Developing Apama Applications in Java

5.2.0

October 2014

This document applies to Apama 5.2.0 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.

Document ID: PAM-Developing_Apama_Applications_in_Java-5.2.0-20141001@237655

Table of Contents

Preface

About this documentation

Developing Apama® Applications in Java provides information and instructions for using Apama's in-process API for Java, called JMon, to write applications that run on the event correlator. To develop an Apama application you can use the correlator's native Event Processing Language (EPL), which is the new name for MonitorScript, or JMon, or Apama's Event Modeler. This document focuses exclusively on how to use JMon to write an application that runs on the correlator.

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

JMon reference documentation is provided in Javadoc format.

The JMon API is supported on Microsoft Windows, Solaris, and Linux against Java 6 update 27 and Java 7 update 5.

Preface

How this book is organized

The information in this book is organized as follows:

- "Overview of Apama JMon Applications" on page 9 introduces the main JMon constructs of event types, monitors, and listeners, and describes the flow of control.
- "Defining Event Expressions" on page 27 discusses how to specify the events that you want to trigger some action.
- "The Concept of Time in the Correlator" on page 49 provides information about how the correlator maintains the current time. An understanding of this is vital to a successful Apama application.
- "Developing and Deploying JMon Applications" on page 55 provides instructions for building, testing, and deploying JMon applications.

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 5.2 > Apama 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 JMon 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.

Title	Contents
Developing Apama Applications in Java	Introduces the Apama in-process API for Java, referred to as JMon, 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.
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 messaging for JMS to reliably send and receive JMS messages in Apama applications. • Use Universal Messaging to connect correlators.
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

Chapter 1: Overview of Apama JMon Applications

The event correlator is Apama®'s core event processing and correlation engine. Interfaces to the correlator let you inject monitors that

- Analyze incoming event streams to find patterns of interest
- Specify the actions to undertake when the correlator identifies such patterns

You can use the Apama JMon API to write applications that are to be deployed on the correlator.

The correlator embeds a Java Virtual Machine in which Apama JMon applications can be loaded and run.

The JMon API provides a suite of Java classes that allow a developer to build a Java application, and then inject it into the correlator. Apama JMon applications can define *listeners*, which specify patterns and sequences of events to look for and actions to carry out when the correlator detects those events.

You can develop Apama JMon applications in Apama Studio. When you use Apama Studio to develop an application, it can automatically generate a framework for your JMon event and JMon monitor files.

For more information on developing JMon applications in Apama Studio, see, "Adding a new JMon application", "Adding an Apama JMon monitor", and "Adding an Apama JMon event" in *Using Apama Studio*.

Note: Apama includes the in-process API for Java (JMon) and the client API for Java. In most cases, the context makes it clear which API the discussion is addressing. When this is not clear, the APIs are referred to as the JMon API or Apama client API for Java.

Introducing JMon API concepts

This section introduces the main concepts behind programming the functionality within Apama using JMon. It describes how events are modeled in JMon and how they are used to drive and trigger *listeners* within JMon *monitor* classes.

Apama is designed to fit within an event (or message) driven world. In event driven systems information is propagated through units of information termed events or messages. Conceptually, an event typically represents an occurrence of a particular item of interest at a specific time, and is usually encoded as an asynchronous network message.

Apama is designed to process thousands of these event messages per second, and to sift through them for sequences of interest based on their contents as well as their temporal relationships to each other. When writing Apama applications using JMon, the Java code you write informs the correlator of the sequences of interest and, when matching event sequences are detected, these are passed to your JMon code for handling. Apama's correlator component is capable of looking for hundreds of thousands of different event sequences concurrently.

In order to program the correlator using JMon, a developer must write their application as a set of Java classes that implement the JMon APIs. This programming model is similar to writing Enterprise JavaBeans intended for use in an application server. These Java classes then need to be loaded (or 'injected') into the correlator, which instantiates and executes them immediately.

Almost all of the standard language functionality provided by Java and its libraries can be used in JMon applications, just as in any other Java applications. However, the power of the correlator is only truly leveraged by invoking its event matching, correlation and event generation capabilities. As streams of events are passed into a correlator, the *listeners* defined in JMon applications sift through the events looking for specific sequences of interest matching a variety of temporal constraints. Once a listener triggers, a method is invoked on a Java object, the *Match Listener* object. The developer specifies this object when the listener is created.

Three kinds of Java class objects can be loaded into the correlator; event types, monitors and match listeners.

- Event type classes serve to define the event types that the correlator can accept from external sources and carry out correlations on.
- Monitor classes program the correlator. They define what event patterns the correlator must look for and allow arbitrary Java code to be executed.
- Match listeners provide a method that is called when a specific event sequence is detected.

These three Java class types will be now be discussed in detail.

Overview of Apama JMon Applications

About event types

Apama events are strongly typed. Each event must be of a specific known type, henceforth called the *event type*. An event type defines the name of the event, and its particular set of parameters. Every parameter is named and can be one of a selection of types. Every event instance of a given event type is therefore identical in structure; every instance has the same set (and order) of parameters.

Before the correlator can understand and process events of a specific event type, it needs to have been provided with an *event type definition*. This allows it to understand the event messages it is passed, create optimal indexing structures, and allows listeners to be set up to look for event sequences involving events of that type.

An event type definition defines the event type's name and the name, type and order of each of its parameters. Parameters can be of any of the following types:

- Java standard types `String`, `long`, `double`, `boolean` or `Map`.
- Java arrays.
- `com.apama.jmon.Location` type — This type corresponds to either a spatial point represented by two coordinates, or a rectangular space expressed in terms of its two bounding corners.

Apama's JMon API supports Java generic maps. Apama recommends that you use these when possible instead of the `Event.getMapFieldTypes()` method. Doing so lets you gain the benefits of compile-time type safety as well as a simpler class definition.

However, while it is valid to declare a parameter to be an array of generic maps, assignment of values to the map elements is not type-safe, and will be rejected by the Java compiler. If you need a parameter that is an array of maps, use the `Event.getMapFieldTypes()` method instead of generic maps.

You can nest a plain `Map` as a value (not a key) at any depth in a parameterized `Map`. You cannot nest a parameterized `Map` in a plain `Map`. This is because you would not be able to specify the parameterized types to be returned from the `getMapFieldTypes()` method. Of course, you can nest a parameterized `Map` as a value (but not a key) in a parameterized `Map`. For example:

EPL:

```
Event BadComplexEventExample {
   Dictionary < string , dictionary < string, SimpleEvent > >  complex;
}
```

Java:

```
Import java.util.Map;
Import java.util.HashMap;
Import com.apama.jmon.Event;
Public class BadComplexEventExample extends Event {
// By using a non-parameterized map you lose the information that the field
// is a dictionary with values that are also dictionaries.
Public Map complex;

Public BadComplexEventExample() {
   This(new HashMap());
}

Public BadComplexEventExample(Map complex) {
   This.complex = complex;
}
}
```

See also the definition of `ComplexEvent` on page 30.

An event can embed an event (potentially of a different type) as a parameter.

Overview of Apama JMon Applications

Simple example of an event type

An event type is defined as a Java class as per the following example,

```
/*
 * Tick.java
 *
 * Class to abstract an Apama stock tick event. A stock tick event
```

```
 * describes the trading of a stock, as described by the symbol
 * of the stock being traded, and the price at which the stock was
 * traded
 */

import com.apama.jmon.Event;

public class Tick extends Event {
   /** The stock tick symbol */
   public String name;

   /** The traded price of the stock tick */
   public double price;

   /**
    * No argument constructor
    */
   public Tick() {
      this("", 0);
   }

   /**
    * Construct a tick object and set the name and price
    * instance variables
    *
    * @param name  The stock symbol of the traded stock
    * @param price The price at which the stock was traded
    */
   public Tick(String name, double price){
      this.name = name;
      this.price = price;
   }
}
```

By Java programming conventions, the previous definition would need to be provided on its own in a stand-alone file, for example, `Tick.java`.

The definition must import the definition of the `Event` class. This is provided as part of the `com.apama.jmon` package provided with your Apama distribution. See "Developing and Deploying JMon Applications" on page 55 on installation and deployment for details of where to locate this package.

`Event` is the abstract superclass of all user classes implementing desired event types. Then we must define our new event class as a subclass of the `Event` type.

The user-defined event class must have three primary elements:

- A set of `public` variables that define the event's parameters
- A 'no argument' constructor, whose purpose is to construct an instance of the event with the parameters set to default values
- A constructor whose parameter list corresponds (in type and order) to the event's parameters. This constructor allows creation of an instance of the event with specific parameter values.

In the above example the event is called `Tick`, and it has two parameters, `name`, of type `String`, and `price`, of type `double`. The previous definition may be considered a simple template for how to write all event definitions.

Note: Non-`public` (like `private` and `protected`) variables are not considered to be part of the event schema.

About event types

Extended example of a JMon event type

Let us now consider an extended example:

```
package test.jmon.example;

import java.util.Map;
import com.apama.jmon.*;

/*
 * TestEvent.java
 *
 * Class to abstract an Apama event whose primary purpose is to
 * showcase how to define an event class containing parameters of
 * all the allowed types, including arrays and Maps.
 */

public class TestEvent extends Event {

   // example of parameters of the basic types
   public long primitiveInteger;
   public double primitiveFloat;
   public boolean primitiveBoolean;
   public String referenceString;
   // example of parameters consisting of arrays of the basic types
   public long[] sequenceInteger;
   public double[] sequenceFloat;
   public boolean[] sequenceBoolean;
   public String[] sequenceString;

   // a nested event of type EmbeddedTestEvent
   public EmbeddedTestEvent referenceNestedTestEvent;

   // a parameter of type Location
   public Location referenceLocation;

   // a parameter of type Map
   public Map<long, String> dictionaryIntegerString;
. . .

}
```

About event types

Comparing JMon and EPL event type parameters

EPL - You might already be familiar with EPL, the Apama complex event processing scripting language through which the correlator can be programmed as an alternative to JMon. Event types defined in JMon can be used in EPL, and vice-versa. Jmon event type parameters map to EPL parameter types as follows:

JMon Type	Equivalent EPL Type
`long`	`integer`
`double`	`float`

JMon Type	Equivalent EPL Type
`boolean`	`boolean`
`String`	`string`
`Location`	`location`
`array`	`sequence` (of the same type)
`Map`	`dictionary` (with the same key and value types)
`com.apama.Event` or its subclass	`event` (with the same equivalent subset of fields as defined in this table)

The correlator's performance can be optimized by *wildcarding* event type definitions where appropriate. This procedure is described in "Optimizing event types" on page 24.

About event types

About event parameters that are complex types

It is possible in both EPL and JMon to declare a field of an event definition to be a complex type. For example, the `SequenceEvent` definition below defines an event that is constructed from a sequence of `DataHolder` events, which in turn contain a string and an integer. This is defined in EPL in two events thus:

```
event DataHolder {
   string name;
   integer age;
}

event SequenceEvent {
   sequence <DataHolder> complex;
}
```

An example constructed `SequenceEvent` event is show below:

```
SequenceEvent([DataHolder("kap", 1), DataHolder("gbs", 2)])
```

The equivalent event definitions for the above in Java are defined below:

```
import com.apama.jmon.Event;

public class DataHolder extends Event {
   /** Event fields */
   public String name;
   public long age;

   /** No argument constructor
   */
   public DataHolder () {
      this("", 0);
   }
   /** Construct a DataHolder object and set the instance variables
   */
   public DataHolder (String n, long a) {
      name = n;
      age = a;
   }
```

```
}

import com.apama.jmon.Event;

public class SequenceEvent extends Event {

   /** Event field */
   public DataHolder[] people;

   /** No argument constructor
   */
   public SequenceEvent() {
      this( new DataHolder[]{} );
   }

   /** Construct a SequenceEvent object and set the instance variable
   */
   public SequenceEvent(DataHolder[] p) {
      this.people = p;
   }
}
```

Sample Java code to create and emit a `SequenceEvent` event is shown below:

```
  s = new SequenceEvent(new DataHolder[] {new DataHolder("kap", 1),
   new DataHolder("gbs", 2)} );
s.emit();
```

Events can also include `Map` types, which are equivalent to EPL `dictionary` types. When you use `Map` types, Apama recommends that you use generic maps whenever you can. For example, in EPL the following event is a dictionary of dictionaries and each internal dictionary is a sequence of `SimpleEvent` types:

```
event ComplexEvent {
  dictionary <string,
    dictionary <string, sequence<SimpleEvent> > > complex;
}
```

You can implement this in Java as follows:

```
import java.util.Map;
import java.util.HashMap;
import com.apama.jmon.Event;
import com.apama.jmon.annotation.EventType;

@EventType(description = "Event that contains a field with a complex structure")
public class ComplexEvent extends Event {

   /** Event field */
   public Map<String, Map<String, SimpleEvent[]>> complex;
   /**
   * No argument constructor
   */
   public ComplexEvent() {
      this(new HashMap<String, Map<String, SimpleEvent[]>>());
   }

   /**
   * Construct a ComplexEvent object, set the instance variable complex
   *
   * @param complex The dictionary/Map to use as the field value
   */
   public ComplexEvent(
      Map<String, Map<String, SimpleEvent[]>> complex) {
        this.complex = complex;
      }
}
```

This example is provided in its complete form as a sample. It is distributed in the folder `samples/java_monitor/complex_event/`.

