of context, style and personal preference.

Using the MemoryStore

Creating and removing rows

To create a row in a table, call the `get()` action on the table you want to add the row to. The action declaration 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();
}
emit "This example has been run " +n.toString() +" time(s) before";
```

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.

Using the MemoryStore

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 := begin();
while not i.done() {
   Row row := i.getRow();
   if row.inTable() then {
      // Put code here to read the row in the way you want.
   }
   i.step();
}
```

The following sample code modifies table content:

```
Iterator i := 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 302.) 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.

Using the MemoryStore

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,*,*):f onPersisted(f);

action onPersisted(Finished f) {
   if not f.success then { 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.

Using the MemoryStore

Exposing in-memory or persistent data to dashboards

You can expose committed in-memory data or committed persistent data as DataViews for use by dashboards. 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.

For information about DataViews, see "Making Application Data Available to Clients" on page 287. For information about dashboards, see *Building Dashboards*.

Using the MemoryStore

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 MemoryStore

Using the distributed MemoryStore

This topic describes Apama's 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. This topic describes 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.

Using the MemoryStore

Overview of the distributed MemoryStore

The MemoryStore supports several types of stores as described in "Using the MemoryStore". 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 a `Storage` object. 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 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 provides 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.
- 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 273.

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:

- Requires more data to be stored than will fit on any single node.
- Elastic (changing) processing capacity required.
- Highly available system needs continuous access to data, even if some nodes fail, and with minimal recovery time.
- High throughput 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:

- Very low latency (sub-millisecond) access to data.
- Very high throughput (>10,000 requests/second) to a single row - the distributed store only scales out well if different rows are being accessed.

Supported providers

Apama includes a driver for connecting to Terracotta BigMemory Max, which provides unlimited in-memory data management across distributed servers. See "BigMemory Max driver specific details" on page 277 for using the BigMemory Max driver.

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 281.

Configuration

In order to use a distributed memory store, a set of configuration files must be created in your project and provided to the correlator. These configuration files typically come in pairs, a `.properties` and `-spring.xml`. 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 271.

Using the distributed MemoryStore

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 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	`Table.get("row1") = "abc"`	

Time:	Monitor1 (on node 1)	Monitor2 (on node 2)
1.2	Change row to be `"abcdef"`	`Table.get("row1") = "abc"` (cached locally)
1.3	`Row.commit("row1" as "abcdef")` succeeds	
1.301	Send event to Host 2	
1.302		Receive event from Host 1
1.303		`Table.get("row1") = "abc"` from local cache)
1.4		Update row to be `"abcghi"`
1.5		`Row.commit("row1 as "abcghi")` fails (not last value)
1.6		`Row.update() = "abcdef"`
1.7		Update row to be `"abcdefghi"`
1.8		`Row.commit("row1" as "abcdefghi")` 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.

Overview of the distributed MemoryStore

Using a distributed store

Distributed stores make use of Java Distributed cache technologies (the specific technologies depend on the driver you select). When you start a correlator with the `--distMemStoreConfig` option (enabled automatically if you use Apama Studio to add a Distributed Memory store configuration to your project), the correlator automatically starts with an embedded Java virtual machine. This JVM is shared by any Apama applications using a Distributed MemoryStore or Correlator-integrated JMS and any Apama Java applications.

A distributed store is defined by a bean in a Spring XML configuration file. The bean specifies the properties that configure the distributed store and the bean's name, which is the name of the store. When an Apama application prepares a distributed store, using the `prepareDistributed()` action, it supplies the name of the bean. For more information on properties used in the configuration file, see "Configuring a distributed store" on page 271 and "Configuration files for distributed stores" on page 274.

Depending on the distributed cache provider you select, the data may be stored in the Java heap. If so, you may need to set an appropriate size for the Java heap, for example, by specifying `-J-Xmx2048M` (to specify a 2GB heap) on the command line that starts the correlator. If you are using BigMemory Max off-heap data, you may need to supply a `-J-XX:MaxDirectoryMemorySize=` command line argument as well. For details, see "http://terracotta.org/documentation/4.0/bigmemorymax/get-

started/quick-start" on page . JVM options must be placed on the correlator command line and prefixed with `-J`. In Apama Studio, this can be configured by opening the project's launch configuration, editing the properties of the Correlator component and selecting the Maximum java off-heap storage in Mb option. See .

The main steps in configuring a distributed store are:

- If using BigMemory Max, configure and start at least one Terracotta Server Array Node.
- In Apama Studio, add the Distributed MemoryStore adapter to the project
- Add a store to the Distributed MemoryStore settings
- Choose a store name that will be used in the EPL application to refer to the store. This is used as the bean's name in the configuration files.
- Provide a driver class name to use a distributed cache of your choice. (If using BigMemory Max, Apama Studio creates a BigMemory Max configuration with the classpath already set)
- Specify a *cluster name*. The exact meaning of *cluster name* depends on the driver. For the BigMemory Max driver, it is a comma-separated list of the `host:port` pairs that identify the Terracotta Server Array nodes. Best practice is to list all nodes configured in the cluster.
- Specify the classpath for both the driver and the distributed cache implementation `.jar` files. (If using Apama Studio's BigMemory Max support, you only need to specify the installation directory for BigMemory Max)
- Specify any other parameters needed by the driver. For further reference, see "Creating a distributed MemoryStore driver" on page 281.

Specifying a cluster name

A cluster name should be provided when opening a distributed store. Some third-party drivers and distributed caches use the cluster name as an identifier, that is, they do not interpret the name in any way. Many distributed caches use broadcast or multicast to automatically discover other cluster nodes on the same network with the same name configured. Thus, during development and testing, a name that is different to the name used by your production system should be used. This is a good practice to follow even if the systems are on separate networks. Cluster names are specified in properties files, which should be different between development and production environments.

You should not create more than one store with the same cluster name on any one correlator.

Overview of the distributed MemoryStore

Configuring a distributed store

Configuring a distributed store consists of adding the Apama Distributed MemoryStore adapter bundle to an Apama project, adding a distributed store to the project, and specifying property settings.

Using the distributed MemoryStore

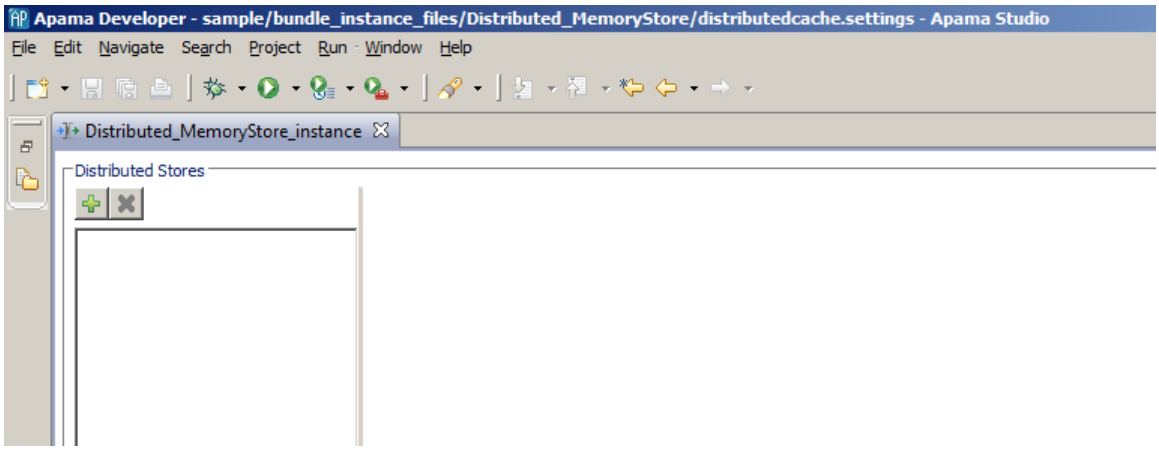
Adding distributed MemoryStore support to a project

To add a distributed store to a project using Apama Studio:

1. In the **Project Explorer**, right-click the project name and select Apama > Add Adapter from the pop-up menu. The **Add Adapter Instance** dialog is displayed.

2. In the Choose adapter field select Distributed MemoryStore (Supports using a distributed cache from MemoryStore) from the list of available adapters.
3. Click OK.

Apama Studio adds the adapter bundle to the project's `Adapters` node and opens the adapter instance in the Apama Studio Distributed MemoryStore editor. The editor is initially blank and the Distributed Stores field contains no distributed stores.

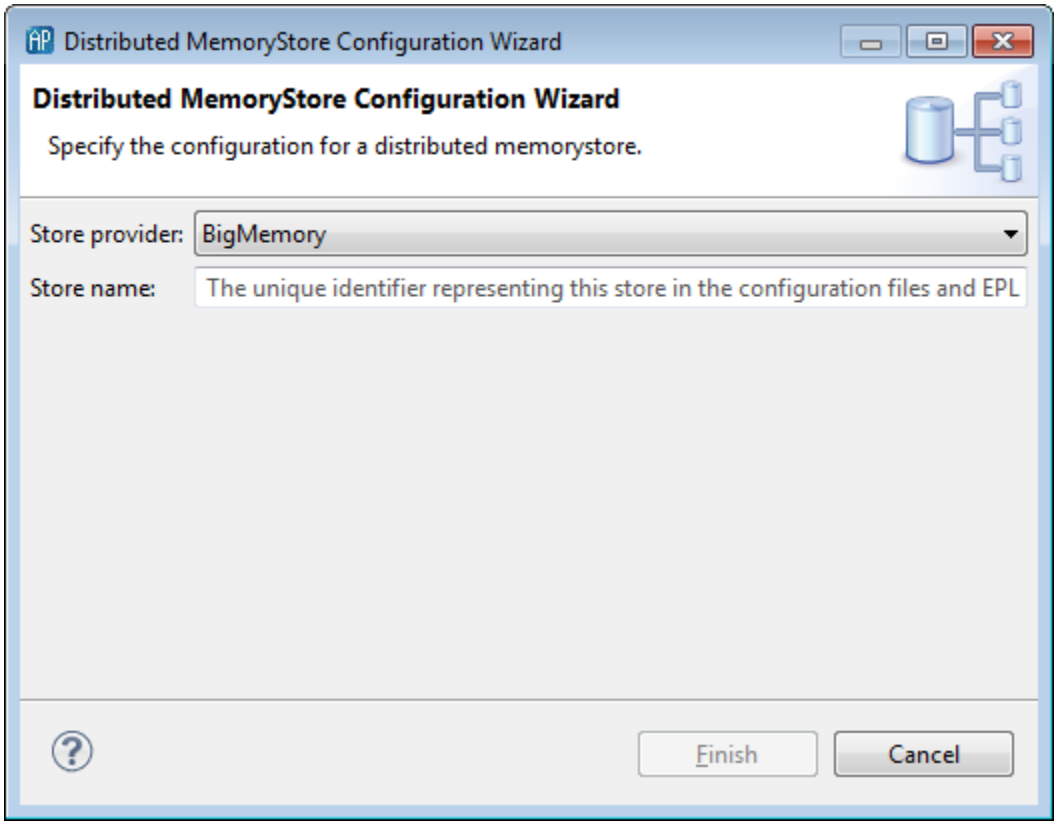


Configuring a distributed store

Adding a distributed store

To configure a new distributed store for use in this project:

1. In the Distributed MemoryStore editor's Distributed Stores panel, click the Add Store button ( ). The **Distributed MemoryStore Configuration** wizard opens:



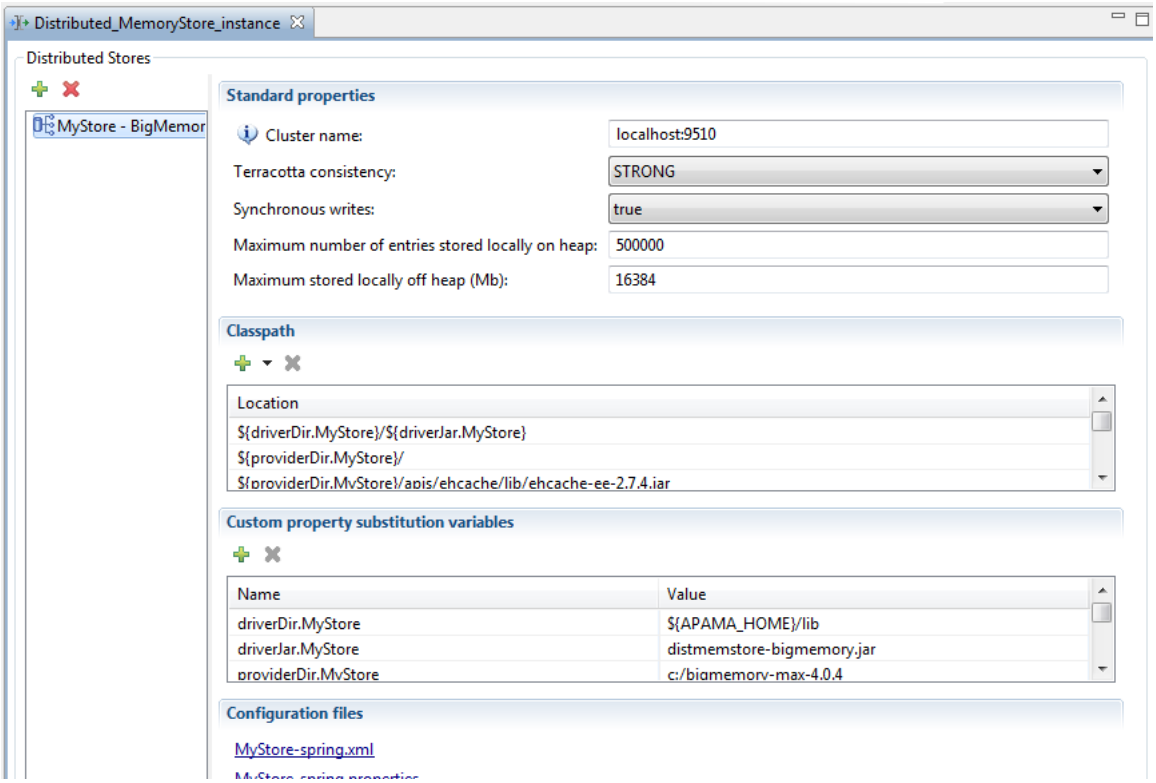
2. In the **Distributed MemoryStore Configuration** wizard, specify the following:
   a. In the Store Provider field, select the third-party cache provider from the drop-down list. If you are using a driver supplied by Apama, such as BigMemory Max, select it from the drop-down list; otherwise select Other.
   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.
3. Click Finish.

Apama Studio adds the name of the store to the Distributed Stores panel in the editor and adds the resources for the store to the project. The default configuration settings for the store are displayed in the editor.

Configuring a distributed store

Configuring a distributed store

You can configure the frequently used settings for a distributed store in Apama Studio's Distributed MemoryStore editor. These settings are those in the `.properties` file. For other settings you need to edit the `.xml` file directly.



1. In the Standard Properties section of the editor, specify the properties required by the third-party distributed cache in use.
2. (Only required if Other was used as the store type.) In the Classpath section, specify the names of the required provider-specific `.jar` files.
   a. Click the Add Location button ( ).
   b. In the new entry, 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. (e.g. use `${installDir.mystore}/lib/my.jar`)
3. In the Custom Property Substitution Variables section, specify the name and values of additional substitution ${...} variables (if any) used by the distributed cache. The `.properties` file contains substitution variables that are used by the `.xml` configuration file.
   a. Click the Add button ( ). A new line will be added to the list of substitution variables.
   b. In the new entry, specify the name and value of the substitution variable you want to add.
4. (If needed.) In the Configuration Files you can access the Spring `.xml` and `.properties` files. Click on the file name link to open them in the appropriate Apama Studio editor.

For more information on specifying property values, see "Configuration files for distributed stores" on page 274

Configuring a distributed store

Launching a project that uses a distributed store

When you add the Distributed MemoryStore adapter bundle to an Apama Studio project, the launch configuration is automatically updated to set the `--distMemStoreConfig` start-up option.

The maximum Java heap size and off-heap storage can be set in the Correlator Configuration dialog in the Run Configurations dialog.

Configuring a distributed store

Interacting with a distributed store

Once prepared, a distributed store behaves much like other MemoryStore `Store` objects as described in "Using the MemoryStore" on page 252. 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 273, below.
- Exceptions – In an in-memory store, only the `Row.commit()` action can throw exceptions. However, in a distributed store, most actions can throw exceptions. Exceptions represent a 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 "Catching exceptions" on page 176. The following are some of the actions that can throw exceptions:
  - `Table.get()`
  - `Table.begin()`
  - `Iterator.next()`
  - `Row.commit()`
  - `Row.update()`
- Performance differences – See "Overview of the distributed MemoryStore" on page 265 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` 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 BigMemory Max supports notifications, it does 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.

Using the distributed MemoryStore

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 will have the following files:

- *`storeName`*`-spring.xml`
- *`storeName`*`.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` *`configDir`* argument, it will load all XML files matching `*-spring.xml` in the specified configuration directory, and also all `*-spring.properties` files in the same directory. (Note, the correlator will not start unless the specified directory contains at least one configuration file.)

When using Apama Studio, these files are generated automatically. New *`storeName`*`-spring.xml` and *`storeName`*`.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 Apama Studio to customize more advanced configuration aspects. To edit the XML, 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 the Design and Source views using the 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. 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 275. 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 281.

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 some customers 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 will load 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 Apama Studio and to use different values during deployment to a production environment using the Apama Ant macros.

Although `.properties` and `-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*`-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

The following four standard properties are supported by Apama distributed cache drivers. These properties should be supported by customer-developed implementations as well.

- `clusterName` – This is a required property. It is a provider-specific string. For BigMemory Max, this is a comma-separated list of `host:port` pairs that identify the servers in the Terracotta Server Array. Some other caches use this as just a name, used to group together distributed store nodes that communicate with each other and share data. Store objects with the same `clusterName` values should operate as a single cluster, sharing data between them. Most providers require this property and will fail to start if it is not set. Care must be taken to ensure that different clusters, and thus `clusterName` 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. Apama's BigMemory Max driver makes it easy to avoid this pitfall since it requires a list of `host:port` pairs. 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 `_testing` or `_production` to the `clusterName` to indicate clearly which environment it belongs to.
- `logLevel` – This is an optional property; the default is provider-specific, but typically is the same as the correlator log level. The `logLevel` property is an Apama log level string (compatible with `com.apama.util.Logger`) such as `ERROR`, `WARN`, `INFO`, `DEBUG` which will be used to set the log level for the provider if possible (some providers will write to the main correlator log file, through log4j or the Apama Logger, but others may write to a separate file). 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 stdout being filled with third party distributed store messages while logging a small number of the most important messages.
- `backupCopies` – This is an optional property; the default is 1. The `backupCopies` 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). Note, 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. 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.
- `initialMinClusterSize` – This is an optional property. It specifies the minimum number of nodes a cluster must have before the `Finished` event is sent in response to a call to `prepareDistributed`.

This provides a way to make sure that a cluster is fully ready for correlator nodes to request and process data. The default is `1`, which specifies that a `Finished` event is sent without waiting for additional nodes when preparing the distributed store.

- `rowChangedOldValueRequired` - Indicates whether the old value is required when there is a notification that a row has changed. If set to false, the value of `oldFieldValues` is empty for `RowChanged.changeType.UPDATE` events. If true, the previous value is available. This currently cannot be set to true for BigMemory Max. The default is true.

If all four 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"/>
</bean>
```