About event types

Non-null values for non-primitive event field types

When the correlator creates an event to pass to the JMon code, it ensures that all fields of a non-primitive type have a non-null value. Note that this is different from the Java default, which is to allow null values for non-primitive types.

The `com.apama.jmon.Event` default constructor uses reflection to initialize non-primitive null fields with the following values:

- `sequence` — an empty array of the specified type
- `dictionary` — an empty `java.util.HashMap` object
- `string` — an empty `java.lang.String` object
- `event` — a default construction of the event, with recursive initialization for any of its non-primitive fields that have null values.

In your application, if you explicitly assign a null value to a non-primitive event field, and your application tries to emit, enqueue, or route that event, the correlator logs an error and terminates your application.

About event types

About monitors

Monitor classes configure the activity of the correlator. This is analogous to how an Enterprise JavaBean effectively defines the activity of an application server.

All monitor classes must implement the `com.apama.jmon.Monitor` interface and define an `onLoad` method. When a monitor class is loaded into the correlator, it is instantiated as an object and its `onLoad` method is executed. In Java parlance, this would be equivalent to the `static void main (args[])` method.

Most Java code (with certain limitations) can be executed within the `onLoad` method, although its primary purpose is probably to configure one or more asynchronous *listeners* for specific events or event sequences.

A monitor class must define a 'no argument' constructor. The Java code within the correlator uses this when the class definition is loaded.

Below is a minimal monitor:

```
import com.apama.jmon.*;

public class Simple implements Monitor {
   /**
    * No argument constructor used by the jmon framework on
    * application loading
    */
   public Simple() {}

   /**
    * Implementation of the Monitor interface onLoad method.
    * Does nothing.
    */
```

```
    public void onLoad() {
    }
}
```

The above monitor class does nothing and is shown here as a template for how to define a monitor class.

EPL - Although there are similarities, the concept of a monitor in EPL and in JMon is not the same. The EPL monitor is a very powerful custom programming structure, whereas in JMon a monitor class is primarily a standard Java class with an entry method that gets automatically executed upon loading (as described below).

Overview of Apama JMon Applications

About event listeners and match listeners

For a monitor class to leverage the intrinsic features of the correlator, it must set up one or more *listeners*.

A listener is a conceptual entity whose function is to sift through all incoming event streams looking for a particular event or sequence of events. The event or sequence of events of interest is represented as an *event expression*.

The simplest way of setting up a listener is by creating an instance of an `EventExpression` and then specifying a `MatchListener` object that gets triggered when the expression becomes true, that is, when a suitable event or event sequence is detected. A more efficient alternative is to use a *prepared event expression*, which is described in "Optimizing event types" on page 24.

A match listener is a Java object that implements the `com.apama.jmon.MatchListener` interface and implements the `match` method. This method is called by the correlator when the event expression it is registered with is detected.

This section discusses the following topics:

Overview of Apama JMon Applications

Example of a MatchListener

The following example illustrates this functionality:

```
import com.apama.jmon.*;
public class Simple implements Monitor, MatchListener {

    /**
     * No argument constructor used by the jmon framework on
     * application loading
     */
    public Simple() {}
    /**
     * Implementation of the Monitor interface onLoad method. Sets up
     * a single event expression looking for all Tick events
     * with a trade price of greater than 10.0. This class instance
```

```
     * is added as a match listener to the event expression.
     */
    public void onLoad() {
       EventExpression eventExpr = new EventExpression("Tick(*, >10.0)");
       eventExpr.addMatchListener(this);
    }

    /**
     * Implementation of the MatchListener interface match method.
     * Prints out
     * a message when the listener triggers
     */
    public void match(MatchEvent event) {
       System.out.println("Pattern detected");
    }
}
```

This example illustrates several new concepts.

Consider the `onLoad` method. Firstly it creates an event expression object variable. This object, of type `com.apama.jmon.EventExpression`, represents an event, or sequence of events, to look for. The constructor of an `EventExpression` is passed a string that defines the actual event expression.

As the syntax of an event expression will be illustrated in the next section it is enough to say that this event expression is specifying "the *first* `Tick` event whose *`price`* parameter is greater than the value `10.0`".

Then, a match listener is registered with the newly created event expression object. A match listener can be any object that implements the `com.apama.jmon.MatchListener` interface and defines the `match(MatchEvent event)` method. For the sake of simplicity, the `Simple` monitor class has here been written to also implement the `MatchListener` interface, and therefore the statement,

```
eventExpr.addMatchListener(this);
```

is passing `this` as the reference to a suitable `MatchListener`.

Once a match listener has been registered with an event expression the correlator creates a listener entity to start looking for the specified event expression.

Listeners are asynchronous. Hence the `match` method may be invoked at any time subsequent to the activation of the listener, but always after all Java code in the current method finishes executing. Therefore in this case all Java statements in the `onLoad` method would finish being executed before `match` is called after a match.

About event listeners and match listeners

Defining multiple listeners

A monitor can define any number of event expressions, and create any number of listeners. The following code,

```
public void onLoad() {
   EventExpression eventExpr1 = new EventExpression("Tick(*, >10.0)");
   EventExpression eventExpr2 =
      new EventExpression("NewsItem(\"ACME\", *)");

   eventExpr1.addMatchListener(this);
   eventExpr2.addMatchListener(this);
}
```

is creating two event expressions, `eventExpr1` and `eventExpr2`. Then each is assigned a match listener, thus activating two distinct listeners. The fact that both are being assigned the same match listener

object, i.e. this same object `this`, is inconsequential. It just means that the same method, the `match` method of this object, will be called when the correlator detects either of the event expressions.

As already described, creating a listener is an asynchronous operation that returns immediately. In the above code, in practice both listeners are created concurrently. It is not possible for the `eventExpr1` listener to trigger before the `eventExpr2` listener is created. However, once the enclosing method's code has completed execution, the listeners can trigger at any time, and independently of each other.

About event listeners and match listeners

Removing listeners

A `MatchListener` instance that is no longer connected to an event expression, and to which there are no references, is garbage collected in the usual way. In some situations, you might want to be notified when the correlator removes its reference to the `MatchListener` (when it can no longer fire). For example, you might need this notification if the `MatchListener` has unmanaged resources (for example, open files) that need to be explicitly cleaned up when it is no longer needed, or your application has other references to the `MatchListener` that need to be removed when the listener can no longer fire so that it can be garbage collected. In those situations, you can define your listener so that it implements the `com.apama.jmon.RemoveListener` interface. There is no requirement to implement this interface. It is up to you to determine whether you need it.

The `RemoveListener` interface extends the MatchListener interface by providing one additional method: `removed()`. If you implement the `RemoveListener` interface, the correlator calls your implementation of the `removed()` method in the following situations:

- The application removes your listener from the event expression it is attached to.
- The event expression your listener is attached to is in a state that will never match. For example, `on A() within (10.0)` after 10 seconds have elapsed without an `A()`.

In the following example, the `removed()` method is called because the event expression dies after 10 seconds.

```
import java.util.HashMap;
import com.apama.jmon.*;

public class Test implements Monitor {
   public Test() {}
   public void onLoad() {
      EventExpression e = new EventExpression("TestEvent() within(10.0)");

      e.addMatchListener(new RemoveListener() {
         public void match(MatchEvent event) {
            System.out.println(Correlator.getCurrentTime() +
               ": Received match");
         }
      public void removed(EventExpression e) {
         System.out.println(Correlator.getCurrentTime() +
            ": Received removed");
      }
      });
   }
}
```

About event listeners and match listeners

Description of the flow of execution in JMon applications

The flow of execution of JMon applications through the correlator at any given time is single threaded. All the listeners of JMon applications are fired in a single-threaded manner. However, during the lifetime of a JMon application, its execution may be moved among a number of threads by the correlator. This is particularly important since thread-local variables will not behave in the same way as you would expect them to in a conventional Java application.

When a number of monitor classes are loaded into the JVM within the correlator their `onLoad` methods are executed in turn, in the same order as the injected classes, and any listeners created are set up and activated.

Control then reverts to the correlator, which takes in one event from its input queue. This event is examined by each of the active listeners in turn (the order is undefined), and each one that triggers immediately calls the `match()` method in its registered `MatchListener` object.

Once all the listeners have processed the event (and hence all `match` methods terminated), control reverts to the correlator to process the next input event. Note that since events can also match listeners in EPL monitors, these would also be processed before control reverts.

However, JMon applications can create other Java threads. In such multi-threaded JMon applications, the correlator has no control of these additional Java threads. Consequently, you should never route or emit an event from a Java thread that was not the thread in which the correlator invoked the JMon application. Doing this results in unpredictable behavior. To communicate from your JMon application to other parts of the correlator, use the `enqueue()` method or preferably, the `enqueueTo()` method.

Overview of Apama JMon Applications

Parallel processing in JMon applications

By default, the correlator operates in a serial manner. If you want, you can implement contexts for parallel processing. You can create contexts only with EPL but you can then use those contexts from your Apama JMon code. This section provides information about how to use contexts in Apama JMon applications.

For general information about contexts and instructions for creating contexts, see "Implementing parallel processing" in *Developing Apama Applications in EPL*.

You can find a sample JMon application that implements the use of contexts in the `samples\java-monitor\context` directory of your Apama installation directory.

The topics in this section include:

Overview of Apama JMon Applications

Overview of contexts in JMon applications

The Apama JMon API provides the `com.apama.jmon.Context` type. This class corresponds to the EPL `context` type, but with a more limited set of features:

- A JMon event definiton can contain a `Context` type field. This lets you transfer a reference to a context to and from an Apama JMon application. You cannot pass context references between the correlator and your Apama JMon application on their own.
- You can enqueue events to
  - Particular contexts:`Event.enqueueTo(Context c)`
  - A list or array of contexts:

    ```
    Event.enqueueTo(java.util.List<Context> ctxList)
    Event.enqueueTo(Context[] ctxList)
    ```

    See "Emitting, routing, and enqueuing events" on page 32.
- You can call `Context.getCurrent()` to obtain a reference to the context that a piece of code is running in.

  See "Obtaining context references" in *Developing Apama Applications in EPL*.
- The `Context` class provides accessor methods for context properties such as context name and context ID.

Parallel processing in JMon applications

Using contexts in JMon applications

To use EPL contexts in JMon applications:

1. In EPL code, create a context that you want to use in your JMon application.
2. In your JMon application, define an event type that contains a `Context` field.
3. Use this event type to obtain a reference to the context you created in EPL.
4. Use the context reference to enqueue events to that context.

For an example, see the `samples\java-monitor\context` directory in your Apama installation directory.

Parallel processing in JMon applications

Using the Context class default constructor

The `com.apama.jmon.Context` class default constructor, `public Context()`, creates a dummy context that provides the same functionality as an uninitialized `context` variable in EPL. A JMon dummy context does not correspond to an actual correlator context. The JMon dummy context corresponds to the implicit context that is created in EPL for uninitialized context variables. The default constructor

is provided for convenience. Use it when you want to enqueue an event to another context from a JMon application and the event happens to have a context field that contains an irrelevant value. As with other JMon types, this value cannot be null. Following is an example, beginning with the event definition:

```
import com.apama.jmon.*;

public class ContextEvent extends Event {
   public long id;
   public boolean req;
   public Context c;
   public ContextEvent(long id) {
      this.id = id;
      this.req = true;
      this.c = new Context();
   }
}
```

Here is the JMon application:

```
public class SampleJMonApp implements Monitor {
   ...
   public void onLoad() {
   ...
   ContextEvent req = new ContextEvent(service_id);
   ...
   // send requests here
   req.route();
   ...
   EventExpression cexpr =
      new EventExpression("all ContextEvent(*,false,*):ackEvt");
         cexpr.addMatchListener(new MatchListener() {
            public void match(MatchEvent event){
               ContextEvent ackEvt =
               (ContextEvent)event.getMatchingEvents().get("ackEvt");
               // extract the context here
               Context serviceContext = ackEvt.evt;
               ...
            }
         });
   }
   ...
}
```

Here is the EPL application:

```
monitor ContextFactory
{
   ...
   action onload() {
   ...
      ContextEvent req;
      on all ContextEvent(*, true, *):req {
         integer svcid;
         ...
         context serviceContext := context("svc");
         ContextEvent ack :=
            ContextEvent(svcid, false, serviceContext);
         route ack;
         ...
      }
   }
   ...
}
```

Parallel processing in JMon applications

Descriptions of methods on the Context class

You can call the following methods on a `Context` object. For more information, see "context" in the "Types" section of the *Apama EPL Reference*.

- `public long getId()`

  Returns the unique identifier for the context. For a `Context` instance that would return the following `toString()` result: `"context(2,"context_name",false)"`, the `getId()` method returns the value `2`. This method returns `0` for a `Context` instance created with the default constructor.

- `public String getName()`

  Returns the name of the context. For example, suppose you create a context with the following EPL code:

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

  If you transfer a reference to this context into your JMon application, a call to the `getName()` method on this context instance returns `"test"`.

  This method returns an empty string for a `Context` instance created with the default constructor.

- `public String toString()`

  Returns a string representation of the context instance. This method produces a string that is identical to the string that EPL produces. For example: `"context(2,"context_name",false)"`. The first item in the string, `2` in this example, is the context's unique identifier. The second item in the string, `"context_name"`, is the name of the context. The third item in the string is the value of the `receivesInput` boolean flag, which indicates whether the context is public or private.

  This method returns `"context(0,"",false)"` for a `Context` instance created with the default constructor.

  For details about public and private contexts, see *Developing Apama Applications in EPL*, "Implementing parallel processing", "Creating contexts".

- `public static Context getCurrent()`

  Returns a `Context` instance that corresponds to the current correlator context. This is the context that contains the code that you are calling. Apama executes single-threaded JMon applications in the main correlator context. Consequently, this method always returns a a `Context` instance that references the main correlator context.

  During execution, JMon applications can create new Java threads. Do not confuse new threads with correlator contexts. The `Context.getCurrent()` method returns null when you call it inside newly created Java threads.

Parallel processing in JMon applications

Identifying external events

In some situations, you might want to determine whether an event originated outside the correlator. To do this, call the `Event.isExternal()` method:

```
public boolean isExternal()
```

This method returns true if the event was sent to the correlator by some external process and that event was then passed into your JMon application.

Overview of Apama JMon Applications

Optimizing event types

"About event types" on page 10 introduced event type classes.

The correlator creates several indexing data structures for every event type. The complexity and efficiency of these data structures depends on the number of parameters an event has, and therefore 'smaller' (with less parameters) events are processed more rapidly.

Therefore, if possible, when designing an application it is preferable to control it using a number of 'smaller' event types rather than through a single event type with a large number of parameters.

Overview of Apama JMon Applications

Wildcarding parameters in event types

Alternatively, if large event types are unavoidable, you can optimize performance by reviewing the usage of these event types in JMon, specifically within event templates in event expressions.

If a parameter of an event is never matched against directly within any event expressions, that is only '*' (or wildcard) ever appears against it in event templates, then the event type's definition can be amended to indicate this. This tells the correlator to ignore this parameter in its internal indexing.

Consider the event type definition presented in "About event types" on page 10

```
/*
 * Tick.java
 *
 * Class to abstract an Apama stock tick event. A stock tick event
 * describes the trading of a stock, as described by the symbol
 * of the stock being traded, and the price at which the stock was
 * traded
 *
 */
import com.apama.jmon.Event;

public class Tick extends Event {
   /** The stock tick symbol */
   public String name;

   /** The traded price of the stock tick */
   public double price;

   /**
    * No argument constructor
    */
   public Tick() {
      this("", 0);
   }

   /**
    * Construct a tick object and set the name and price
    * instance variables
    *
```

```
    * @param name  The stock symbol of the traded stock
    * @param price The price at which the stock was traded
    */
   public Tick(String name, double price){
      this.name = name;
      this.price = price;
   }
}
```

If all references to this event type in event expressions look similar to this,

```
Tick("ACME", *)
```

that is, where the second parameter `price` is always specified as a `*`, then this parameter could be *wildcarded* in the event type definition.

This can be done by annotating the field in the event type class, as shown here

```
/** The traded price of the stock tick */
@com.apama.jmon.annotation.Wildcard
public double price;
```

This definition in the `Tick` class will override the default behavior, and it lets the correlator know that it can optimize its indexing by ignoring the `price` parameter.

As many parameters as desired can be wildcarded in this way. For example, if both `price` and `name` were to be wildcarded in `Tick`, they should be defined as follows,

```
/** The stock tick symbol */
@com.apama.jmon.annotation.Wildcard
public String name;

/**The traded price of the stock tick */
@com.apama.jmon.annotation.Wildcard
public double price;
```

Of course, if you were to do this, then

```
Tick(*, *)
```

would be the only valid event template that can be expressed in JMon. Any other expression would cause a Java runtime error.

Optimizing event types

Logging in JMon applications

The logging facilities in JMon are provided by `Log4j`, a publicly available logging library for Java. These logging facilities are included in `com.apama.util.Logger`, for which reference information in `javadoc` format is provided (`doc\javadoc\index.html` in your Apama installation directory).

Note: Full documentation for `Log4j` and the Apache Logging Service project can be found at http://logging.apache.org.

By default, the JMon classes will log at `WARN` level. The log level can be changed as described in the Javadoc for the `Logger` class. The Javadoc also provides instructions on how to get a reference to the `Logger` object in your own code so that you can produce your own logging output.

To ensure that the correlator can serialize logging behavior, specify that instances of `Logger` are static.

Overview of Apama JMon Applications

Using EPL keywords as identifiers in JMon applications

If you use EPL keywords as event name or field identifiers, then in the following situations you must escape such identifiers by preceding them with hash (#) symbols:

- You refer to the JMon identifier in EPL code — You must escape the identifier in the EPL code that contains the reference.
- You refer to the JMon identifier in a JMon event expression — You must escape the identifier in that JMon event expression.

For example, consider the following Java code:

```
class test extends Event {
   int id;
   float price;
   int integer;
}
```

Now suppose you want to write the following EPL code:

```
on all test(id=7): f {
   print f.toString();
   emit f;
}
```

No escaping is necessary. However, suppose you want to write this EPL code:

```
print f.integer.toString();
```

In this case, you must escape `integer` as follows: print f.#integer.toString();

Likewise, you must escape `integer` in the following JMon event expression:

```
new EventExpression("all test(#integer > 5)");
```

For a list of EPL keywords, see the Apama EPL Reference, "Lexical Elements", "Keywords".

Overview of Apama JMon Applications

Chapter 2: Defining Event Expressions

Consider this code snippet from the previous example:

```
public void onLoad() {
   EventExpression eventExpr =
      new EventExpression("Tick(*, >10.0)");
   eventExpr.addMatchListener(this);
}
```

The highlighted code is creating an event expression, and embeds the following event expression definition string:

```
Tick(*, >10.0)
```

This is the simplest form of an event expression; specifically it contains a single *event template*.

In this case the event expression is specifying "the first `Tick` event whose `price` parameter contains a value greater than 10.0".

EPL - If you are already familiar with EPL, the syntax for writing JMon event expressions is the same as for EPL event expressions.

About event templates

The first part of an event template defines the event type of suitable events (in this case `Tick`), while the section in brackets describes filtering criteria that must be applied to the contents of events of the desired type for them to match.

In the example at the beginning of the chapter, the first parameter within the event template has been set to a wildcard (`*`), specifying that all `Tick` events, regardless of the value of their `name` parameter, are suitable. That is, as long as their second parameter, `price`, is greater than 10.The filtering criteria supplied are applied to the event's contents in the same order as within the event definition for that event type. This is known as *positional syntax*.

"Specifying parameter constraints in event templates" on page 29 lists all the filtering operators (like "`>`" above) that can be applied to the value of a parameter within an event template.

Defining Event Expressions

Specifying positional syntax

In positional syntax, the event template must define a value (or a wildcard) to match against for every parameter of that event's type, in the same order as the parameter's definition in the event type definition. Therefore, for the event type,

```
public class MobileUser extends Event {
   public long userID;
   public Location position;
   public String hairColour;
   public String starsign;
   public long gender;
   public long incomeBracket;
   public String preferredHairColour;
   public String preferredStarsign;
   public long preferredGender;
   // … Constructors
}
```

a suitable event template definition might look like

```
MobileUser(*,*, "red", "Capricorn", *, *, *, *, 1)
```

This can get unwieldy when you are working with event types with a large number of parameters and very few of them are actually being used to filter on. An alternative syntax can be used that addresses this. The above can instead be expressed as:

```
MobileUser(hairColour="red", starsign="Capricorn",
  preferredGender=1)
```

This is known as *named parameter syntax* and in this style all other non-specified fields are set to wildcard.

Given the following event types:

```
public class A extends Event {
   public long a;
   public String b;

   // … Constructors
}

public class B extends Event {
   public long a;

   // … Constructors
}

public class C extends Event {
   public long a;
   public long b;
   public long c;

   // … Constructors
}
```

Here are some equivalent event expressions that demonstrate how to use the two syntaxes:

	Positional Syntax	Name/Value Syntax
Using constants and literals	`on A(3,"string")` `on A(=3,="string")`	`on A(a=3,b="string")` `on A(b="string",a=3)`

	Positional Syntax	Name/Value Syntax
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()`

More details about the operators and expressions possible within event templates are given in the next section.

Note that it is possible to mix the two styles as long as you specify positional parameters before named ones. There cannot be any positional parameters after named ones. Therefore the following syntax is legal:

```
D(3,>4,i in [2:4])
```

while the following is not:

```
E(k=9,"error")
```

About event templates

Specifying completed event templates

In some situations, you want to ensure that the correlator completes all work related to a particular event before your application performs some other work. In your event template, specify the `completed` keyword to accomplish this. For example:

```
on all completed A(f < 10.0) {}
```

When an event that matches the template comes into the correlator, the correlator

1. Runs all of the event's normal and unmatched listeners.
2. Processes all routed events that result from those listeners.
3. Calls the `completed` listeners.

About event templates

Specifying parameter constraints in event templates

The first part of an event template defines the event type of the event the listener is to match against, while the section in brackets describes further filtering criteria that must be satisfied by the contents of events of that type for a match.

Event template parameter operators specify constraints that define what values, or range of values, are acceptable for a successful event match.

Table 1. Event template parameter operators

Operator	Meaning
`[value1 : value2]`	Specifies a range of values that can match. The values themselves are included in the range to match against. For example: `stockPrice(*, [0 : 10])` This event template will match a `stockPrice` event where the price is between 0 and 10 inclusive. This range operator can only be applied to `double` and `long` types.
`[value1 : value2)`	Specifies a range of values that can match. The first value itself is included while the second is excluded from the range to match against. For example: `stockPrice(*, [0 : 10))` This example will match a `stockPrice` event where the price is between 0 and 9 inclusive (assuming the parameter was of `long` type). This range operator can only be applied to `double` and `long` types.
`(value1 : value2]`	Specifies a range of values that can match. The first value itself is excluded from while the second is included in the range to match against. For example: `stockPrice(*, (0 : 10])` This example will match a `stockPrice` event where the price is between 1 and 10 inclusive (assuming the parameter was of `long` type). This operator can only be applied to `double` and `long` types.
`(value1 : value2)`	Specifies a range of values that can match. The values themselves are excluded in the range to match against. For example: `stockPrice(*, (0 : 10))` This example will match if a `stockPrice` event where the price is between 1 and 9 inclusive (assuming the parameter was of `long` type). This operator can only be applied to `double` and `long` types.
`> value`	All values greater than the value supplied will satisfy the condition and match. This operator can only be applied to `double` and `long` types.
`< value`	All values less than the value supplied will satisfy the condition and match.This operator can only be applied to `double` and `long` types.
`>= value`	All values greater than or equal to the value supplied will satisfy the condition and match. This operator can only be applied to `double` and `long` types.

Operator	Meaning
`<= value`	All values less than or equal to the value supplied will satisfy the condition and match. This operator can only be applied to `double` and `long` types.
value	Only a value equivalent to the value supplied will satisfy the condition and match. A `String` value must be enclosed in double quotes (`" "`), and therefore these need to be preceded with an escape character inside event expression definitions in an `EventExpression` constructor (`\"`) A `Location` value must consist of a structure with four `double`s representing the coordinates of the corners of the rectangular space being represented.
`*`	Any value for this parameter will satisfy the condition and match.