Using the distributed MemoryStore

BigMemory Max driver specific details

Apama Studio can create configuration files for BigMemory Max; the BigMemory Max installation directory (where the zip files were unpacked) needs to be specified as the providerDir property.

The `.properties` file for a distributed store contains an option for choosing consistency. The options are `STRONG` or `EVENTUAL` consistency. See the BigMemory Max documentation for the trade-offs between these two modes.

Use the *`storeName`*`-spring.properties` file to set configuration properties for the BigMemory Max driver. Documentation is available here: "http://terracotta.org/files/documentation/Terracotta-Documentation-4.0.pdf" on page

Using off-heap storage requires setting `-XX:MaxDirectMemorySize=`. Specify this in the command line for starting the correlator as `-J-XX:MaxDirectMemorySize=`. See the documentation "http://terracotta.org/files/documentation/Terracotta-Documentation-4.0.pdf" on page for recommendations for specifying the value of this property. In Apama Studio, when you add a correlator to a correlator launch configuration you can select the Maximum java off-heap storage in Mb option. See .

You can set the following BigMemory Max driver properties in the `-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 `-spring.xml`configuration file using the `ehcacheConfigFile`. If this is done, many of the properties in the `spring.xml` configuration file will be ignored; the settings derived from the `ehcache.xml` file will be used instead. Refer to the BigMemory Max documentation if using an `ehcache.xml` file.

Property Name	Type	Description
`cache Configuration`	CacheConfiguration	EHCache `CacheConfiguration` bean, shared by all caches (`Table`s). Typically used as a compound bean name, for example, `cacheConfiguration.overflowToOffHeap`.

cacheDecorator Factory	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.
cacheDecorator FactoryProperties	Properties	Properties to pass to a cacheDecoratorFactory. Allows use of the same class for many caches.
clusterName	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	Configuration	EHCache Configuration bean. Typically used as a compound bean name, for example, configuration.monitoring.
maxMBLocal OffHeap	long	Number of MB of local off-heap data. Total across all tables, per correlator process.
pinning	String	Either an attribute value of "inCache" (default) or "localMemory" or a <null/> XML element (i.e.<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
terracotta Configuration	Terracotta Configuration	EHCache TerracottaConfiguration bean. Typically used as a compound bean name, for example, terracottaConfiguration.consistency.
ehcache ConfigFile	String	Path to an ehcache.xml configuration file. Note that if this is specified, any other properties listed in this table will be ignored.

You can set the following BigMemory Max driver properties in the `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	Type	Description
`backupCopies`	`int`	Ignored. Not supported. The number of backups is governed by the TSA topology defined in tc-config.xml used to configure the TSA nodes.
`initialMin ClusterSize`	`int`	The minimum cluster size (number of correlators) that must be connected for prepare to finish.
`logLevel`	`String`	The log level.
`rowChangedOld ValueRequired`	`boolean`	Whether to expose old values in `rowChanged` events. Must be set to false.
`useCompare AndSwap`	`boolean`	Whether to use compare and swap (CaS) operations or just put/remove. Some versions of BigMemory Max support only CaS in `Strong` consistency.
`useCompare AndSwapMap`	`Map(String, Boolean)`	Per-table (cache) configuration for whether to use CaS or put/remove.