Defining Event Expressions

Obtaining matching events

An event template provides a definition against which several event instances could match. Once a listener triggers, sometimes it is necessary to get hold of the *actual* event that matched the template.

This can be achieved through *event tagging*.

EPL - If you are familiar with EPL, event tagging in JMon is similar in principle to variable coassignment in EPL. For this reason the term *coassigned* is sometimes used to refer to event tagging.

Consider this revised `Simple` monitor:

```
import com.apama.jmon.*;

public class Simple implements Monitor, MatchListener {

   /**
    * No argument constructor used by the jmon framework on
    * application loading
    */
   public Simple() {}

   /**
    * Implementation of the Monitor interface onLoad method. Sets up
    * a single event expression looking for all stock trade events
    * with a trade price of greater than 10.0. This class instantiation
    * is added as a match listener to the event expression.
    */
   public void onLoad() {
      EventExpression eventExpr = new EventExpression("Tick(*, >10.0):t");
      eventExpr.addMatchListener(this);
   }

   /**
    * Implementation of the MatchListener interface match method.
    * Extracts the tick event that caused the event expression to
    * trigger and emits the event onto the default channel
    */
```

```
   public void match(MatchEvent event) {
      Tick tick = (Tick)event.getMatchingEvents().get("t");
      System.out.println("Event details: "  + tick.name
         + " "   + tick.price);
      tick.emit();
   }
}
```

Note the revised event expression

```
Tick(*, >10.0):t
```

This specifies that when a suitable `Tick` event is detected, it must be recorded with the `t` tag. This allows a developer to get hold of the actual event that matched the event expression within the registered match listener's `match` method.

Once the `eventExpr` listener detects a suitable event it will trigger and call `match`, passing to it a `MatchEvent` object. This object embeds within it all the individual event instances that together caused the event expression to be satisfied and were tagged.

In this example our event expression still consists of a single event template, and since this is tagged, then the `MatchEvent` object will contain the single `Tick` event that triggered the `eventExpr` listener. This will be tagged as `t`.

A `MatchEvent` object has two methods:

- `HashMap getMatchingEvents()` - Get the set of tagged `Event`s that caused the match. This method returns a `Map` of the tagged `Event` objects that hold the values that matched the source `EventExpression`.
- `Event getMatchingEvent(String key)` - Get one of the tagged `Event`s that caused the match. This method returns the tagged Event object that matched in the source `EventExpression`.

Refer to the reference documentation provided in Javadoc format for complete class and method signatures (`doc\javadoc\index.html` in your Apama installation directory).

The lines:

```
Tick tick = (Tick)event.getMatchingEvents().get("t");
```

or

```
Tick tick = event.getMatchingEvent("t");
```

show how the tagged event can be extracted by using the tag as a key.

Defining Event Expressions

Emitting, routing, and enqueuing events

Once the event has been extracted it can also be *emitted*, *routed*, or *enqueued*.

This functionality is provided by the following methods of the `Event` class:

- `route()` — Route this event internally within the correlator.
- `emit()` — Emit this event from the correlator onto the default channel.
- `emit(String channel)` — Emit this event from the correlator onto the named channel.
- `enqueue()` — Route this event internally within the correlator to a special queue just for enqueued events.

- `enqueueTo()` — Route this event internally within the correlator to the input queue of the specified context or contexts.

The `route` method generates a new event that is dispatched back into the correlator. Any active listeners seeking that event then receive this. There is no difference between an externally sourced event (passed in through a live message feed) and an event that was issued internally through a `route` method, other than that internally routed events are placed at the front of the input queue, although in the same order as they are routed within an action.

The `emit` method dispatches events to external registered event receivers, i.e. sends them out from the correlator. Active listeners will not receive events that are emitted.

Events are emitted onto named *channels*. For an application to receive events from the correlator it must register itself as an event receiver and *subscribe* to one or more channels. Then if events are emitted to those channels they will be forwarded to it.

Channels effectively allow both *point-to-point* message delivery as well as through *publish-subscribe*. Channels can be set up to represent topics. External applications can then subscribe to event messages of the relevant topics. Otherwise a channel can be set up purely to indicate a destination and have only one application connected to it.

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

Use the `enqueue()` method when you want to ensure that the correlator processes the generated event after it processes all routed events. This means that you want the correlator to finish processing the current external event. Completion of processing the current external event means that all routed events that resulted from that external event have been processed.

In a parallel application, you can enqueue an event to a particular context by calling the following method on an instance of `com.apama.jmon.Event`:

```
public void enqueueTo(Context ctx)
```

This method provides the same functionality provided by the EPL `enqueue ... to` statement. See "Enqueing an event to a particular context" in *Developing Apama Applications in EPL*.

However, it is important to mention that when you enqueue an event to a particular context the event goes on that context's input queue and not on the special queue for enqueued events. Consequently, when you call this method from an application thread that was created from the main JMon application and the destination context's input queue is full, this method blocks until the queue is able to accept the event.

Call the following method to enqueue an event to a array of contexts:

```
public void enqueueTo(Context[] ctxArray)
```

Call the following method to enqueue an event to a list of contexts:

```
public void enqueueTo(List < Context> ctxList)
```

Defining Event Expressions

Specifying temporal sequencing

If you want to search for a temporal sequence of two events, for example, “locate the sequence of a `NewsItem` event followed by a `Tick` event”, there are two ways you can proceed in JMon.

Defining Event Expressions

Chaining listeners

You can *chain* listeners, as follows:

```
// Code within the monitor class

public void onLoad() {
   EventExpression eventExpr = new EventExpression("NewsItem(*, *)");
   eventExpr.addMatchListener(matchListener1);
}

// Code within the first Match Listener class - matchListener1
public void match(MatchEvent event) {

   // Arbitrary additional code …
   EventExpression eventExpr = new EventExpression("Tick(*, *)");
   eventExpr.addMatchListener(matchListener2);
}

// Code within the second Match Listener class - matchListener2

public void match(MatchEvent event) {
   System.out.println("Detected a NewsItem followed"
      + " by a Tick event, both regarding any company.");
}
```

The Java code above shows how to set up a listener to seek the first event, and then once that is located, start searching for the second. This programming style is particularly appropriate when further actions need to be taken at each stage of the event detection, in this case between detecting the `NewsItem` and seeking the `Tick`.

It is also the only way in which the event templates can be ‘linked’ together. If the desired effect was to locate ‘any’ first `NewsItem` and then seek a `Tick` specifically for the same company mentioned in the `NewsItem`, you could amend the example as follows,

```
// Code within the monitor class
public void onLoad() {
   EventExpression eventExpr
      = new EventExpression("NewsItem(*, *):n");
   eventExpr.addMatchListener(matchListener1);
}

// Code within the first Match Listener class - matchListener1
public void match(MatchEvent event) {

   NewsItem newsItem = (NewsItem)event.getMatchingEvents().get("n");
   EventExpression eventExpr
      = new EventExpression("Tick(\"" + newsItem.name + "\", *)");
   eventExpr.addMatchListener(matchListener2);
}

// Code within the second Match Listener class - matchListener2
public void match(MatchEvent event) {
```

```
   System.out.println("Detected a NewsItem, followed"
      + " by an Tick event regarding the same company.");
}
```

Note how the above code seeks out a `NewsItem` on any company, but then extracts the actual `NewsItem` event detected, and uses its `name` parameter to create the event template for seeking the `Tick` event.

Specifying temporal sequencing

Using temporal operators

Let us return to how to express searching for a temporal sequence. If there is no requirement to execute any arbitrary code in between events and there is no requirement to link searches as illustrated above, then you can embed a temporal event expression within a single listener.

The first code excerpt could be re-written as follows,

```
// Code within the monitor class
   public void onLoad() {
      EventExpression eventExpr
         = new EventExpression("NewsItem(*,*) -> Tick(*,*)");
      eventExpr.addMatchListener(matchListener1);
  }

// Code within the first (and only) Match Listener class - matchListener1
   public void match(MatchEvent event) {
      System.out.println("Detected a NewsItem followed"
         + " by a Tick event, both regarding any company.");
  }
```

The event expression definition for `eventExpr` no longer consists of a single event template. It now has multiple clauses and contains a temporal operator.

In this case, the operator used is `->`, or the *followed-by* operator. This is the primary temporal operator for use in event expressions. It allows a developer to express a sequence of events to match against within a single listener, with the listener triggering once the whole sequence is encountered.

In Java, an event sequence does not imply that the events have to occur right after each other, or that no other events are allowed to occur in the meantime.

For the sake of brevity, let `A, B, C` and `D` represent event templates, and `A', B', C'` and `D'` be individual events that match those templates, respectively. If a listener is created to seek the event expression `(A -> B)`, the event feed {`A',C',B',D'`} would result in a match once the `B'` is received by the correlator.

Followed-by operators can be chained to express longer sequences. Therefore you could write,

```
A -> B -> C -> D
```

within an event expression definition.

The next section focuses on the use of temporal operators in event expressions.

Specifying temporal sequencing

Defining advanced event expressions

An event template is the simplest form of an event expression. All event expression operators, including `->`, can themselves take entire event expressions as operands.

It is useful to think of event expressions as being Boolean expressions. Each clause in an event expression can be true or false, and the whole event expression must evaluate to true before the listener triggers and calls the match listener's `match` method.

As before, for the sake of brevity, let us use the letters `A`, `B`, `C` and `D` to represent event templates, and `A'`, `B'`, `C'` and `D'` to represent individual events that match those templates, respectively.

Once more, consider this representation of an event expression,

```
A -> B -> C -> D
```

When the listener is first activated it is helpful to consider the expression as starting off by being false. When an event that satisfies the `A` clause occurs, the `A` clause becomes true. Once `B` is satisfied, `A -> B` becomes true in turn, and evaluation progresses in a similar manner until eventually all `A -> B -> C -> D` evaluates to true. Only then does the listener trigger and call the associated match listener's `match` method. Of course, this event expression might never become true in its entirety (as the events required might never occur) since no time constraint (see "Specifying the timer operators" on page 43) has been applied to any part of the event expression.

Defining Event Expressions

Specifying other temporal operators

For a listener to trigger on an event sequence, the event expression defining what to match against must evaluate to `true`.

The `or` operator allows you to specify event expressions where a variety of event sequences could lead to a successful match. It effectively evaluates two event templates (or entire nested event expressions) simultaneously and returns true when either of them become true.

For example,

```
A or B
```

means that either `A` or `B` need to be detected to match. That is, the occurrence of one of the operand expressions (an `A` or a `B`) is enough to satisfy the listener.

The `and` operator specifies an event sequence that might occur in any temporal order. It evaluates two event templates (or nested event expressions) simultaneously but only returns true when they are both true.

```
A and B
```

will seek 'an `A` followed by a `B`' or 'a `B` followed by an `A`'. Both are valid matching sequences, and the listener will seek both concurrently. However, the first to occur will terminate all monitoring and trigger the listener.

The following example code snippets indicate a few patterns that can be expressed using the three operators presented so far.

Example	Meaning
`A -> (B or C)`	Match on an `A` followed by either a `B` or a `C`.
`(A -> B) or C`	Match on either the sequence `A` followed by a `B`, or just a `C` on its own.

Example	Meaning
`A -> ((B -> C) or (C -> D))`	Find an `A` first, and then seek for either the sequence `B` followed by a `C` or `C` followed by a `D`. The latter sequences will be looked for concurrently, but the monitor will match upon the first complete sequence that occurs. This is because the `or` operator treats its operands atomically, i.e. in this case it is looking for the sequences themselves rather than their constituent events.
`(A -> B) and (C -> D)`	Find the sequence `A` followed by a `B` (`A -> B`) followed by the sequence `C -> D`, or else the sequence `C -> D` followed by the sequence `A -> B`. The `and` operator treats its operands atomically—that is, in this case it is looking for the sequences themselves and the order of their occurrence, rather than their constituent events. It does not matter when a sequence starts but it occurs when the last event in it is matched. Therefore {`A',C',B',D'`} would match the specification, because it contains an `A -> B` followed by a `C -> D`. In fact the specification would match against either of the following sequences of event instances; `{A',C',B',D'}`, `{C',A',B',D'}`, `{A',B',C',D'}`, `{C',A',D',B'}`, `{A',C',D',B'}`, and `{C',D',A',B'}`.