The following compound properties are also exposed in the `.properties` file, or set by default in the `spring.xml`configuration file:

Property Name	Type	Notes
`cacheConfiguration. eternal`	`boolean`	Disables expiration (removing old, unused values) of entries if true. Set to true in the default `spring.xml` configuration file.
`cacheConfiguration. maxEntriesLocalHeap`	`int`	The number of entries for each table. This is the `maxEntriesLocalHeap` entry in the `.properties` file.
`cacheConfiguration. overflowToOffHeap`	`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.
`pinning`	`String`	Set to `inCache` by default.
`terracotta Configuration. localCacheEnabled`	`boolean`	Whether to cache entries in the correlator process. Set to true in the default `spring.xml` configuration file.
`terracotta Configuration. clustered`	`boolean`	Whether to use a TSA. Set to true in the default `spring.xml` configuration file.
`terracotta Configuration. consistency`	`String`	Either `'STRONG'` or `'EVENTUAL'`. `STRONG` gives MemoryStore-like guarantees, while `EVENTUAL` is faster but may have stale values read. This is the `terracottaConfiguration.consistency` entry in the `.properties` file.
`terracotta Configuration. synchronousWrites`	`boolean`	If true, then data is guaranteed to be out of process by the time a `Row.commit()` action completes. Disabling this can increase speed. This is the `terracottaConfiguration.synchronousWrites` entry in the `.properties` file.

Note the following when using the BigMemory Max driver:

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`

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.

Configuration files for distributed stores

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 all such bean property values are specified using `${varname}` substitution variables specified in `.properties` files for test vs. production environments. For example, for 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 275.

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 `distMemStoreConfig` directory 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.

Using the distributed MemoryStore

Creating a distributed MemoryStore driver

The Apama installation includes a driver for integrating the distributed MemoryStore with 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 section of the Apama documentation presents an introduction to the SPI and a description of its essential elements.

Complete Javadoc information for the SPI is available in `doc/javadoc/index.html` in your Apama installation - 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
  - Java bean-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 will be called by Spring during correlator start-up 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 distributed store - `spring.xml` configuration file.

- `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 `addRowChangedListener` and `removeRowChangedListener`, 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`.

- `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 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 under the `samples/distmemstore_driver/bigmemory` path in the Apama installation directory. 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 distributed MemoryStore

Using the Management interface

The Management interface defines actions that let you do the following:

- Obtain information about the correlator
- Request a persistence snapshot

To use the Management interface, add the `Correlator Management` bundle to your Apama project. Alternatively, you can directly use the EPL interfaces provided in `APAMA_HOME\monitors\Management.mon`.

Obtaining information about the correlator

The Management interface provides the following actions for obtaining information about the correlator that the Management interface is being used in:

- `getHostname()` - Returns the host name of the host the correlator is running on. The host name is dependent on the environment's name resolution configuration, and the name can be used only if the name resolution is correctly configured. The name is the same as that logged in the correlator log file, for example, `dev3.acme.com`.
- `getComponentPort()` - Returns the port the correlator is running on.
- `getComponentPhysicalId()` - Returns the physical ID of the correlator.
- `getComponentLogicalId()` - Returns the logical ID of the correlator.
- `getComponentName()` - Returns the name that is used to identify the correlator. You can set this name by specifying the `-N` correlator command line flag (or by means of the `extraArgs` attribute in the ANT macros). The default name of the correlator is `correlator`.

These actions are defined in the `com.apama.correlator.Component` event. There are `engine_management` utility options that you can specify

- To retrieve the same information from outside the correlator
- Or to retrieve the same information for IAF or sentinel agent processes

The correlator also logs all of these values to its log file at startup.

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 205.

The Management interface defines the `Persistence` event:

```
package com.apama.correlator;
event Persistence {
   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() {
      Number n;
      on all Number(*):n {
         myNumbers.append(n.i);
         counter := counter + 1;
         if(counter % 10 == 0) then {
            doCommit();
         }
      }
   }

   action doCommit() {
      Persistence p = new Persistence;
      p.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.

Using Correlator Plug-ins in EPL

Interfacing with user-defined correlator 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.

A developer can address this situation by writing custom correlator plug-ins using Apama's C and C++ Plug-In Development Kits. A plug-in consists of an appropriately formatted library of C or C++ functions which can be called from within EPL while Apama is executing monitors. Apama and its event correlator components do 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.

Once a plug-in is developed, a developer can call the functions it contains directly from a monitor in EPL, passing EPL variables and constants as parameters, and getting return values that can be manipulated. For information on developing your custom event correlator plug-in, see *Writing Correlator Plug-ins*.

Note: The correlator's plug-in interface is versioned. For a correlator plug-in to be compatible with an event correlator they both need to support the same plug-in interface version. See *Developing Correlator Plug-ins* for information about how to ensure that your correlator plug-in is compatible with the event correlator it will run in.

In order to access a function implemented in an event correlator plug-in, the developer must first import the plug-in, for example:

```
import "apama_math" as math;
```

This will look for Apama Plug-in file `libapama_math.so` (on Solaris or Linux) or for `apama_math.dll` (on Windows). These must be located on the standard library path (in `LD_LIBRARY_PATH` in Unix, and in the `bin` folder on Windows). 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 in C or C++ 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 types and sequences (which map to native arrays in the plug-in) are passed by-reference. In addition, the chunk data type can be used to 'pass-through' data returned from one function call to another plug-in function, as shown below.

Using Correlator Plug-ins in EPL

About the chunk data type

The chunk data 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 data 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 complex.test4() method prints output to stdout. The source code for complex_plugin is in the samples\correlator_plugin\cpp directory of your Apama installation directory.

```
monitor ComplexPluginTest {

   // Load the plugin
  import "complex_plugin" as complex;

   // Opaque chunk value
   chunk myChunk;

   action onload() {
      // Generate a new chunk
      myChunk := complex.test3(20);

      // Do some computation on the chunk
      complex.test4(myChunk);
   }
}
```

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 data 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.

Using Correlator Plug-ins in EPL

10 Making Application Data Available to Clients

- Adding the DataView service bundle to your project 288
- Creating DataView definitions 289
- Deleting DataView definitions 291
- Creating DataView items 292
- Deleting DataView Items 294
- Updating DataView Items 296
- Looking Up Field Positions 299
- Using multiple correlators 300

Apama provides the `DataViewService`, which enables EPL (or Java) application writers to expose a view onto some of their data for easy consumption by remote client applications, such as Dashboard Builder dashboards.

The service uses two central concepts:

- DataView definition
- DataView item

A DataView definition specifies a unique DataView name, a set of field names and field types (each type is one of `string`, `float`, `integer`, and `boolean`), and optionally a set of key fields.

Each DataView item is associated with a DataView definition, and specifies values for the defined fields.

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 `DataViewService_Interface.mon` in the `monitors` directory of your Apama installation directory. It defines the API for working with DataView definitions and DataView items.

You can create DataViews in only the main context. You cannot create them in any contexts you create.

You can also use the MemoryStore to create DataViews and you can do this in any context. See "Exposing in-memory or persistent data to dashboards" on page 264.

Adding the DataView service bundle to your project

To use the `DataViewService`, add the `DataView Service` bundle to your Apama project:

1. In Apama Studio, to add a bundle to your project, open the project in the Apama Developer perspective.
2. In the Developer Project Explorer View, right-click the project name and select Apama > Add Bundle from the menu that appears.
3. Select the DataView Service bundle and click OK.

Adding the `DataViewService` bundle to your project 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`

The `DataViewService` is designed primarily to interact with other EPL or Java applications that reside in the same correlator, but see "Using multiple correlators" on page 300.

Making Application Data Available to Clients

Creating DataView definitions

Use the following event types in order to create DataView definitions:

- "DataViewAddDefinition" on page 289
- "DataViewDefinition" on page 290
- "DataViewException" on page 290

Making Application Data Available to Clients

DataViewAddDefinition

Syntax

```
event DataViewAddDefinition {
   string msgId;
   string dvName;
   string dvDisplayName;
   string dvDescription;
   sequence<string> fieldNames;
   sequence<string> fieldTypes;
   sequence<string> keyFields;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order to create a DataView definition. The response is provided by a "DataViewDefinition" on page 290 or "DataViewException" on page 290 event.

Syntax description

msgId is an optional field for a message ID that applications can use to identify responses.

dvName is a unique name for this DataView definition (for example, `"DataView_Weather001"`).

dvDisplayName is the display name for this DataView definition (for example, `"Weather Manager"`).

dvDescription is an optional field whose value is a description of this DataView definition (for example, `"This DataView exposes temperature, humidity, and visibility data for a given location."`).

fieldNames includes the names of the fields that contain the data being exposed.

fieldTypes includes the names of the types of the fields that contain the data being exposed. Each type is one of `string`, `float`, `integer`, and `boolean`.

keyFields Names of field or fields whose values in a DataView item are to be combined to make a unique key that can be used instead of the *dvItemId* field of `DataViewDeleteItem`, `DataViewUpdateItem`, and `DataViewUpdateDelta` events.

extraParams is an optional field that future implementations may use.

Creating DataView definitions

DataViewDefinition

Syntax

```
event DataViewDefinition {
   string msgId;
   string dvName;
   string dvDisplayName;
   string dvDescription;
   sequence<string> fieldNames;
   sequence<string> fieldTypes;
   sequence<string> keyFields;
   dictionary<string, string> extraParams;
}
```

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`* (an optional field that future implementations may use).

Creating DataView definitions

DataViewException

Syntax

```
event DataViewException {
   string msgId;
   string dvName;
   wildcard string message;
   dictionary<string, string> extraParams;
}
```

These events occur under exceptional circumstances in response to `DataViewAddDefinition` or `DataViewDeleteDefinition` events, or any circumstance under which a DataView cannot be identified.

Syntax description

`msgId` is the ID of the message to which this is a response.

`dvName` is the unique name of the DataView specified in the event to which this is a response.

`message` is the message in this exception. This is designed to be human readable, and may change between implementations or versions (hence the `wildcard` specification).

`extraParams` is an optional field that future implementations may use.

Creating DataView definitions

Example

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;
DataViewException dvException;
on all DataViewException(): dvException {
log "*** Weather monitor error: " +
   dvException.toString() at ERROR;
}
```

Creating DataView definitions

Deleting DataView definitions

Use the following event types in order to delete DataView definitions:

- "DataViewDeleteDefinition" on page 291
- "DataViewDefinitionDeleted" on page 291

Making Application Data Available to Clients

DataViewDeleteDefinition

Syntax

```
event DataViewDeleteDefinition {
   string msgId;
   string dvName;
   dictionary<string, string> extraParams;
}
```

Create and route events of this type in order to delete a DataView definition. The response is provided by a "DataViewDefinitionDeleted" on page 291 or "DataViewException" on page 290 event.

Syntax description

msgId is an optional field for a message ID that applications can use to identify responses.

dvName is the unique name of the DataViewDefinition to be deleted.

extraParams is an optional field that future implementations may use.

Deleting DataView definitions

DataViewDefinitionDeleted

Syntax

```
event DataViewDefinitionDeleted {
   string msgId;
   string dvName;
   dictionary<string, string> extraParams;
}
```

These events are responses to `DataViewDeleteDefinition` events. They indicate the successful deletion of a DataView definition.

Syntax description

`msgId` is the ID of the message to which this is a response.

`dvName` is the unique name of the DataView specified in the event to which this is a response.

`extraParams` is an optional field that future implementations may use.

Deleting DataView definitions

Creating DataView items

Use the following event types in order to create DataView items:

- "DataViewAddItem" on page 292
- "DataViewItem" on page 293
- "DataViewException" on page 290

Making Application Data Available to Clients

DataViewAddItem

Syntax

```
event DataViewAddItem {
   string msgId;
   string dvName;
   string owner;
   float timeStamp;
   sequence<string> fieldValues;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order to create a DataView item. A response is provided by a "DataViewItem" on page 293 or "DataViewException" on page 290 event.

Syntax description

`msgId` is an optional field for a message ID that applications can use to identify responses.

`dvName` is the unique name of this item's associated DataView definition.

`owner` is the owner (user) of the item. Specify "`*`" to allow all users to access the new DataView item.

`timeStamp` is the timestamp of the initial update (seconds since epoch). If the value given is `-1.0`, the service will populate it using correlator `currentTime`. Note that the default value of the field is `0.0`, which prevents trend graphs from updating.