The `not` operator is unary and acts to invert the truth value of the event expression it is applied to.

```
A -> B and not C
```

therefore means that the correlator will match only if it encounters an `A` followed by a `B` without a `C` occurring at any time before the `B` is encountered.

Note: The `not` operator can cause an event expression to reach a state where it can never evaluate to true any more, that is, it will become *permanently false*.

Consider this listener event sequence:

```
on (A -> B) and not C
```

The listener will start seeking both `A -> B` and `not C` concurrently. If an event matching `C` is received at any time before one matching `B`, the `C` clause will evaluate to true, and hence `not C` will become `false`. This will mean that `(A -> B) and not C` will never be able to evaluate to `true`, and hence this listener will never trigger. In practice the correlator cleans out these *zombie* listeners periodically.

Note: It is possible to write an event expression that always evaluates to true immediately, without any events occurring.

Consider this listener:

```
on (A -> B) or not C
```

Assuming that `A`, `B`, and `C` represent event templates, their value will start off as being `false`. However, that means that `not C` will become `true` immediately, and hence the whole expression will become `true` right away. This listener will therefore trigger immediately as soon as it is instantiated. If any of `A`, `B` or `C` were nested event expressions the same logic would apply for their own evaluation.

Defining advanced event expressions

Specifying a perpetual listener for repeated matching

So far all the examples given have created listeners that will trigger on the first occurrence of an event (or sequence of events) that satisfies the supplied event expression.

For example,

```
public void onLoad() {
   EventExpression eventExpr = new EventExpression("Tick(*, >10.0)");
   eventExpr.addMatchListener(this);
}
```

locates the *first* occurrence of a `Tick` event that satisfies the `Tick(*, >10.0)` event template. This first suitable event triggers the listener and calls the `match` method of the registered match listener object.

However, you might want to detect *all* `Tick` events that satisfy the above event template (or event expression). To do this you must create a *perpetual* listener, that is, one that does not terminate on the first suitable occurrence, but instead stays alive and triggers repeatedly on every subsequent occurrence.

This effect can be achieved through use of the `all` event expression operator.

If the above is rewritten as follows,

```
public void onLoad() {
   EventExpression eventExpr =
      new EventExpression("all Tick(*, >10.0)");
   eventExpr.addMatchListener(this);
}
```

the listener created will now seek the first `Tick` event whose price is greater than 10. Upon detecting such an event it will trigger and call the `match` method. It will then return to monitoring the incoming event streams to look for the next suitable occurrence. This behavior will be repeated indefinitely until the listener is explicitly deactivated. This means that potentially the `match` method could be invoked multiple times.

Defining advanced event expressions

Deactivating a listener

A listener whose event expression embeds an `all` operator will stay active indefinitely and trigger repeatedly. It will continue doing this until it is explicitly deactivated. This can be done using the `removeMatchListener` method on the `EventExpression` object.

Refer to the Apama API for Java (JMon) reference information provided in Javadoc format for complete class and method signatures (`doc\javadoc\index.html` in your Apama installation directory).

Defining advanced event expressions

Temporal contexts

Imagine that we have seven event templates defined, which for the sake of brevity are represented by the letters `A`, `B`, `C`, `D`, `E`, `F` and `G` in the following text. Now, consider a stream of incoming events, where `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 sequence of incoming events:

```
C1 A1 F1 A2 C2 B1 D1 E1 B2 A3 G1 B3
```

Given the above event sequence, what should the event expression

```
A -> B
```

match upon?

In theory the combinations of events that correspond to "an `A` followed by a `B`" are:

```
{A1, B1}, {A1, B2}, {A1, B3}, {A2, B1}, {A2, B2}, {A2, B3}, {A3, B3}
```

In practice it is unlikely that a developer wanted their monitor to match seven times on the above example sequence, and it is uncommon for all the combinations to be useful.

In fact, consistent with the truth-value based matching behavior already described, the event expression `A -> B` will only match on the first event sequence that matches the expression. Given the above event sequence the listener will trigger only on {`A1`, `B1`}, call the associated `match` method, and then terminate.

If a developer wishes to alter this behavior, and have the monitor match on more of the combinations, they can use the `all` operator within the event expression.

If the listener's specification was rewritten to read:

```
all A -> B
```

the listener would match on 'every `A`' and the first `B` that follows it.

The way this works is that upon encountering an `A`, a second *child* listener (or *sub-listener*) is created to seek for the next `A`. Both listeners would continue looking for a `B` to successfully match the sequence specified. If more `A`'s are encountered the procedure is repeated; this behavior continues until the *master* listener is explicitly deactivated.

Therefore `all A -> B` would match on {`A1`, `B1`}, {`A2`, `B1`} and {`A3`, `B3`}.

Note that `all` is a unary operator and has higher precedence than `->`, `or` and `and`. Therefore `all A -> B` is the same as `(all A) -> B` or `( ( all A ) -> B )`.

The following table illustrates how the execution of `on all A -> B` proceeds over time as the above sequence of input events is processed by the correlator. The timeline is from left to right, and each stage is labeled with a time `tn`, where `tn+1` occurs after `tn`. To the left are listed the listeners, and next to each one (after the `?`) is shown what event template that listener is looking for at that point in time. In the example, assuming `L` was the initial listener, `L'`, `L''` and `L'''` are other sub-listeners that are created as a result of the `all` operator.

Guide to the symbols used:

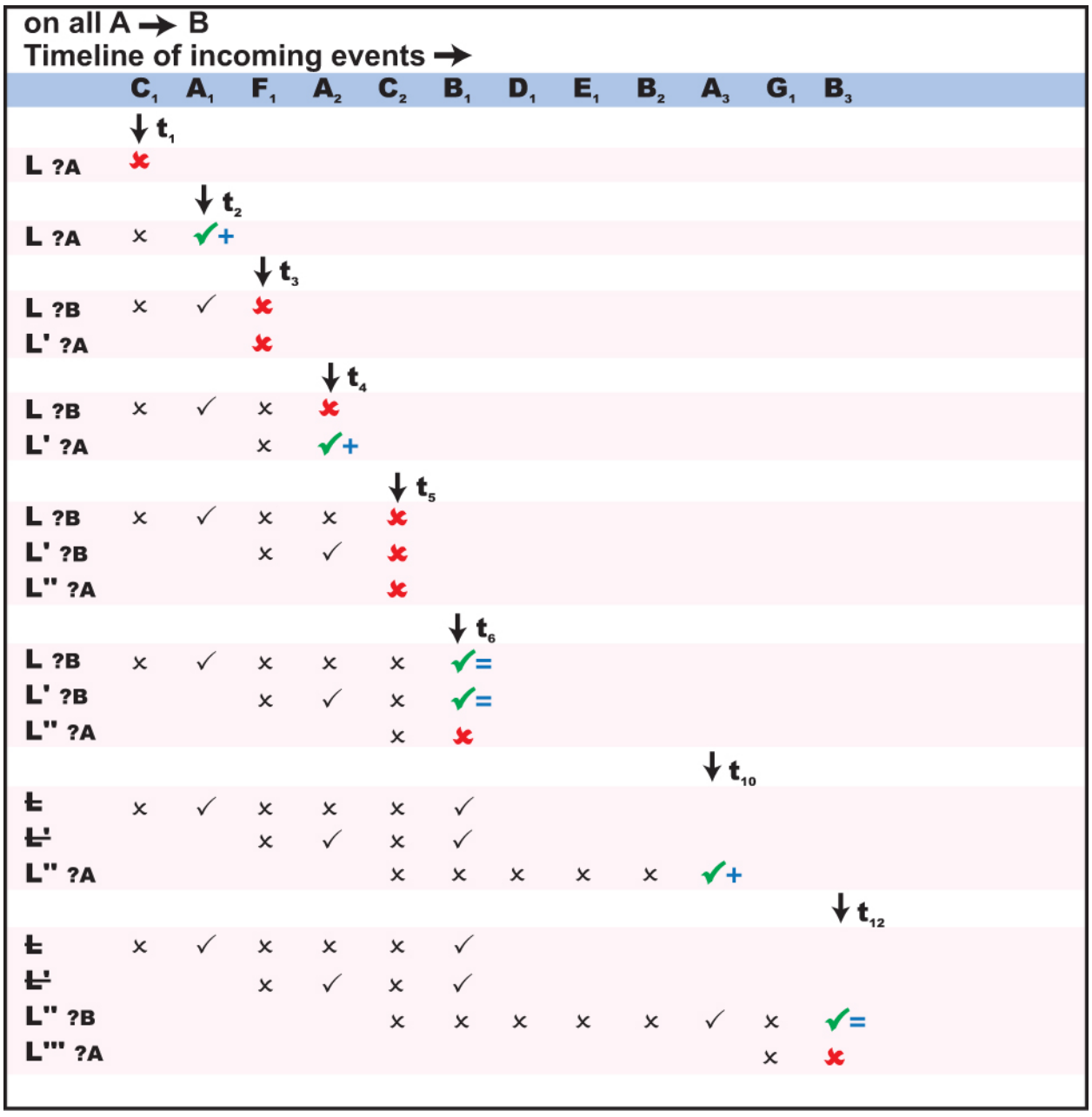
- ↓ indicates a specific point in time when a particular event is received

✱ 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 listener denoted by `all A -> B` will never terminate as there will always be a sub-listener active looking for an `A`.

If, on the other hand, the specification is written as,

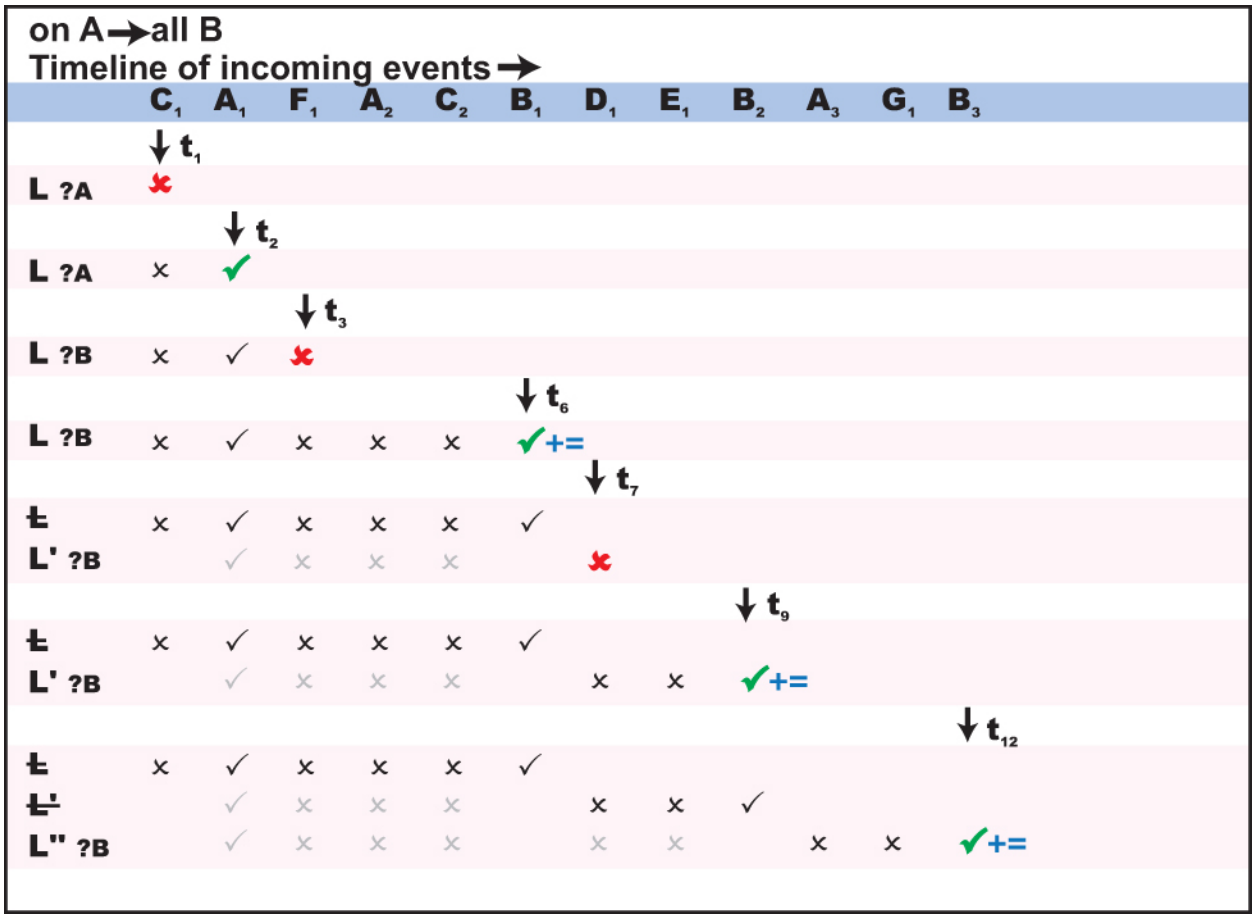
```
A -> all B
```

the listener would now match on all the sequences consisting of the first `A` and each possible following `B`.