`fieldValues` is a `sequence` of field values in string form. The *`i`*th value in the `sequence` is the value of the *`i`*th field specified in the *`fieldNames`*`equence` of the associated DataView definition.

`extraParams` is an optional field that future implementations may use.

Creating DataView items

DataViewItem

Syntax

```
event DataViewItem {
   string msgId;
   string dvName;
   integer dvItemId;
   string owner;
   sequence<string> fieldValues;
   dictionary<string, string> extraParams;
}
```

These events are responses to `DataViewAddItem` events. They indicate the successful creation of a DataView item. The contents of the fields are exactly those of the `DataViewAddItem` event to which this is a response, except possibly *`extraParams`*, and with the addition of the *`dvItemId`* field.

Syntax description

`dvItemID` is the ID of the Item within the DataView. This ID distinguishes the DataView item from all other DataView items that are associated with the same DataView definition. A different DataView item with a different definition might have the same *`dvItemID`*.

`extraParams` is an optional field that future implementations may use.

Creating DataView items

DataViewItemException

Syntax

```
event DataViewItemException {
   string msgId;
   string dvName;
   integer dvItemId;
   wildcard string message;
   dictionary<string, string> extraParams;
}
```

These events occur under exceptional circumstances in response to `DataViewDeleteItem`, `DataViewUpdateItem`, or `DataViewUpdateItemDelta` events.

Creating DataView items

Example

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;
```

```
...
DataViewAddItem item := new DataViewAddItem;
item.dvName := "Weather";
item.fieldValues :=
   [location,temp.toString(),humidity.toString(),
   visibility.toString()];
route item;
DataViewItem added;
on DataViewItem (dvName="Weather"):added {
   log("Weather monitor - DataViewItem: " +
      added.dvItemId.toString());
}
```

Creating DataView items

Deleting DataView Items

Use the following event types in order to delete DataView items:

- "DataViewDeleteItem" on page 294
- "DataViewItemDeleted" on page 295
- "DataViewDeleteAllItems" on page 296
- "DataViewAllItemsDeleted" on page 296
- "DataViewException" on page 290
- "DataViewItemException" on page 293

Making Application Data Available to Clients

DataViewDeleteItem

Syntax

```
event DataViewDeleteItem {
   string msgId;
   string dvName;
   integer dvItemId;
   sequence<string> keyFields;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order to delete a DataView item. A response is provided by a "DataViewItemDeleted" on page 295, "DataViewException" on page 290 or "DataViewItemException" on page 293 event.

Syntax description

msgId is an optional field for a message ID that applications can use to identify responses.

dvName is the unique name of the DataView definition associated with the item to be deleted.

dvItemId is the service-generated ID (see "DataViewItem" on page 293) of the item to be deleted, or `-1` if the `keyFields` field is specified instead.

keyFields is an optional field that specifies the item to be deleted by specifying a `sequence` of one or more key values. The *i*th value in the `sequence` is the value of the *i*th field in the `keyFieldssequence` of DataView definition specified by `dvName`.

extraParams is an optional field that future implementations may use.

Deleting DataView Items

DataViewItemDeleted

Syntax

```
event DataViewItemDeleted {
   string msgId;
   string dvName;
   integer dvItemId;
   sequence<string> keyFields;
   dictionary<string, string> extraParams;
}
```

These events are responses to `DataViewDeleteItem` events. They indicate the successful deletion of a DataView item.

Syntax description

msgId is the ID of the message to which this is a response.

dvName is the unique name of the DataView specified in the event to which this is a response.

dvItemId is the service-generated ID (see "DataViewItem" on page 293) of the deleted item, or `-1` if the `keyFields` field was specified instead.

keyFields is an optional field containing the key values in the event to which this is a response, if specified.

extraParams is an optional field that future implementations may use.

Deleting DataView Items

Example

Here is an example that creates and routes a `DataViewDeleteItem` event and handles the `DataViewItemDeleted` response by logging the deletion of the item:

```
using com.apama.dataview.DataViewDeleteItem;
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;
DataViewItemDeleted deleted;
on DataViewItemDeleted (dvName="Weather"):deleted {
   log("Weather monitor - DataViewItemDeleted:
      "+deleted.dvItemId.toString());
}
```

Deleting DataView Items

DataViewDeleteAllItems

Syntax

```
event DataViewDeleteAllItems {
   string msgId;
   string dvName;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order to delete all DataView items associated with a specified DataView definition. A response is provided by a "DataViewAllItemsDeleted" on page 296, "DataViewException" on page 290, or "DataViewItemException" on page 293 evens.

Syntax description

msgId is an optional field for a message ID that applications can use to identify responses.

dvName is the unique name of the DataView definition all of whose associated items are to be deleted.

Deleting DataView Items

DataViewAllItemsDeleted

Syntax

```
event DataViewAllItemsDeleted {
   string msgId;
   string dvName;
   dictionary<string, string> extraParams;
}
```

These events are responses to `DataViewDeleteAllItem` events. They indicate the successful deletion of all items associated with a given DataView definition.

Syntax description

msgId is an optional field containing the ID, if specified, of the message to which this is a response.

dvName is the unique name of the DataView specified in the event to which this is a response.

extraParams is an optional field that future implementations may use.

Deleting DataView Items

Updating DataView Items

Use the following event types in order to update DataView definitions:

- "DataViewUpdateItem" on page 297
- "DataViewUpdateItemDelta" on page 298
- "DataViewItemException" on page 293

Making Application Data Available to Clients

DataViewUpdateItem

Syntax

```
event DataViewUpdateItem {
   string msgId;
   string dvName;
   integer dvItemId;
   float timeStamp;
   sequence<string> fieldValues;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order to update a data item by specifying a `sequence` of new filed values. If the update does not succeed, a response is provided by a "DataViewItemException" on page 293 event.

Syntax description

`msgId` is an optional field for a message ID that applications can use to identify responses.

`dvName` is the unique name of this item's associated DataView definition.

`dvItemId` is the service-generated ID (see "DataViewItem" on page 293) of the item to be deleted, or `-1` if the `keyFields` field is specified instead.

`timeStamp` is the timestamp of the update (seconds since epoch). If the value given is `-1.0`, the service will populate it using correlator `currentTime`. Note that the default value of the field is `0.0`, which prevents trend graphs from updating.

`fieldValues` is a `sequence` of new field values in string form. The *`i`*th value in the `sequence` is established as the value of the *`i`*th field specified in the `fieldNamesequence` of the associated DataView definition.

`extraParams` is an optional field that future implementations may use.

Updating DataView Items

Example

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;
```

Updating DataView Items

DataViewUpdateItemDelta

Syntax

```
event DataViewUpdateItemDelta {
   string msgId;
   string dvName;
   integer dvItemId;
   float timeStamp;
   dictionary<integer,string> fieldValues;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order 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" on page 293 event.

Syntax description

msgId is an optional field for a message ID that applications can use to identify responses.

dvName is the unique name of this item's associated DataView definition.

dvItemId is the service-generated ID (see "DataViewItem" on page 293) of the item to be deleted, or `-1` if you supply all key field values in `fieldValues` (see below).

timeStamp is the timestamp of the update (seconds since epoch). If the value given is `-1.0`, the service will populate it using correlator `currentTime`. Note that the default value of the field is `0.0`, which prevents trend graphs from updating.

fieldValues is a dictionary of field-position/field-value pairs that specifies the new field values. Field values are in string form. A field's position is its index into the `fieldNamessequence` specified in the associated DataView definition (see also "Looking Up Field Positions" on page 299). If `dvItemId` is `-1`, `fieldValues` must include the key values.

extraParams is an optional field that future implementations may use.

Updating DataView Items

Example

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;
```

Updating DataView Items

Looking Up Field Positions

Use the following event types in order to lookup the numerical position of a given field-name for a given DataView definition:

- "DataViewGetFieldLookup" on page 299
- "DataViewFieldLookup" on page 299
- "DataViewException" on page 290

Making Application Data Available to Clients

DataViewGetFieldLookup

Syntax

```
event DataViewGetFieldLookup {
   string msgId;
   string dvName;
   dictionary<string, string> extraParams;
}
```

Create and route an event of this type in order to request a helper dictionary that supports lookup of field position for a given field name. The response is provided by a "DataViewFieldLookup" on page 299 or "DataViewException" on page 290 event.

Syntax description

msgId is an optional field for a message ID that applications can use to identify responses.

dvName is the unique name for the DataView definition whose field positions you want to look up.

extraParams is an optional field that future implementations may use.

Looking Up Field Positions

DataViewFieldLookup

Syntax

```
event DataViewFieldLookup {
   string msgId;
   string dvName;
   dictionary <string, integer> fields;
   dictionary<string, string> extraParams;
}
```

These events are responses to `DataViewGetFieldLookup` events. They contain a dictionary that supports lookup of the field position for a given field name.

Syntax description

msgId is an optional field containing the ID, if specified, of the message to which this is a response.

dvName is the unique name of the DataView specified in the event to which this is a response.

fields is a dictionary of field-name/field-position pairs. A field's position is its index into the *fieldNames*sequence specified in the associated DataView definition.

extraParams is an optional field that future implementations may use.

Looking Up Field Positions

Using multiple correlators

The DataView Service is designed to primarily interact with other EPL or Apama Java applications that reside in the same correlator. Therefore, the DataView Service implementation does not emit any events. You can inject the following optional additional monitors in order to emit the events when that is required:

- DataViewService_ServiceEmitter.mon
- DataViewService_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 DataViewService and ScenarioService to which any client dashboard can connect.

With this architecture, the individual applications in the separate correlators need to emit DataView Service request events to a channel that has been connected to the view correlator. These applications can either emit the events directly, or with the ApplicationEmitter injected they can route the events and the extra monitor will emit them to the channel. The DataViewService in the view correlator routes its responses (as normal), but the ServiceEmitter 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 Apama Studio that provides these monitors.

Making Application Data Available to Clients

11 Testing and Tuning EPL

- Optimizing EPL programs 302
- Best practices for writing EPL 302
- Structure of a basic test framework 304
- Using event files 304
- Handling runtime errors 305
- Capturing test output 307
- Avoiding listeners and monitor instances that never terminate 307
- Handling slow or blocked receivers 308
- Diagnosing infinite loops in the correlator 309
- Tuning contexts 309

This section provides information about testing and tuning your EPL applications.

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 unecessarily. See "Avoiding unnecessary allocations" on page 303.
- Specify `wildcard` on non-essential event fields. See "Wildcard fields that are not relevant" on page 303.
- Use plug-ins when you cannot write efficient EPL to accomplish your purpose. See "When to use plug-ins" on page 232.
- 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.

Testing and Tuning EPL

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 .. then .. 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.

Testing and Tuning EPL

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 69.

Best practices for writing EPL

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;
}
```

Best practices for writing EPL

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,*):p and not InAuction() {
```

```
      on Price(stock,>p.price*1.01) and not InAuction() {
         sellStock();
      }
   }
   on InAuction() inAuction();
}
```

Best practices for writing EPL

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 Studio project and then use Apama Studio 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.

Testing and Tuning EPL

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 enqueued between multiple contexts, or when plug-ins are used.

Testing and Tuning EPL

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.
- No concept of null, so there are no null pointer exceptions.

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 Catching exceptions .

Testing and Tuning EPL

What happens

When the correlator detects a runtime error, it 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 a 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.

Handling runtime errors

Using ondie() to diagnose runtime errors

You cannot catch and handle runtime errors like you can handle exceptions in other languages. You cannot prevent the correlator from terminating the monitor instance. However, 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 "sub-monitor 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 emits 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.

Handling runtime errors

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 *Deploying and Managing Apama Applications*, "Event Correlator Utilities Reference", "Setting logging attributes for packages, monitors, and events". 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`.

Another option is to use the Log File Manager plug-in. See "Using the Log File Manager plug-in" on page 248.

See also "Logging and printing" on page 178.

Handling runtime errors

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. For example:

```
2009-01-10 18:59:39.665 INFO  [12] - Status: sm=314 nctx=315 ls=313 rq=0
   eq=0 iq=1 oq=0 rx=0 tx=0 rt=75 nc=0 vm=1089840
```

These values have the following meanings:

- `sm` — number of monitor instances across all contexts.
- `nctx` — number of contexts, including the main context.

- `ls` — sum of the number of listeners in all contexts.
- `rq` — sum of the number of routed events on all contexts' input queues.
- `eq` — number of events on the enqueued events queue. A separate thread moves events from this queue to the input queue of each public context. Events that are enqueued with `enqueue...to` are not on the enqueued events queue and so are not included in this count.
- `iq` — sum of the number of entries on the input queues of all contexts.
- `oq` — number of events on output queue.
- `rx` — number of events received.
- `tx` — number of events transmitted.
- `rt` — number of events routed since the correlator was started.
- `nc` — number of receivers.
- `vm` — number of kilobytes of virtual memory being used by the correlator process.

You can use this information to diagnose common problems.

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.

Handling runtime errors

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.

Testing and Tuning EPL

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():t spawn handle();         // missing "unmatched" action
...
action handle() {
   on all Trade(symbol=t.symbol):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.

Testing and Tuning EPL

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 bad 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 bad receiver.

Bad receivers include:

- Slow 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 bad. 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 *Deploying and Managing Apama Applications*, "Event Correlator Utilities Reference", "Starting the event correlator", "Determining whether to disconnect slow receivers".

Testing and Tuning EPL

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.

Testing and Tuning EPL

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" on page 310
- "Parallel processing for long-running calculations" on page 311

Testing and Tuning EPL

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 enqueue 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 {
      Tick t;
      on all Tick():t {
         // If the context for this symbol does not yet exist, create it.
         if(not symbolMapping.hasKey(t.symbol)) then {
            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.
         enqueue 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) {
      Tick t;
      // 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():t {
         ...
      }
```

```
   }
}
```

Tuning contexts

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 {
   action onload() {
      on all Tick() {
         numTicks:=numTicks+1;
         emit NumberTicks(numTicks);
      }
      Calculate calc;
      on all Calculate():calc {
         integer atNumTicks:=numTicks;
         integer calcId:=integer.getUnique();
         spawn doCalculation(calc, calcId, context.current())
            to context("Calculation");
         CalculationResponse response;
         on CalculationResponse(calcId):resp {
            emit CalculationResult(resp, atNumTicks, numTicks);
         }
      }
   }
   action doCalculation(Calculate req, integer id, context caller) {
      float value:=actual_calculation_function(req);
      enqueue 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 enqueue a `CalculationResponse` event for 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 enqueued `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.

Tuning contexts

12 Generating Documentation for Your EPL Code

- Code constructs that are documented 314
- Steps for using ApamaDoc 314
- Inserting ApamaDoc comments 315
- Inserting ApamaDoc tags 316
- Inserting ApamaDoc references 318
- Generating ApamaDoc in headless mode 319

Just as you can use the Javadoc tool to generate documentation for Java, you can use the ApamaDoc tool 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 Apama Studio. 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 in headless mode by invoking the `apamadoc` utility from the command line.

Code constructs that are documented

ApamaDoc generates documentation for the following code constructs:

- Packages
- Events (defined outside monitors)
- Monitors
- Inner events (defined inside monitors)
- Plug-in imports
- Members — variables and constants in monitors, fields in events
- Custom aggregate functions
- Wildcard modifiers
- Actions, including the following:
  - Parameters
  - Return types
  - Events routed
  - Events enqueued
  - Events emitted
  - Events being listened for
  - Actions spawned — actions that are the target of `spawn` statements

Generating Documentation for Your EPL Code

Steps for using ApamaDoc

The general steps for using ApamaDoc are as follows:

1. In Apama Studio, create a project.
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 315, "Inserting ApamaDoc tags" on page 316, and "Inserting ApamaDoc references" on page 318.
4. Save and build the project.
5. Right-click the project name and select Export... from the popup context menu.
6. In the Export dialog, expand Apama, 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.

Generating Documentation for Your EPL Code

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 316. 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);
    Configuration c;
    listener l:=on Configuration(scenarioId=scenarioId):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

Generating Documentation for Your EPL Code

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` *name*	There are no restrictions on the name. It can span multiple lines. It is ended by the start of the next tag.	Imports, events, monitors, aggregate functions
`@deprecated [` *description*`]`	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.	Imports, events, monitors, actions, and members (variables and named constants)
`@emits` *eventRef* `[` *description* `]`	Documents events that are emitted. *eventRef* 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 318.	Actions and their return types
`@enqueues` *eventRef* `[` *description* `]`	Documents events that are enqueued. *eventRef* specifies a link to an event definition. The optional description can be anything pertinent to the enqueueing of the event. See "Inserting ApamaDoc references" on page 318.	Actions and their return types
`@listens` *eventRef* `[` *description* `]`	Dccuments events that are being listened for. *eventRef* specifies a link to an event definition. The optional description can be anything pertinent to	Actions and their return types

Tag	Description	Use For
	the event listener. See "Inserting ApamaDoc references" on page 318.	
`@param codeRef [ description ]`	Documents arguments to actions and custom aggregate functions. *codeRef* specifies a link to a variable. The optional description can be anything pertinent to the parameter. See "Inserting ApamaDoc references" on page 318.	Actions and custom aggregate functions
`@private`	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: `/** @private /*`	All code constructs except packages and action contents
`@return codeRef [ description ]`	Documents return values from actions and aggregate functions. *codeRef* specifies a link to a variable. The optional description can be anything pertinent to the return value.	Actions and aggregate functions
`@routes eventRef [ description ]`	Documents events that are being routed. *eventRef* 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 318.	Actions and their return types

`@see`	There are three forms of this tag. Each form documents a relationship between a code fragment and some other information.
	`@see "description"`
	Lets you insert text that explains the relationship.
	`@see codeRef description`
	Lets you reference an EPL code construct and describe the relationship between this construct and that construct. *codeRef* specifies a link to some other EPL code. See "Inserting ApamaDoc references" on page 318.
	`@see <a href="url">linkText</a> [description]`
	Lets you specify an HTML link to an external resource. Optionally, you can add more information.
		All code constructs except packages and action contents

`@since version`	Documents when a code construct was introduced. Replace *version* with a particular version number, for example, `5.1`.	All code constructs except packages and action contents

Tag	Description	Use For
`@spawns actionRef [ description ]`	Lets you document the lifecycle of a monitor. *actionRef* specifies a link to an action definition. See "Inserting ApamaDoc references" on page 318.	Monitors and actions
`@version version`	Lets you specify a version of the current incarnation of this code. Replace *version* with a particular version number, for example, `5.1`.	Monitor definitions, event type definitions, custom aggregate function definitions, and actions

Generating Documentation for Your EPL Code

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.

Generating Documentation for Your EPL Code

Generating ApamaDoc in headless mode

Headless mode lets you generate ApamaDoc from a command line as a standalone operation on Windows platforms. This is useful if you want to control what ApamaDoc generates without user-interface intervention, for example, when you are running nightly build integrations. Also, from the command line, you can control which files are exported and which files are omitted.

To generate ApamaDoc in headless mode, run the `apamadoc.bat` script, which is in the `%APAMA_HOME%\bin` folder. The `apamadoc.bat` file uses the `%APAMA_HOME%\utilities\apamadoc.xml` ant script to generate ApamaDoc.

The format for invoking the `apamadoc` utility is as follows:

```
apamadoc [-v] output_folder
   monitor_file_base_folder|monitor_file_path|file_path ...
```

`-v`	Optional. Displays verbose output on `stdout` about the process that is generating ApamaDoc.
output_folder	Identifies the folder that will contain the generated ApamaDoc.
monitor_file_base_folder	Specifies a folder that contains EPL `.mon` files for which you want to generate ApamaDoc. You can specify zero, one, or more folders. Insert a space between names. The `apamadoc` utility recursively processes specified folders.
monitor_file_path	Specifies an EPL `.mon` file for which you want to generate ApamaDoc. You can specify zero, one, or more `.mon` files. Insert a space between names.
file_path	File that lists the EPL source files for which you want to generate ApamaDoc. Specify this file by prepending an @

	sign at the beginning of the path, for example, `@C:\docfiles\inputEPLFilePaths.txt`. In the specified file, specify one source file on each line. You cannot specify `@location` in the specified file; that is, this facility is not recursive.
`-h`	Optional. Specify `-h` to display usage information.

Note: The `apamadoc` utility requires Apache Ant. If you are not running the `apamadoc` utility from an Apama command prompt, you must ensure that the `PATH` variable for the headless ApamaDoc command line contains an entry for the Ant installation `bin` folder, for example, `C:\ant_1.7.6\bin`. This makes the Apama `ant.bat` file accessible to ApamaDoc generation.

On the command line, you can mix file paths, monitor file paths, and folder paths in any combination. The following example generates ApamaDoc for all monitor files in the `C:\mon_files_dir` folder as well as for the `C:\Apama_monfiles/MyMonitor.mon` file and all the files listed in the `inputEPLFilePaths.txt` file. The `mon_files_dir` folder is processed recursively. The generated ApamaDoc is put into the `C:\generated_apamadocs` folder.

```
apamadoc C:\generated_apamadocs
   C:\mon_files_dir
   C:\Apama_monfiles\MyMonitor.mon
   @C:\docfiles\inputEPLFilePaths.txt
```

The next example generates ApamaDoc and puts it in the `C:\MyApplication\ApamaDoc` folder. Specification of the `-v` option displays names of the files being processed on the command line. The files being processed are listed in the `EPLsource.txt` file.

```
apamadoc -v C:\MyApplication\ApamaDoc @C:\MyApplication\doc\EPLsource.txt
```

Headless mode for generating ApamaDoc is available on only Windows platforms.

Generating Documentation for Your EPL Code

A EPL Naming Conventions

It is recommended that you use the following naming conventions in EPL. These conventions closely follow Sun's 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 `iafInterface` for an attribute or `IafInterface` for a monitor are easier to read than `iAFInterface` and `IAFInterface`.

Actions	lowerCamelCase	Actions should be verbs, in mixed case with the first letter lowercase, and the first letter of each internal word capitalized. For example:
		`handleQuery();`
		`startDaemonProcess();`
		`quit();`

Constants	ALL_CAPITALS	Identifiers for constants should be all uppercase with words separated by underscores. For example:
		`constant integer MAX_SIZE;`
		`constant string DEFAULT_HOST;`

Contexts	UpperCamelCase	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:
		`context("Calculation");`
		`context("Inventory", true);`

Custom aggregate functions	lowerCamelCase	Custom aggregate funcitons should be in mixed case with the first letter lowercase, and the first letter of each internal word capitalized. `aggregate bounded myCustomAggregate() returns integer { aggregateBody }`

Events	UpperCamelCase	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:
		`event Tick`
		`event SubscriptionConfiguration`
		`event IafEvent`

Monitors	UpperCamelCase	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:

Item	Convention	Notes and Examples
		`monitor SubscriptionManager` `monitor IafMonitorService`
Packages	lowercase	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. 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: `com.apamax.accounting`

Variables	lowerCamelCase	Variables and parameters should have initial lowercase. This is left to your discretion, but lowercase is preferable. Internal words start with capital letters.
		
		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`, `j`, `k`, `m`, and `n` for integers.
		
		`integer i;`
		`float myPrice;`
		`MyEvent myEvent;`

B EPL Keyword Quick Reference

EPL is case-sensitive.