The way this works is by creating a second listener upon matching a `B` that then goes on to search for an additional `B`, and so on repeatedly until the listener is explicitly killed.

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

Graphically this would now look as follows:



The table shows the early states of `L'` and `L''` in light color because those listeners actually never really went through those states themselves. However, since they were created as a clone of another listener, it is as though they were.

The master listener denoted by `A -> all B` will never terminate, as there will always be a sub-listener looking for a `B`.

The final permutation is to write the monitor as,

```
all A -> all B
```

Now the listener would match on an `A` and create another listener to look for further `A`'s. Each of these listeners will go on to search for a `B` after it encounters an `A`. However, in this instance all listeners are duplicated once more after matching against a `B`.

The effect of this would be that `all A -> all B` would match {`A1`, `B1`}, {`A1`, `B2`}, {`A1`, `B3`}, {`A2`, `B1`}, {`A2`, `B2`}, {`A2`, `B3`} and {`A3`, `B3`}, i.e. all the possible permutations. This could cause a very large number of sub-listeners to be created.

Note: The `all` operator must be used with caution as it can create a very large number of sub-listeners, all looking for concurrent patterns. This is particularly applicable if multiple `all` operators are nested within each other. This can have an adverse impact on performance.

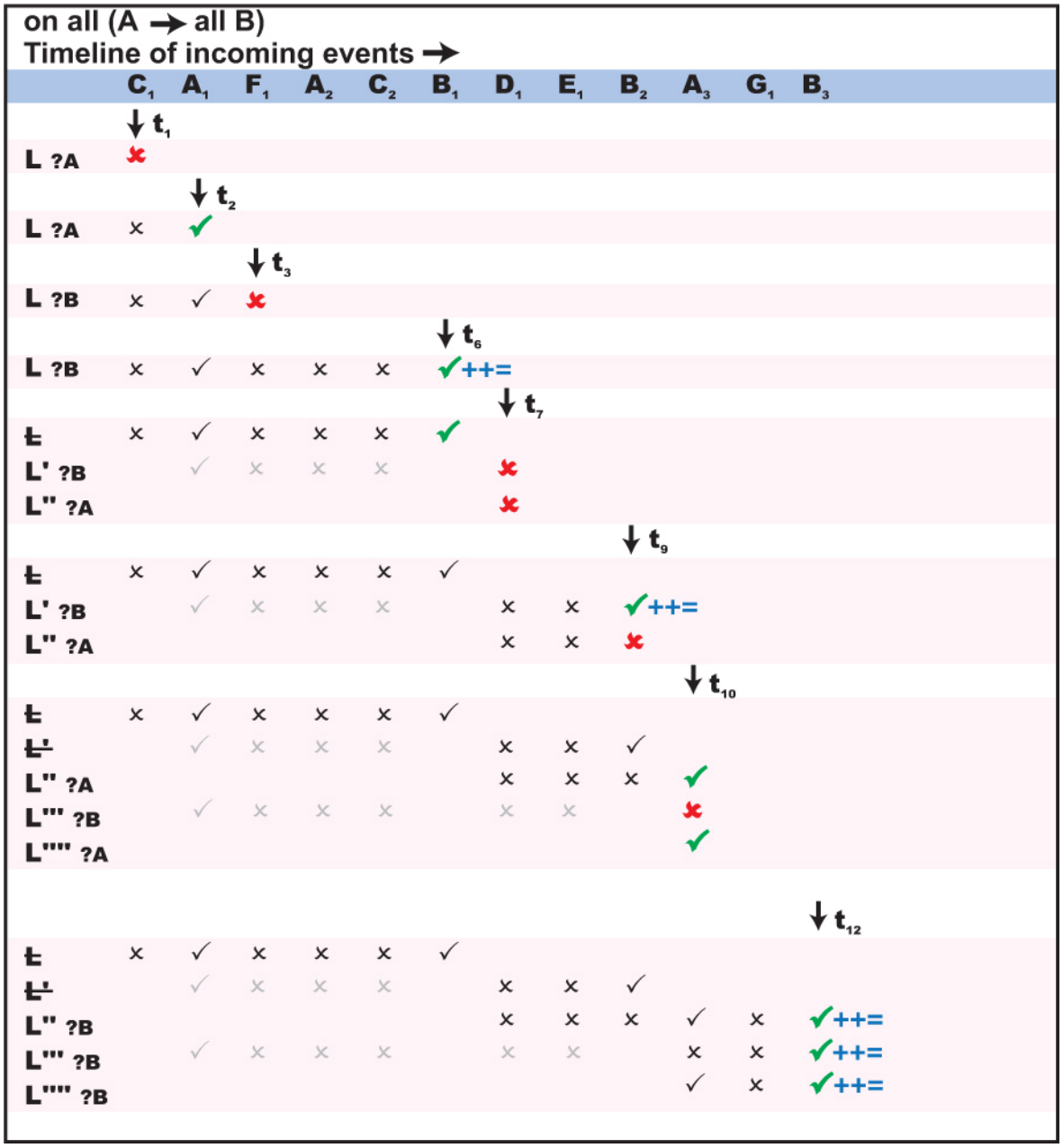
As with all other event expression operators, the `all` operator can be used within nested event expressions, and be nested within the operating context of another `all` operator. This can have a dramatic effect on the number of sub-listeners created.

Consider the example,

```
all (A -> all B)
```

This will match the first `A` followed by all subsequent `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-listeners that might match on the same sequences, thus causing duplicate triggering.

On the same event sequence as previously, graphically, this would be evaluated as follows:



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

Defining advanced event expressions

Specifying the timer operators

So far we have shown how to use event expressions to define interesting sequences of events to look for, where the events of interest depend not only on their type and content, but also on their temporal relationship to (whether they occur before or after) other events.

Being able to define temporal relationships can be useful, but typically it also needs to be constrained over some temporal interval.

Defining advanced event expressions

Looking for event sequences within a set time

Consider this earlier example:

```
// Code within the monitor class
public void onLoad() {
   EventExpression eventExpr = new EventExpression(
      "NewsItem(\"ACME\",*) -> Tick(\"ACME\",*)");
   eventExpr.addMatchListener(matchListener1);
}

// Code within the first (and only) Match Listener
// class - matchListener1

   public void match(MatchEvent event) {
      System.out.println("Detected a NewsItem followed"
         + " by an Tick event, both regarding the ACME company.");
   }
```

This will look for the event sequence of a news item about a company followed by a stock price tick about that company. Once improved this could be used to detect the beginning of a rise (or fall) in the value of shares of a company following the release of a relevant news headline.

However, unless a temporal constraint is put in place, the monitor is not going to be that pertinent, as it might trigger on an event sequence where the price change occurs weeks after the news item. That would clearly not be so useful to a trader, as the two events were most likely unrelated and hence not indicative of a possible trend.

If the event expression above is rewritten as follows,

```
EventExpression eventExpr = new EventExpression(
   "NewsItem(\"ACME\",*) -> Tick(\"ACME\",*) within(30.0)");
```

the `Tick` event would now need to occur within 30 seconds of `NewsItem` for the listener to trigger.

The `within(float)` operator is a postfix unary operator that can be applied to an event expression (the `Tick` event template in the above example). Think of it like a stopwatch. The clock starts ticking as soon as the event expression it is attached to becomes active, i.e. when the listener actually starts looking for it. If the stopwatch reaches the specified figure before the event expression evaluates to `true` the event expression becomes permanently `false`.

In the above code, the timer is only activated once a suitable `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 listener.

As already specified, the `within` operator can be applied to any event expression, hence `A within(x)`, where `A` represents just an event template and `x` is a float value specifying a number in seconds, is perfectly valid.

Specifying the timer operators

Waiting within a listener

The second timer operator available for use within event expressions is `wait(float)`.

`wait` allows you to insert a 'temporal pause' within an event expression. Once activated, a `wait` expression becomes true automatically once a set amount of time passes. For example,

```
A -> wait(x seconds) -> C
```

will proceed as follows; activate the listener and look for the `A` event expression or template, then once `A` becomes true pause (i.e. wait) for `x` seconds, then finally start looking for the `C` event expression or template.

In addition to being part of an event expression, `wait` can also be used on its own,

```
wait(20.0)
```

is a valid event expression in its own right. When its listener activates it just waits for the number of seconds specified (here being `20`), then it evaluates to true and calls any registered `match` methods.

Therefore a `wait` clause starts off being false, and then turns to true once its time period expires. This behavior can be inverted through use of `not`. The expression

```
not wait(20.0)
```

would start off being true, and stay true for 20 seconds before becoming false.

The following,

```
B and not wait(20.0)
```

is an interesting example. It effectively means that this listener will trigger only if a `B` occurs within 20 seconds of its activation. After that the `not wait(20)` clause would become false and prevent the listener from ever triggering.

By using `all` with `wait`, you can easily implement a periodic repeating timer,

```
all wait(5.0)
```

This listener will trigger every 5 seconds and calls any registered `match` methods.

Specifying the timer operators

Working with absolute time

The final temporal operator is the `at` operator. This operator allows you to express temporal activity with regards to absolute time.

The `at` operator allows triggering of a timer:

- At a specific time; for example, 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 and 40 minutes past

The syntax is as follows

```
at(minutes , hours , days_of_the_month , month , days_of_the_week
  [ , seconds ])
```

where the last operand, *`seconds`*, is optional.

Valid values for each operand are as follows:

Timer operand	Values
`minutes`	`0` to `59`, indicating minutes past the hour.

Timer operand	Values
hours	0 to 23, indicating the hours of the day.
days_of_the_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.
month	1 to 12, indicating months of the year, with 1 corresponding to January
days_of_the_week	0 to 6, indicating days of the week, where 0 corresponds to Sunday.
seconds	0 to 59, indicating seconds past the minute.

The operator can be embedded within an event expression in a manner similar to the `wait` operator. If used outside the scope of an `all` operator it will trigger only once, at the *next* valid time as expressed within its elements. In conjunction with an `all` operator, it will trigger at *every* valid time.

The wildcard symbol (`*`) can be specified to indicate that all values are valid, i.e.

```
at(5, *, *, *, *)
```

would trigger at the next "five minutes past the hour", while

```
all at(5, *, *, *, *)
```

would trigger at five minutes past each hour (i.e. every day, every month).

Whereas,

```
all at(5, 9, *, *, *)
```

would trigger at 9:05am every day.

However,

```
all at(5, 9, *, *, 1)
```

would trigger at 9:05am only on Mondays, and never on any other weekday. This is because the effect of the wildcard operator is different when applied to the *days of the week* and the *days of the month* elements. This is due to the fact that both specify the same entity. The rule is therefore as follows:

- As long as both elements are set to wildcard, then each day is valid.
- If either of the *days of the week* or the *days of the month* elements is not a wildcard, then only the days that match that element will be valid. The wildcard in the other element is effectively ignored.
- If both the *days of the week* and the *days of the month* elements are not a wildcard, then the days valid will be the days which match either. That is, the two criteria are '*or*' 'ed, not '*and*' 'ed.

A range operator (`:`) can be used with each element to define a range of valid values. For example

```
all at(5:15, *, *, *, *)
```

would trigger every minute from 5 minutes past the hour till 15 minutes past the hour.

A divisor operator (`/x`) can be used to specify that every `x`'th value is valid. Therefore

```
all at(*/10, *, *, *, *)
```

would trigger every ten minutes, that is, at 0, 10, 20, 30, 40 and 50 minutes past every hour.

If you wish to specify a combination of the above operators you must enclose the element in square brackets ([]), and separate the value definitions with a comma (,). For example,

```
all at([*/10,30:35,22], *, *, *, *)
```

indicates as following values for minutes to trigger on; 0,10, 20, 22, 30, 31, 32, 33, 34, 35, 40 and 50.

A further example,

```
all at(*/30,9:17,[*/2,1],*,*)
```

would trigger every 30 minutes from 9am to 5pm on even numbered days of the month as well as specifically the first day of the month.

Specifying the timer operators

Optimizing event expressions

When a developer creates an event expression, a substantial percentage of the computational overhead goes into parsing the event expression itself.

If you need to create several instances of an event expression where only literal values in event templates vary, this repeated parsing cost can be removed through the use of a *prepared* event expression.

Instead of writing,

```
EventExpression eventExpr1 = new EventExpression(
   "NewsItem(\"ACME\",*) -> Tick(\"ACME\",*)");
EventExpression eventExpr2 = new EventExpression(
   "NewsItem(\"EMCA\",*) -> Tick(\"EMCA\",*)");
eventExpr1.addMatchListener(matchListener1);
eventExpr2.addMatchListener(matchListener2);
```

you could write,

```
PreparedEventExpressionTemplate et
   = new PreparedEventExpressionTemplate(
      "NewsItem(?,*) -> Tick(?,*)");

PreparedEventExpression pex1=et.getInstance();
pex1.setString(0, "ACME");
pex1.setString(1, "ACME");

PreparedEventExpression pex2=et.getInstance();
pex2.setString(0, "EMCA");
pex2.setString(1, "EMCA");

pex1.addMatchListener(matchListener1);
pex2.addMatchListener(matchListener2);
```

The above example shows how instead of creating two very similar event expressions you can create a single prepared event expression template, and then customize multiple instances of it. The main advantage of the latter approach is the fact that the event expression was parsed in Java only once. With an example as simple as the ones above this would in fact hardly make any difference, but in Java code with hundreds of such event expressions the difference in performance can be significant.

As shown in the code snippet above, the procedure for creating listeners with prepared event expressions is slightly different from that of normal event expressions.

You must create a `PreparedEventExpressionTemplate` and define within that the event expression. The syntax for event expression definitions is the same as previously with the exception of the `?` operator. This can be used instead of any literal value. The next step is to get an instance of a `PreparedEventExpression`, and then to set values for any literals replaced by `?` in the prepared event expression template. Finally, you can create listeners on the `PreparedEventExpression` instances just as with normal event expressions.

Defining Event Expressions

Validation of event expressions

When an `EventExpression` or `PreparedEventExpressionTemplate` is created or when `addMatchListener()` is called on an event expression within a JMon monitor the event expression is not validated immediately. It is queued for processing later when the JMon monitor yields control back to the correlator. This means that a badly formed event expression does not cause an exception to be thrown from the constructor. Instead, the correlator logs an error message later when it tries to validate the event expression.

Defining Event Expressions

Chapter 3: The Concept of Time in the Correlator

An understanding of how the correlator handles time is essential to writing EPL applications.

Correlator timestamps and real time

When the correlator receives an event, it gives the event a timestamp that indicates the time that the correlator received the event. The correlator then places the event on the input queue of each public context. The correlator processes events in the order in which they appear on input queues.

An input queue can grow considerably. In extreme cases, this might mean that a few seconds pass between the time an event arrives and the time the correlator processes it. As you can imagine, this has implications for whether the correlator triggers listeners. However, the correlator uses event timestamps, and not real time, to determine when to trigger listeners.

As an extreme example, suppose that the correlator has a monitor that is looking for `A -> B within(2.0)`. The correlator receives event `A`. However, the queue has grown to a huge size and the correlator processes event `A` three seconds after event `A` arrives. The correlator receives event `B` one second after it receives event `A`. Some events in the queue before event `B` cause a lot of computation in the correlator. The result is that the correlator processes event `B` five seconds after event `B` arrives. In short, event `B` arrives one second after event `A`, but the correlator processes event `B` three seconds after it processes event `A`.

If the correlator used real time, `A -> B within(2.0)` would not be triggered by this sequence. 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 listeners. Consequently, `A -> B within(2.0)` does trigger, because the correlator received event `B` one second after event `A`, and so their timestamps are within 2 seconds of each other.

As you can see, the number of events on an input queue never affects temporal comparisons.

The Concept of Time in the Correlator

Event arrival time

As mentioned before, when an event arrives, the correlator assigns a timestamp to the event. The timestamp indicates the time that the event arrived at the correlator.

The correlator uses clock ticks to specify the value of each timestamp. The correlator generates a clock tick every tenth of a second. The value of an event's timestamp is the value of the last clock tick before the event arrived.

When you start the correlator, you can specify the `--frequency`*`hz`* option if you want the correlator to generate clock ticks at an interval other than every tenth of a second. Instead, the correlator generates clock ticks at a frequency of *`hz`* per second. Be aware that there is no value in increasing *`hz`* above the rate at which your operating system can generate its own clock ticks internally. On UNIX and some Windows machines, this is 100 Hz and on other Windows machines it is 64 Hz.

When you start the correlator, you can specify the `-Xclock` option to disable the correlator's internal clock and replace it with externally generated time events. See "Externally generating time events" on page 53.

The Concept of Time in the Correlator

Getting the current time

In the correlator, the current time is the time indicated by the most recent clock tick. There are two exceptions to this:

- If you specify the `-Xclock` option when you start the correlator, the correlator does not generate clock ticks. Instead, you must send time events (`&TIME`) to the correlator. The current time is the time indicated by the most recent externally generated time event. See "Externally generating time events" on page 53.
- When the correlator is firing a timer, the current time is the timer's trigger time. See "About timers and their trigger times" on page 51.

The information in the remainder of this topic assumes that the current time is the time indicated by the most recent clock tick.

Use the static method `double com.apama.jmon.Correlator.getCurrentTime()` to obtain the current time. The value returned by the `getCurrentTime()` method is the current time represented as seconds since the epoch, January 1st, 1970 in UTC.

In the correlator, the current time is never the same as the current system time. In most circumstances it is a few milliseconds behind the system time. This difference increases when public context input queues grow.

When a listener triggers, it causes a call to the listener's `match()` method. The correlator executes the entire method before the correlator starts to process another event. Consequently, while the listener is executing a method, time and the value returned by the `getCurrentTime()` method do not change.

Consider the following code snippet,

```
double a;
void checkTime() {
   a = Correlator.getCurrentTime();
}

// A listener calls the following method some time later
void logTime() {
   System.out.println("a: "+a );
      // The time when checkTime() was called

   System.out.println("current time: "+Correlator.getCurrentTime() );
   // The time now
```

```
}
```

In this code, a method sets `double` variable `a` to the value of `getCurrentTime()`, which is the time indicated by the most recent clock tick. Some time later, a different listener prints the value of `a` and the value of `getCurrentTime()`. The values logged might not be the same. This is because the first use of `getCurrentTime()` might return a value that is different from the second. If the two listeners have processed the same event, the logged values are the same. If the two listeners have processed different events, the logged values are different.

The Concept of Time in the Correlator

About timers and their trigger times

In an event expression, when you specify the `within`, `wait`, or `at` operator you are specifying a timer. Every timer has a trigger time. The trigger time is when you want the timer to fire.

- When you use the `within` operator, the trigger time is when the specified length of time elapses. If a `within` timer fires, the listener fails. In the following listener, the trigger time is 30 seconds after `A` becomes true.

  ```
  A -> B within(30.0)
  ```

  If `B` becomes true within 30 seconds, the trigger time for the timer is not reached, the timer does not fire, the listener triggers, and the monitor calls any attached JMon listeners. If `B` does not become true within 30 seconds, the trigger time is reached, the timer fires, and the listener fails. The monitor does not call the `MatchListener`.

- When you use the `wait` operator, the trigger time is when the specified pause during processing of the event expression has elaspsed. When a `wait` timer fires, processing continues. In the following expression, the trigger time is 20 seconds after `A` becomes true. When the trigger time is reached, the timer fires. The listener then starts watching for `B`. When `B` is true, the monitor calls any attached listeners.

  ```
  A -> wait(20.0) -> B
  ```

- When you use the `at` operator, the trigger time is one or more specific times. An `at` timer fires at the specified times. In the following expression, the trigger time is five minutes past each hour every day. This timer fires 24 times each day. When the timer fires, the monitor calls any attached JMon listeners.

  ```
  all at(5, *, *, *, *)
  ```

At each clock tick, the correlator evaluates each timer to determine whether that timer's trigger time has been reached. If a timer's trigger time has been reached, the correlator fires that timer. When a timer's trigger time is exactly at the same time as a clock tick, the timer fires at its exact trigger time. When a timer's trigger time is not exactly at the same time as a clock tick, the timer fires at the next clock tick. This means that if a timer's trigger time is .01 seconds after a clock tick, that timer does not fire until .09 seconds later.

When a timer fires, the current time is always the trigger time of the timer. This is regardless of whether the timer fired at its trigger time or at the first clock tick after its trigger time.

A single clock tick can make a repeating timer fire multiple times. For example, if you specify `all wait(0.01)`, this timer fires 10 times every tenth of a second.

Because of rounding constraints,

- A timer such as `all wait(0.1)` drifts away from firing every tenth of a second. The drift is of the order of milliseconds per century, but you can notice the drift if you convert the value of the current time to a string.
- Two timers that you might expect to fire at the same instant might fire at different, though very close, times.

  The rounding constraint is that you cannot accurately express `0.1` seconds as a float because you cannot represent it in binary notation. For example, the `on wait(0.1)` listener waits for `0.10000000000000000555` seconds.

To specify a timer that fires exactly 10 times per second, calculate the length of time to wait by using a method that does not accumulate rounding errors. For example, calculate a whole part and a fractional part:

```
@Application(author="Tim Berners", company="Apama",
description="Demonstrate tenth of second timers", name="Tenth",
version="1.0")
@MonitorType
public class TenthOfSecond implements Monitor {

   private static final Logger LOGGER =
      Logger.getLogger(TenthOfSecond.class);
   private static final NumberFormat formatter =
      NumberFormat.getInstance();
   static { formatter.setGroupingUsed(false); }

   double startTime;
   double fraction;

   public void onLoad() {
      startTime = Math.ceil( Correlator.getCurrentTime() );
      fraction = Math.ceil(
         (Correlator.getCurrentTime() - startTime) * 10.0);
      setupTimeListener();
   }
   void setupTimeListener() {
      fraction++;
      if (10.0 <= fraction) {
         fraction = 0.0;
         startTime++;
      }
      EventExpression ee = new EventExpression("wait("+ ((startTime +
         (fraction / 10.0))-Correlator.getCurrentTime()) +")");
      ee.addMatchListener(new MatchListener() {
         public void match(MatchEvent evt) {
            LOGGER.info(formatter.format(Correlator.getCurrentTime()));
            // System.out.println(Correlator.getCurrentTime());
            // This would go to STDOUT, and isn't as pretty
            new TestEvent(Correlator.getCurrentTime()).emit();
            setupTimeListener();
         }
      });
   }
}
// TenthOfSecond
```

When a timer fires, the correlator processes items in the following order. The correlator

1. Triggers all listeners that trigger at the same time.
2. Routes any events, and routes any events that those events route, and so on.
3. Fires any timers at the next trigger time.

The Concept of Time in the Correlator

Externally generating time events

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 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.

The Concept of Time in the Correlator

About &TIME events

`&TIME` events have the following format:

```
&TIME(floatseconds)
```

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 1012, 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 the internal time (the current time) in that context to the value encoded in the event. Every event that the correlator processes after an `&TIME` event and before the next `&TIME` event has the same timestamp. That is, they have the timestamp indicated by the value of the previous `&TIME` event. For example:

```
&TIME(1)
A()
B()
&TIME(2)
C()
```

Events `A` and `B` each hava 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.

Externally generating time events

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 sequence:

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 sequence 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.

Externally generating time events

About &SETTIME events

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 sequence 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.

For information about when you might want to use external time events, see *Deploying and Managing Apama*, "Determining whether to disable the correlator's internal clock".

Externally generating time events

Chapter 4: Developing and Deploying JMon Applications

This section describes the steps required to develop and deploy a JMon application. You can develop JMon applications in Apama Studio (recommended) or manually, outside Apama Studio. When you use Apama Studio some development steps are performed automatically for you. This section describes all development steps and notes which steps Apama Studio automatically performs.

For more information on developing JMon applications, see, "Working with projects" in *Using Apama Studio*.

Steps for developing JMon applications in Apama Studio

The procedure for developing JMon applications in Apama Studio is as follows:

1. Add Java support to a project.

   To create a new project with Java support:

   a. Select File > New > Apama Project.

   b. Enter a project name and click Next.

   c. At the bottom of the dialog, select Add Apama Java support and click Finish.

   To add Java support to an existing project:

   a. Right click the project in the Project Explorer panel.

   b. Select Apama > Add Apama Java Nature.