There are a number of identifiers that EPL has reserved for future use. In this release, if you use a reserved identifier, the correlator logs a warning. For a list of reserved identifiers, see "Keywords" in the "Lexical Elements" section of the *Apama EPL Reference*.

The following table describes EPL keywords and special identifiers. Some keywords are reserved only in the scope of a query. Where applicable, this is noted in the description. You can use an EPL keyword as an identifier if you prefix it with a hash symbol (#). See "Keywords" in the "Lexical Elements" section of the *Apama EPL Reference*.

Keyword	Description	Syntax and Example

`action`	References or declares an action. Required in each action declaration.
	
	Also an EPL data type.
		`action action_name([arglist])returns retType{ do_something>; }`
		
		`action notifyUser(){ log "Event sequence detected."; }`

`aggregate`	Keyword required in the definition of a custom aggregate function that can be used in a stream query.	`aggregate [bounded|unbounded] aggregateName ( [arglist]) returns retType { aggregateBody }`
		
		`aggregate bounded wstddev( decimal x, decimal w ) returns decimal { do something}`

`all`	Appears just before an event template to indicate that you want to continue listening for all instances of the specified event, and not just the first matching event.	`all event_template`
		
		`on all StockTick(*,*):newTick processTick();`

	Appears just before an event template that uses no other operators and creates a stream rather than an event listener. This is a stream source template, which continuously listens for all instances of the specified event and inserts all matching	`all event_template_with_no_other_operators`
		
		`stream<Tick> ticks := all Tick(symbol="APMA");`

Keyword	Description	Syntax and Example
	events into a newly created stream.	
	See " retain all" on page 332.	

and	Logical operator in an event expression.	`on event_template and event_template action;`
		
		`on A() and B() executeAction();`

	Logical operator in an `if` statement or other Boolean expression.	`if ordinary_exp and ordinary_exp then block;`
		
		`if x and y then {myBlock;}`

as	Specified to import a correlator plug-in.	`import plug-in as identifier;`
		
		`import TimeFormatPlugin as foo;`

at	Temporal operator in event expressions. Triggers a timer at a specific time or at repeated intervals.	`at(minutes, hours, days_of_month, months, days_of_week [,seconds])`
		
		`on all at(5, 9, *, *, *) success;`

	Identifies the log level in a `log` statement.	`log string [at log_level];`
		
		`log "Your message here" at INFO;`

boolean	Boolean data type. Value is `true` or `false`.	`boolean identifier;`
		
		`boolean marketOpen;`

bounded	Optional keyword in a custom aggregate function definition. Indicates a function that can be used only with a bounded stream query window.	See `aggregate`.
break	In a `for` or `while` statement, transfers control to the next statement following the block that encloses the `break` statement.	`break;`
by	Part of a `partition by` or `group by` clause in a stream query.	See "group by" on page 327. See "partition by" on page 331.

Keyword	Description	Syntax and Example
	Valid as an identifier outside a stream query.	
`catch`	Part of a `try...catch` statement for handling exceptions.	See `try`.

`chunk`	Data type. References a dynamically allocated opaque object whose contents cannot be seen or directly manipulated in EPL. Typically used to manage plug-in data.	`chunk identifier;`
		
		`chunk complexProductInfo;`

`completed`	Event expression that matches only after all other processing on the matching event is completed.	`on all completed event_expression action;`
		
		`on all completed A(f < 10.0) {}`

`constant`	Specifies an unchanging literal value.	`constant type name := literal;`
		
		`constant float GOLDEN := 1.61803398874;`

`context`	Data type. Enables parallel processing.	`context(string name)`
		`context(string name, boolean receivesInput)`
		
		`context c:=context("test");`

`continue`	In a `for` or `while` statement, ends execution of the current iteration and transfers control to the beginning of the loop.	`continue;`
`currentTime`	Special EPL variable that returns the current time in the correlator.	`log currentTime.toString();` `emit TestEvent(currentTime);`

`decimal`	Data type. Signed floating point decimal number with `d` at the end to distinguish it from a `float` data type.	`decimal identifier;`
		
		`decimal exactValue;`
		`exactValue := 1.2345d;`

`dictionary`	Data type. Stores and retrieves data based on a key.	`dictionary <key_type, data_type> identifier;`
		
		`dictionary <integer, string> myOrders;`

Keyword	Description	Syntax and Example

`die`	Terminates execution of the monitor instance.	`die;`
		
		`on NewStock (chosenStock.name,`
		`  chosenStock.owner) die;`

`else`	Part of an `if` statement.	See example of `if`. statement.

`emit`	Publishes an event on the correlator's output queue.	`emit event;`
		
		`emit newEvent;`

`emit...to`	To publish an event to a named channel of the correlator's output queue, specify `to channel`.	`emit event to channel;`
		
		`emit newEvent to "com.apamax.pricechanges";`

`enqueue`	Sends an event to the back of the input queue of each public context.	`enqueue event;`
		
		`enqueue newEvent;`

`enqueue ...to`	To send an event to the back of the input queue of a particular context specify `to context_expr`.	`enqueue event_expr to context_expr;`
		
		`enqueue tick to c;`

`event`	Declares an event type.
	
	Required in each event type definition.
		`event event_type {`
		`  [ [wildcard] field_type field_name; |`
		`  constant field_type field_name := literal; |`
		`action_definition ... ]`
		`}`
		
		`event StockTick {`
		`  string name;`
		`  float price;`
		`}`

`every`	In a stream query, if you specify a `within` window, specification of `every` updates the window every `batchPeriodExpr` seconds.
	
	The `every` keyword is valid as an identifier outside a stream query.
		`every batchPeriodExpr`
		
		`from v in values`
		`  within 3.0 every 3.0`
		`  select v`

	If you specify a `retain` window without also specifying `within`, specification of `every` updates the	`every batchSizeExpr`
		
		`from v in values`
		`  retain 3 every 3`
		`  select v`

Keyword	Description	Syntax and Example
	window after every *batchSizeExpr* items are received.	
false	Possible value of a Boolean variable.	
float	Data type. Signed floating point number.	float *identifier*; float squareRoot;

for	Iterates over the members of a sequence and executes the enclosing statment or block once for each member.	*forStatement* ::= for *counter*
		in *sequenceblock*;
		
		for i in s {
		print i.toString();
		}

from	Introduces a stream query definition. Specifies the stream, and optionally a window (stream subset), that the stream query is operating on.	from *itemIdentifier* in *streamExpr* [*windowDefinition*] from t in ticks retain 3
	Two consecutive from clauses specify a cross-join, which combines items from two streams to create one stream.	from *itemIdentifier* in *streamExpr* [*windowDefinition*] from *itemIdentifier* in *streamExpr* [*windowDefinition*] from x in letters retain 2 from y in numbers retain 2 select P(x,y)

	Specifies a stream listener that obtains items from a stream and passes them to procedural code.	[*listener* :=] from *streamExpr* :
		identifier statement
		
		float p;
		from t in all Tick(symbol="APMA")
		select t.price : p {
		print "'APMA' price is: " +
		p.toString();
		}

group by	Controls how a stream query groups data when generating aggregate output items. It is valid to use group as an identifier outside a stream query.	group by *groupByExpr* [, *groupByExpr*]... from t in ticks within 60.0 group by t.symbol select mean(t.price)
having	Filter the items coming out of a stream query's aggregate projection.	from t in all Temperature() within 60.0 having count() > 10 select mean(t.value)

Keyword	Description	Syntax and Example
	Valid as an identifier outside of a stream query.	

`if`	Conditionally executes a statement or block.	*ifStatement* ::=
		  if *booleanExpression* then *block*
		  | if *booleanExpression*
		  then *block* else *block*
		  | if *booleanExpression*
		  then *block* else *ifStatementblock*
		  ::= {
		statementList
		  }
		
		`if floatVariable > 5.0 then {`
		`  integerVariable := 1;`
		`} else if floatVariable < -5.0 then {`
		`  integerVariable := -1;`
		`} else {`
		`  integerVariable := 0;`
		`}`

`import`	Loads a plug-in into the correlator and makes it available to your monitor, event, or aggregate function.	`import "`*plug-in_name*`" as` *identifier*`;`
		
		`import "complex_plugin" as complex;`

`in`	Identifies range membership in an event expression.	`on` *event_name*`(`*event_field* `in [`*range*`])`
		
		`on all A(m in [0:10])`

	Part of `for` statement.	See `for` statement example.
	Part of `from` statement.	See `from` statement example.

`integer`	Data type. Negative, zero, and positive integers.	`integer` *identifier*`;`
		
		`integer count;`

`join`	Combines matching items from two streams to create one stream. This is an equi-join.
	
	Valid as an identifier outside a stream query.
		`join` *itemIdentifier* `in` *streamExpr*
		  `[`*windowDefinition*`]`
		  `on` *joinKeyExpr1* `equals` *joinKeyExpr2*
		
		`from r in priceRequest`
		`  join p in prices`
		`  partition by p.symbol retain 1`
		`  on r.symbol equals p.symbol`
		`  select p.price`

	Built-in method on strings that concatenates a `sequence` of strings.	`join(sequence<string>` *s*`)`
		
		`sequence<string> s :=`
		`  ["Something", "Completely", "Different"];`
		`  print ", ".join(s);`
		
		This prints the following:
		
		`"Something, Completely, Different"`

Keyword	Description	Syntax and Example
`largest`	Reserved for future use.	
`location`	Data type. An EPL data type used to describe rectangular areas in a two-dimensional, unitless, Cartesian, coordinate plane. Locations are defined by the `float` coordinates of two points `x1`, `y1` and `x2`, `y2` at diagonally opposite corners of an enclosing boundary rectangle.	`location(15.23, 24.234, 19.1232, 28.873)`

`log`	Writes messages and accompanying date and time information to the correlator's log file	`log` *`string`* `[at` *`log_level`*`];`
		
		`log "Your message here" at INFO;`

`monitor`	Declares a monitor. Required in each monitor definition. Braces enclose event type definitions, global variable declararions, and actions.	`monitor` *`monitor_name`* `{`
		`...`
		`}`
		
		`monitor SimpleShareSearch {`
		`...`
		`}`

`new`	Allocates a new object.	`new` *`typeName`*`;`
		
		`b := new Foo();`

`not`	Logical operator in an event expression.	`not` *`event_template`* `on A() and not B() executeAction();`

	Logical operator in an `if` statement or other Boolean expression.	`if not` *`ordinary_exp`* `then` *`block`*`;`
		
		`if not x then myBlock;`

`on`	Declares an event listener.	`on [all]` *`event_expression action`*`;`
		
		`on NewsItem("ACME",*) findStockChange();`

	Part of an equi-join clause.	See `join`.

`onBegin Recovery`	If defined, action that the correlator executes	`action onBeginRecovery() { }`
		
		`action onBeginRecovery() {`
		`  if (timeFormatPlugin.getTime() -`
		`  currentTime > (60.0 * 60.0 * 2)`

Keyword	Description	Syntax and Example

	when the correlator restarts.
	
	Note that `onBeginRecovery` is not a keyword. It is a special identifier. It is good practice to refrain from using this identifier for any other purpose.
		`then`
		`{ longDowntime:=true;`
		`... // do something if`
		`// downTime was long`
		`}`
		`}`

`onConclude Recovery`	If defined, action that the correlator executes when the correlator finishes recovery.
	
	Note that `onConcludeRecovery` is not a keyword. It is a special identifier. It is good practice to refrain from using this identifier for any other purpose.
		`action onConcludeRecovery() { }`
		
		`action onConcludeRecovery() {`
		`initiateListener(); // go back`
		`//to normal`
		`}`

`ondie`	If defined, action that the correlator executes when a monitor instance terminates.
	
	Note that `ondie` is not a keyword. It is a special identifier. It is good practice to refrain from using this identifier for any other purpose.
		`action ondie() { }`
		
		`action ondie() {`
		`log "sub-monitor terminating for "`
		`+ myId;`
		`route InternalError("Foo");`
		`}`

`onload`	Name of the action that the correlator executes when you inject a monitor. Every monitor must declare an `onload` action.
	
	Note that `onload` is not a keyword. It is a special identifier. It is good practice to refrain from using this
		`action onload(){ ... }`
		
		`action onload() {`
		`on all StockTick(*,*):newTick {`
		`processTick();`
		`}`
		`}`

Keyword	Description	Syntax and Example
	identifier for any other purpose.	

onunload	If defined, action that the correlator executes when the last instance of a particular monitor terminates.
	
	Note that onunload is not a keyword. It is a special identifier. It is good practice to refrain from using this identifier for any other purpose.
		action onunload() { };
		
		action onunload() {
		route LastMonitorTerminating();
		}

optional	Reserved for future use.	
or	Logical operator in an event expression.	on *event_template* or *event_template action*; on A() or B() executeAction();
	Logical operator in an if statement or other Boolean expression.	if *ordinary_exp* or *ordinary_exp* then *block*; if x or y then myBlock;
package	Mechanism for adding context to monitor and event names. Monitors and global events in the same package must each have a unique name within the package.	package *identifier*; package com.apamax.orders;
partition by	Effectively creates a separate window for each encountered distinct value of the partition by expression. partition is valid as an identifier outside a stream query.	partition by *partitionByExpr* [, *partitionByExpr*]... from t in all Tick() partition by t.symbol retain 10 with unique t.price select t.price
persistent	At the beginning of a monitor declaration, indicates you want	persistent monitor *string* persistent monitor ManageOrders

Keyword	Description	Syntax and Example
	that monitor to be persistent.	

`print`	Writes textual messages followed by a newline to the correlator's standard output stream — `stdout`.	`print string;`
		
		`print "Your message here.";`

`retain`	Specifies a stream query window that contains only the last *n* items received.
	
	Valid as an identifier outside a stream query.
		`retain windowSizeExpr`
		
		`from v in values`
		`  retain 10`
		`  select mean(v)`

`retain all`	Specifies a stream query window that aggregates values calculated over the lifetime of the query. This is an unbounded window.	`retain all`
		
		`from v in values`
		`  retain all`
		`  select mean(v)`

`return`	In an `action` body, specifies the value to return from that action. Required if an action returns a value.	`returns typeToReturn`
		`return retValue`
		
		`action complexAction(`
		`  integer i, float f) returns string {`
		`  // do something`
		`  return "Hello";`
		`}`

`returns`	In an `action` declaration, specifies the type of value returned by an action. Required if an action returns a value. Also used in custom aggregate function declarations and when naming `action` types.	See previous example.

`route`	Sends an event to the front of the current context's input queue.	`route event();`
		
		`route StockTick();`

`rstream`	In a query with a window definition and a simple projection,	`select [rstream] selectExpr`
		
		`from i in inputs`
		`  retain 2`

Keyword	Description	Syntax and Example
	indicates that you want the query to output its *remove* stream, that is, the items it removes from the window. Specification of `rstream` in an aggregate projection is not useful so it is not allowed. Valid as an identifier outside a stream query.	`select rstream i;`

`select`	Identifies the item(s) you want the query to output.
	This keyword is valid as an identifier outside a stream query.
		`select [rstream] selectExpr`
		
		`from v in values`
		`  retain 10`
		`  select mean(v);`

`sequence`	Data type. Ordered set or array of entries whose values are all of the same primitive or reference type.	`sequence<data_type> identifier;`
		
		`sequence<float> myPrices;`

`smallest`	Reserved for future use.	

`spawn`	Creates a copy of the currently executing monitor instance.	`spawn action([parameter_list]);`
		
		`action onload() {`
		`  spawn forward("a", "channelA");`
		`  spawn forward("b", "channelB");`
		`}`

`spawn...to`	To create a copy of the currently executing monitor instance in the specified context specify `spawn` with `to`*`context_expr`*.	`spawn action([arg_list]) to context_expr;`
		
		`spawn doCalc(cal) to context("Calculation");`

`static`	Reserved for future use.	

`stream`	Data type. Refers to a stream of items. An item can be a `boolean`, `decimal`, `float`, `integer`, `string`,	`stream<type> name;`
		
		`stream<decimal> prices;`

Keyword	Description	Syntax and Example
	`location`, or `event` type.	
`streamsource`	Reserved for future use.	

`string`	Data type. Text string.	`string identifier;`
		
		`string message;`

`then`	Part of conditional `if` statement.	See example for `if`.
`throw`	Reserved for future use.	
`to`	Indicates target of an `emit`, `enqueue`, or `spawn` operation.	See examples for `emit...to`, `enqueue...to`, and `spawn...to`.
`true`	Possible value of a Boolean variable.	

`try`	Part of a `try...catch` statement for handling exceptions.	`try block1`
		`  catch(Exception variable) block2`
		`try {`
		`  return float.parse(prices[fxPair]);`
		`  } catch(Exception e) {`
		`  return 1.0;`
		`  }`

`unbounded`	Optional keyword in a custom aggregate function definition. Indicates a function that can be used with only an unbounded (`retain all`) stream query window.	See `aggregate`.
`unique`	Part of the optional `with unique` clause in a stream query.	See `with unique`.

`unmatched`	Except for `completed` and `unmatched` event expressions, the event is not a match with any event expression currently within the context.	`on all unmatched event_expression[:coassignment] action;`
		
		`on all unmatched Tick():tick processTick();`

`using`	In a stream query, allows use of a custom	`using packageName.{aggregateName|eventName};`
		
		`using com.apamax.custom.myAggregateFunction;`

Keyword	Description	Syntax and Example
	aggregate function that is defined in another package.	

`wait`	Temporal operator in an event expression. Inserts a pause in an event expression. Once activated, a `wait` expression becomes true automatically once the specified amount of time passes.	`wait(float)`
		
		`on A() -> wait(10.0) -> C() success;`

`where`	Filter the items in the stream query's window or the items that result from a join operation.
	
	Valid as an identifier outside a stream query.
		`where` *`booleanExpr`*
		
		`from t in ticks`
		`  retain 100 where t.price*t.volume>threshold`
		`  select mean(t.price)`

`while`	Repeatedly evaluates a `boolean` expression and executes an enclosed statement or block as many times as the expression result is found to be `true`.	*`whileStatement`* `::= while` *`booleanExpression block`*
		
		`while integerVariable > 10 {`
		`  integerVariable := integerVariable - 1;`
		`  on StockTick(`
		`  "ACME", integerVariable) doAction(); }`

`wildcard`	In an event type definition, indicates a parameter that you will never specify as a match criteria in an event template.	`wildcard` *`param_typeparam_name`*`;`
		
		`event StockTick {`
		`  string name;`
		`  float price;`
		`  wildcard string exchange;`
		`}`

`with unique`	In a stream query, 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 part of the result set.
	
	`with` and `unique` are valid as identifiers outside a stream query.
		`with unique` *`keyExpr`*
		
		`from p in pairs`
		`  retain 3 with unique p.letter`
		`  select sum(p.number)`

Keyword	Description	Syntax and Example

`within`	Temporal operator in an event expression. Specifies a time limit for the event listener to be active.	`within(float)`
		
		`on A() -> B() within(30.0) notifyUser();`

	In a stream query, specifies a window that contains only those items received in the last *windowDurationExpr* seconds.	`within windowDurationExpr`
		
		`from v in values`
		`  within 20.0`
		`  select mean(v);`

`xor`	Logical exclusive `or` operator that can apply to an event template.	`xor event_template`
		
		`on A() xor B() notifyUser();`

	Logical operator in an `if` statement or other Boolean expression.	`if ordinary_exp xor ordinary_exp then block;`
		
		`if x xor y then myBlock;`

`#`	Escapes names of variables that clash with EPL keywords.	`#identifier`
		
		`print f.#integer.toString();`

C EPL Methods Quick Reference

- action methods 338
- boolean methods 338
- chunk methods 338
- context methods 338
- decimal and float methods 339
- dictionary methods 341
- event methods 342
- Exception methods 343
- integer methods 343
- listener methods 344
- location methods 344
- sequence methods 345
- StackTraceElement methods 345
- stream methods 346
- string methods 346
- built-in monitor methods 347

action methods

The only operation that you can peform 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 an example and additional details, see "Using action type variables" in *Developing Apama Applications in EPL*.

EPL Methods Quick Reference

boolean methods

Method	Result
`canParse(`*`string`*`)`	Returns a `boolean` true if the string argument can be successfully parsed.
`parse(`*`string`*`)`	Returns the `boolean` instance represented by the `string` argument.
`toString()`	Returns a `string` representation of the `boolean`.

EPL Methods Quick Reference

chunk methods

Method	Result
`clone()`	Returns a new `chunk` that is an exact copy of the `chunk` that `clone()` was called on.
`empty()`	Returns a `boolean` true if the chunk is empty.
`getOwner()`	Returns `string` that contains the name of the correlator plug-in that the chunk belongs to.

EPL Methods Quick Reference

context methods

Method	Result
`current()`	Returns a `context` object that is a reference to the current context.
`getId()`	Returns an `integer` that is the ID of the context.
`getName()`	Returns a `string` that is the name of the context.

Method	Result
`isPublic()`	Returns a `boolean` true if the context is public.
`toString()`	Returns a `string` that contains the properties of the context.

In addition, the `current()` static method returns a reference to the current `context`.

EPL Methods Quick Reference

decimal and float methods

Unless noted otherwise, if you call a method on a `decimal` type the return value is a `decimal`, and if you call the method on a `float` type, the return value is a `float`.

Method	Result
`abs()`	Returns the absolute value.
`acos()`	Returns the inverse cosine.
`acosh()`	Returns the inverse hyperbolic cosine.
`asin()`	Returns the inverse sine in radians.
`asinh()`	Returns the inverse hyperbolic sine.
`atan()`	Returns the inverse tangent.
`atan2(`*y*`)`	Returns the two-parameter inverse tangent.
`atanh()`	Returns the inverse hyperbolic tangent.
`bitEquals(`*decimal*`)` or `bitEquals(`*float*`)`	Returns a `boolean` true if the value it is called on and the value passed as an argument to the method are the same. The value the method is called on and the argument to the method must both be `decimal` types or must both be `float` types.
`canParse(`*string*`)`	Returns a `boolean` true if the argument can be successfully parsed.
`cbrt()`	Returns the cube root.
`ceil()`	Returns the smallest possible `integer` that is greater than or equal to the operand.
`cos()`	Returns the cosine.
`cosh()`	Returns the hyperbolic cosine.
`erf()`	Returns the error function

Method	Result
`exp()`	Returns `ex`, where *x* is the value of the `decimal` or `float` and `e` is approximately 2.71828183.
`exponent()`	When called on a `float` value, this method returns the exponent where *x* = `mantissa*2`$^{\texttt{exponent}}$ assuming `0.5 <= |mantissa| < 1.0`. When called on a `decimal` value, this method returns the exponent where *x*=`mantissa*10`$^{\texttt{exponent}}$ assuming `0.1 <= |mantissa| < 1.0`.
`floor()`	Returns the largest possible `integer` that is less than or equal to the operand.
`fmod(y)`	Returns the operand `mod` *y* in exact arithmetic.
`formatFixed(`*integer*`)`	Returns a `string` representation of the operand where the value is rounded to the number of decimal places specified in the argument.
`formatScientific(` *integer*`)`	Returns a `string` representation of the value the method is called on where the value is truncated to the number of significant figures specified in the argument and formatted in Scientific Notation.
`fractionalPart()`	Returns the fractional component.
`gammal()`	Returns the logarithm of the gamma function.
`ilogb()`	Returns an `integer` that is the binary exponent of non-zero operand.
`isFinite()`	Returns a `boolean` true if and only if the value it is called on is not `+Infinity`, `-Infinity`, or `NaN`.
`isInfinite()`	Returns a `boolean` true if and only if the value it is called on is `+Infinity` or `-Infinity`.
`isNaN()`	Returns a `boolean` true if and only if the value it is called on is `NaN`.
`ln()`	Returns the natural log.
`log10()`	Returns the log to base 10.
`mantissa()`	When called on a `float` value, this method returns a mantissa where *x* = `mantissa*2`$^{\texttt{exponent}}$ assuming that `0.5 <= |mantissa| < 1.0`. When called on a `decimal` value, this method returns a mantissa where *x* = `mantissa*10`$^{\texttt{exponent}}$ assuming that `0.1 <= |mantissa| < 1.0`.
`max(`*decimal*`, `*decimal*`)` or `max(`*float*`, `*float*`)`	Returns the value of the larger operand. You can call this method on the `decimal` or `float` type or on an instance of a `decimal` or `float` type.
`min(`*decimal*`, `*decimal*`)` or `min(`*float*`, `*float*`)`	Returns the value of the smaller operand. You can call this method on the `decimal` or `float` type or on an instance of a `decimal` or `float` type.

Method	Result
`nextafter(y)`	Returns the next distinct floating-point number after the operand that is representable in the underlying type in the direction toward *y*.
`parse(string)`	Returns the `decimal` or `float` instance represented by the `string` argument.
`pow(decimal)` or `pow(float)`	Returns the operand to the power *y*.
`rand()`	Returns a random value from 0.0 up to (but not including) the operand.
`round()`	Rounds to the nearest integer and returns that `integer`.
`scalbn(integer)`	When called on a `float`, returns *operand*`*2`*integer*. When called on a `decimal`, returns *operand*`*10`*integer*.
`sin()`	Returns the sine.
`sinh()`	Returns the hyperbolic sine.
`sqrt()`	Returns the positive square root.
`tan()`	Returns the tangent.
`tanh()`	Returns the hyperbolic tangent.
`toDecimal()`	Returns a `decimal` representation of the `float`
`toFloat()`	Returns a `float` representation of the `decimal`.
`toString()`	Returns a `string` representation.

EPL Methods Quick Reference

dictionary methods

Method	Result
`add(key, item)`	Adds an entry to the dictionary.
`canParse(string)`	When the item type is parseable returns a `boolean` true if the argument can be successfully parsed to create a `dictionary` object.
`clear()`	Sets the size of the `dictionary` to `0`, deleting all entries.
`clone()`	Returns a new `dictionary` that is an exact copy.
`getOr(key, alternative)`	Returns the *item* that corresponds to the specified key. If the specified key is not in the dictionary, the `getOr()` method returns *alternative*.

Method	Result
`getOrDefault(`*`key`*`)`	Retrieves an existing item by its key or returns a default instance of the dictionary's item type if the dictionary does not contain the specified key.
`getOrAdd(`*`key, alternative`*`)`	Retrieves an existing item by its key or adds the specified key to the dictionary with *`alternative`* as its value if it is not already present and also returns the specified *`alternative`*.
`getOrAddDefault(`*`key`*`)`	Retrieves an existing *`item`* by its key or, if it is not already present, adds the specified key with a default instance of the dictionary's item type and returns that instance.
`hasKey(`*`key`*`)`	Returns a `boolean` true if a key exists within the dictionary.
`keys()`	Returns a `sequence` of the dictionary's keys sorted in ascending order.
`parse(string)`	When the item type is parseable returns the `dictionary` object represented by the argument.
`remove(`*`key`*`)`	Removes an entry by key.
`size()`	Returns as an `integer` the number of elements in the dictionary.
`toString()`	Converts the entire dictionary in ascending order of key values to a `string`.
`values()`	Returns a `sequence` of the dictionary's items sorted in ascending order of keys.
`[key]`	Retrieves or overwrites an existing item by its key, or creates a new item.

EPL Methods Quick Reference

event methods

Method	Result
`canParse(`*`string`*`)`	On events that are parseable returns a `boolean` true if the argument can be successfully parsed.
`clone()`	Returns a new `event` that is an exact copy
`getFieldNames()`	Returns a `sequence` of strings that contain the field names of an event type.
`getFieldTypes()`	Returns a `sequence` of strings that contain the type names of an event type's fields.
`getFieldValues()`	Returns a `sequence` of strings that contain the field values of an event.
`getName()`	Returns a `string` whose value is an event's type name.

Method	Result
`getTime()`	Returns a `float` that indicates a time expressed in seconds since the epoch, January 1st, 1970.
`isExternal()`	Returns a `boolean` true if the event was generated by an external source.
`parse(`*`string`*`)`	On events that are parseable returns the `event` object represented by the argument.
`toString()`	Returns a `string` representation of the event.

EPL Methods Quick Reference

Exception methods

`com.apama.exceptions.Exception`

Method	Result
`getMessage()`	Returns a `string` that contains the exception message.
`getStackTrace()`	Returns a `sequence` of `StackTraceElement` objects that represent the stack trace for when the exception was first thrown.
`getType()`	Returns a `string` that contains the exception type.
`toString()`	Returns a `string` that contains the exception message and the exception type.
`toStringWithStackTrace()`	Returns a `string` that contains the exception message, the exception type, and the stack trace elements.

EPL Methods Quick Reference

integer methods

Method	Result
`abs()`	Returns as an `integer` the absolute value.
`canParse(`*`string`*`)`	Returns a `boolean` true if the argument can be successfully parsed.
`getUnique()`	Generates a unique `integer` in the scope of the correlator. This is a type method as well as an instance method.
`max(`*`integer, integer`*`)`	Returns as an `integer` the value of the larger operand. You can call this method on the `integer` type or on an instance of an `integer` type.

Method	Result
`min(integer, integer)`	Returns as an `integer` the value of the smaller operand. You can call this method on the `integer` type or on an instance of an `integer` type.
`parse(string)`	Returns the `integer` instance represented by the argument. You can call this method on the `integer` type or on an instance of an `integer` type.
`pow(integer)`	Returns as an `integer` the value of the operand to the power of the argument.
`rand()`	Returns a random `integer` value from `0` up to (but not including) the value of the operand.
`toDecimal()`	Returns a `decimal` representation.
`toFloat()`	Returns a `float` representation.
`toString()`	Returns a `string` representation.

EPL Methods Quick Reference

listener methods

Method	Result
`quit()`	Immediately terminates the listener.

EPL Methods Quick Reference

location methods

Method	Result
`canParse(string)`	Returns a `boolean` true if the argument can be successfully parsed.
`clone()`	Returns a new `location` that is an exact copy.
`expand(float)`	Returns a new `location` expanded by the value of the parameter in each direction.
`inside(location)`	Returns a `boolean` true if the location is entirely enclosed by the space defined by the parameter.
`parse(string)`	Returns the `location` instance represented by the argument.
`toString()`	Returns a `string` representation.

EPL Methods Quick Reference

sequence methods

Method	Result
`append(item)`	Appends the item to the end of the operand.
`appendSequence(sequence)`	Appends the sequence to the end of the operand..
`canParse(string)`	Returns a `boolean` true if the argument can be successfully parsed to create a `sequence` object.
`clear()`	Sets the size of the sequence to `0`, deleting all entries.
`clone()`	Returns a new `sequence` that is an exact copy.
`indexOf(item)`	Returns as an `integer` the location of the first matching item.
`insert(item, integer)`	Inserts the item specified in the location indicated by the second argument.
`parse(string)`	Returns the `sequence` object represented by the argument.
`remove(integer)`	Removes the n^{th} element in the sequence, moves all the elements above it down, which reduces the size by 1. The first element in a `sequence` is at location `0`.
`reverse()`	Reverses the order of the items in the sequence.
`setCapacity(integer)`	Sets the amount of memory initially allocated for the sequence.
`setSize(integer)`	Sets the number of elements in the sequence.
`size()`	Returns as an `integer` the number of elements in the sequence.
`sort()`	Sorts the sequence in ascending order.
`toString()`	Converts the sequence to a `string`.
`[integer]`	Retrieves or overwrites the sequence entry located at the index specified. EPL sequence elements are indexed from `0`

EPL Methods Quick Reference

StackTraceElement methods

`com.apama.exceptions.StackTraceElement`

Method	Result
`getActionName()`	Returns a `string` that contains the name of the action in which the exception occurred.
`getFilename()`	Returns a `string` that contains the name of the file that contains the code in which the exception occurred.
`getLineNumber()`	Returns an `integer` that indicates the line number of the code in which the exception occurred.
`getTypeName()`	Returns a `string` that indicates the type (event, aggregate, monitor) that contains the action in which the exception occurred..
`toString()`	Returns a string whose format is "*`typeName`*`.`*`actionName`*`()` *`filename`*`:`*`linenumber`*".

EPL Methods Quick Reference

stream methods

Method	Result
`clone()`	Returns the original `stream`. It does not clone it.
`quit()`	Causes a stream listener to terminate.

EPL Methods Quick Reference

string methods

Method	Result
`canParse(`*`string`*`)`	Returns a `boolean` true if the argument can be successfully parsed.
`clone(`*`string`*`)`	Returns a reference to the specified `string`.
`find(`*`substring`*`)`	Returns an `integer` indicating the index position of the argument. EPL string indices start at `0`.
`findFrom(`*`substring`*`,` *`fromIndex`*`)`	Behaves like `find()` but starts searching at *`fromIndex`*.
`intern()`	Marks the `string` as interned. Subsequent incoming events that contain a `string` that is identical to an interned `string` use the same `string` object.
`join(sequence<string>` *`s`* `)`	Concatenates the strings in *`s`* using the operand as a separator.

Method	Result
`length()`	Returns an `integer` indicating the length of the string.
`ltrim()`	Returns a `string` where all white space characters at the beginning have been removed.
`parse(string)`	Returns the `string` value represented by the `string` argument without enclosing that value in quotation marks. You can call this method on the `string` type or on an instance of a `string` type.
`replaceAll(string1, string2)`	Makes a copy of the operand `string`, replaces instances of *`string1`* with instances of *`string2`* and returns the revised `string`.
`rtrim()`	Returns a `string` where all *whitespace* characters at the end have been removed.
`split(string)`	Returns a `sequence` of strings that represent the argument split at occurrences of the operand string.
`substring(integer, integer)`	Returns the substring indicated by the `integer` arguments.
`toBoolean()`	Returns a `boolean` true if the string is "`true`".
`toDecimal()`	Returns a `decimal` representation of the string.
`toFloat()`	Returns a `float` representation of the string.
`toInteger()`	Returns an `integer` representation of the string.
`tokenize(string)`	Categorizes each character in the argument as either part of a delimiter (the character appears in the operand string) or part of a token (any other character), divides the argument into tokens separated by delimiters, and returns the tokens as a `sequence` of strings.
`toLower()`	Returns an all-lowercase `string` representation.
`toUpper()`	Returns an all-uppercase `string` representation

EPL Methods Quick Reference

built-in monitor methods

Method	Result
`onload()`	Invoked immediately after a monitor has been loaded.
`ondie()`	Invoked when a monitor instance terminates.

Method	Result
`onunload()`	Invoked after all instances of a monitor have terminated.
`onBeginRecovery()`	Invoked at the start of recovery of a persistence-enabled correlator.
`onConcludeRecovery()`	Invoked at the end of recovery of a persistence-enabled correlator.

EPL Methods Quick Reference