   When a project has Java support, Apama Studio does the following:

   - Uses `ApamaJavaLibrary` to add all necessary `JAR` files to the project's Java build path.
   - Creates and updates the `jmon-jar.xml` file. This is the deployment descriptor file required by each JMon application. Inside the correlator, the JVM processes the deployment descriptor file and uses it as a guide to the event types and monitor classes to load.

- Generates and maintains your application's JMon JAR file in the *project_name* java application files folder of your project.

If you select Apama > Add Java Nature you are adding standard Eclipse Java support to your project. Selection of Add Apama Java Nature includes standard Eclipse Java support.

2. Create your application's source files.

   Select File > New > Java Event or select File > New > Java Monitor.

   Or, in the Project Explorer, right-click your project and select New > Java Event or select New > Java Monitor.

   A wizard appears that lets you specify the event or monitor's name, the package, a description, the Java source folder and Java package. Apama Studio automatically adds an entry for the event or monitor to the jmon-jar.xml deployment descriptor file and regenerates the JMon JAR file to include the new event or monitor.

   If you want to build your JAR files manually, right-click your project and select Apama > Build JAR Files. This is useful if you unselected the Build jar files automatically option in the apama_java.xml file, which is in the config directory of your project. One reason you might not want to build the JAR files automatically is that the build takes too long. When Build jar files automatically is selected, Apama Studio builds the JAR files every time you modify a JMon file.

   If there are events that you defined in JMon and you refer to those events, or listen for those events in EPL code, then you must define those events in EPL as well as JMon. If you do not also define the events in EPL, Apama Studio flags EPL references to those events as errors.

   See also "Creating new files for JMon applications" in *Using Apama Studio*.

3. Create your application's launch configuration.

   Apama Studio adds all JMon JAR files to the correlator initialization list and all non-JMon JAR files to the correlator class path.

   If you want to build your project's files outside Apama Studio and Eclipse right-click your project and select Apama > Generate Ant Buildfiles. Apama Studio generates an ant build file (with the name build-*project-name*.xml), which you can use only to build your project's JMon JAR files outside of the Eclipse environment. Note that this is unrelated to the Apama Studio feature for exporting an Ant build file that you can use for deployment.

   See "Defining custom launch configurations" in *Using Apama Studio*.

4. Run and test your application.

   See "Launching Projects" in *Using Apama Studio*.

5. Debug your application.

   See "Debugging JMon Applications" in *Using Apama Studio*.

6. Deploy your application.

   See "Deploying JMon applications" on page 58.

   Apama Studio generates your application's JMon JAR file in the *jmon_config_name* java application files folder of your project's directory. By default, *jmon_config_name* is the project name.

You can manage the content of the JMon JAR file and jmon-jar.xml file by using the Apama Studio editor to update the apama_java.xml file, which is located in the project's config folder. You can use the Apama Studio editor to do the following:

- Set JMon metadata.
- Set the injection order of the events and monitors.
- Add non-JMon Java classes to the JMon JAR files.
- Add JMon classes that were not created by Apama Studio wizards to the JMon JAR file.

Developing and Deploying JMon Applications

Java prerequisites

The following are Java prerequisites for using Apama's JMon API. Apama Studio includes the required Java compiler.

- For development of JMon applications:

  While Apama installation includes the Java Runtime Environment (JRE), it does not include a compiler or the jar utility. To write and compile JMon applications, you must have installed a Java development kit (JDK) on your system. In particular, you need a Java compiler, such as javac, and the jar utility.

  You can download these from the following locations:

  http://www.oracle.com/technetwork/java/javase/downloads/index.html

  Apama includes Oracle JRE 7.0 and it is recommended that you use JDK 7.0 to develop, build, and test your applications. Use of any JRE other than the one that Apama ships with is discouraged.

- For deployment of JMon applications

  When you start the correlator and specify the -j option, the correlator starts a Java Virtual Machine. It uses the first JRE/JDK in the PATH environment variable.

  For deployed applications, it is highly recommended that you use JRE 6.0 update 27, which is the version that Apama ships with.

When you install Apama, the installation script installs the JMon API as correlator_extension_api5.2.jar in the Apama lib directory.

Developing and Deploying JMon Applications

Steps for developing JMon applications manually

The procedure for developing JMon applications outside Apama Studio is as follows:

1. Ensure that correlator_extension_api5.2jar is in your Java CLASSPATH environment variable.
2. Create a folder in which to develop your application.

3. In this development folder, define one `.java` file for each event type and one `.java` file for each monitor class.
4. Ensure that there is a deployment descriptor file named `jmon-jar.xml`. See "Creating deployment descriptor files" on page 59.
5. In your development folder, compile all your Java source code.

   ```
   javac *.java
   ```

   If `correlator_extension_api5.2.jar` is not already in your `CLASSPATH` environment variable, you can specify the `-classpath` command-line option to point to `correlator_extension_api5.2.jar`.
6. In your development folder, create a `JAR` file that contains the deployment descriptor and all class files. The command line format is as follows:

   ```
   jar -cf application_name.jar META-INF/jmon-jar.xml *.class
   ```

   Replace *`application_name`* with a name you choose for your application. On Windows, use backslashes "\" instead of forward slashes "/".

   If your application uses an event type definition class that is also used by another JMon application, you must include the event type definition class in the `JAR` file of each application that uses it. If you do not include a shared event type definition class in your application's `JAR` file, injection fails with an `ApplicationVerificationException`.

   You cannot specify the location of a shared event type definition class in your `CLASSPATH` environment variable. The correlator uses a separate classloader for each application, and it cannot use the system classloader for event type definition classes.
7. If any of your application's `.class` files are in your `CLASSPATH` environment variable, remove them. If the JRE can resolve a class path by using either your application's `JAR` file or your `CLASSPATH` environment variable, Apama fails to load your application.

Developing and Deploying JMon Applications

Deploying JMon applications

To deploy and run your application outside Apama Studio:

1. Start a correlator with Java enabled:

   ```
   correlator -j other_options
   ```
2. Inject the application `JAR` file:

   ```
   engine_inject -j application_name.jar
   ```

   Apama creates an object instance of each monitor class defined in the deployment descriptor file and executes its `onLoad` method. If there are multiple monitor classes, they are injected in the order in which they are specified in the `jmon-jar.xml` file.

   The classes in the application's `JAR` file cannot also exist (have the same packaging and name) anywhere else on the classpath. If they do, it causes the application to fail to load.

   When you start the correlator, you can pass properties and options to the embedded JVM with the `-J` option. Specify the `-J` option with each property or option you want to specify. However, you cannot use this mechanism to pass the classpath to the JVM. The correlator sets the classpath

implicitly as the last option, which overrides any value you might have set. Set the CLASSPATH environment variable if you want to set a particular classpath.

Developing and Deploying JMon Applications

Removing JMon applications from the correlator

To stop and delete a running JMon application, execute the engine_delete operation:

```
engine_delete [options_to_identify_correlator]application_name
```

If the application you want to delete is not running on the local host on the default correlator port, be sure to specify options that indicate the correlator that is running the application you want to delete.

Replace *application_name* with the name of the application as specified in the deployment descriptor. This is not necessarily the same as the name of the application's JAR file.

Deleting a JMon application does the following:

- Terminates the application's active listeners.
- Deletes the application's monitor classes.
- Leaves the event type definitions loaded in the correlator. To remove the event type definitions, execute engine_delete and specify the files that contain the event type definitions.

Developing and Deploying JMon Applications

Creating deployment descriptor files

The JMon application's JAR file must contain a deployment descriptor file. Inside the correlator, the JVM processes the application's deployment descriptor file and uses it as a guide to the event types and monitor classes to load. The name of the deployment descriptor file must be jmon-jar.xml.

When you use Apama Studio Java support to develop your JMon application, Apama Studio generates the deployment descriptor file for you. If you develop your JMon application outside Apama Studio, there are two ways to create a deployment descriptor file:

- Manually write the deployment descriptor file. Use your favorite editor to create this XML file according to the "Format for deployment descriptor files" on page 60.
- Insert Java annotations in your source files and run a utility to generate the deployment descriptor file. The annotations you can insert are defined in the java.apama.jmon.annotation package.

Of course, you can use the utility to generate the deployment descriptor file and then manually edit the result. If you then run the utility again, you would lose any manual changes you had made.

The technique you use is largely a matter of personal preference — hand-coded or machine-generated. If you have a very large application with many event types and monitors, you might prefer to insert the annotations and generate the deployment descriptor file. If you have a small application, you might find it easier to write the deployment descriptor file.

To help you create deployment descriptor files, this section discusses the following topics:

- "Format for deployment descriptor files" on page 60

Format for deployment descriptor files

The format of the deployment descriptor file must be compliant with the XML defined by the following XML Document Type Definition (DTD):

`http://www.apama.com/dtd/jmon-jar_1_0.dtd`

You should become familiar with this DTD to understand the exact definition of the deployment descriptor file. However, the normal structure of the file is as follows. In the following format, all text inside XML element tags, *`which is in italic typeface`*, indicates placeholders for which you would supply an actual value.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jmon-jar PUBLIC "-//Apama, Inc.//DTD Java Monitors 1.0//EN"
  "http://www.apama.com/dtd/jmon-jar_1_0.dtd">

<jmon-jar>

  <name>Application Name in the Correlator</name>
  <version>Version Number</version>
  <author>Author</author>
  <company>Company Name</company>
  <description>Description of this application</description>
  <application-classes>

    <event>
      <event-name>Event Type name in the Correlator</event-name>
      <event-class>Event Type's class location</event-class>
      <description>Description of Event Type</description>
    </event>

    <monitor>
      <monitor-name>Monitor's name in the Correlator</monitor-name>
      <monitor-class>Monitor's class location</monitor-class>
      <description>Description of Monitor class</description>
    </monitor>
  </application-classes>
</jmon-jar>
```

The most important part of the deployment descriptor file is the `application-classes` element. This element must contain an `event` element for each event type your JMon application defines. It must also contain a `monitor` element for each monitor your JMon application defines.

The application name that you specify in the `name` element is important because it defines the JMon application's name in the correlator. The `engine_inspect` management tool displays this name when it lists data for your application. If you want to delete your application, you specify this name. The application name must be unique across all currently loaded applications. If the application name is not unique, injection fails.

Creating deployment descriptor files

Defining event types in deployment descriptor files

The deployment descriptor file must define an `event` element for each event type class in your JMon application's `JAR` file. Each `event` element must contain the following two elements:

- `event-name` — The name by which this event type is to be defined within the correlator. The correlator has a single namespace. Consequently, this name must be unique across *all* applications. For example, `Tick` or `SimpleApp.Tick`. If you specify a package qualified name, it is the qualified name that must be unique.
- `event-class` — The name of the Java class in which this event type is defined. This must correspond to the fully qualified name of the class, for example, `Tick` if the event type class is defined within the default Java package, or `com.apama.example.types.Tick` if the event type class is defined in the `com.apama.example.types` package. The file, for example, `Tick.java`, is expected to be located within a folder structure that maps to the packaging, as per standard Java convention.

The `event` element can optionally contain a third element. This is the `description` element. Specify a description of the event type. For example:

```
<event>
   <event-name>Tick</event-name>
   <event-class>Tick</event-class>
   <description>Event that signals a stock trade</description>
</event>
```

JMon and EPL share a single namespace for event types. After an event type is loaded into the correlator, using either JMon or EPL, it is available for use in either environment. However, within a JMon application, you cannot instantiate variables of an event type defined in EPL.

When you try to inject an event type definition that has the same name as a loaded event type, the correlator checks whether the two definitions are duplicates. If they are, the correlator ignores the duplicate you are trying to load. If the definitions are different, the correlator generates an injection error.

Creating deployment descriptor files

Defining monitor classes in deployment descriptor files

The deployment descriptor file must define a `monitor` element for each monitor class in your JMon application's `JAR` file. Each `monitor` element must contain the following two elements:

- `monitor-name` — The name by which this monitor is to be defined within the correlator. The correlator has a single namespace. Consequently, this name must be unique across *all* applications. For example, `SimpleMon` or `SimpleApp.SimpleMon`. If you specify a package qualified name, it is the qualified name that must be unique.
- `monitor-class` — The name of the Java class in which this monitor is defined. This must correspond to the fully qualified name of the class, for example, `SimpleMon` if the monitor class is defined within the default Java package, or `com.apama.example.monitors.SimpleMon` if the monitor class is defined in the `com.apama.example.monitors` package. The file, for example, `SimpleMon.java`, is

expected to be located within a folder structure that maps to the packaging, as per standard Java convention.

The `monitor` element can optionally contain a third element. This is the `description` element. Specify a description of the monitor. For example:

```
<monitor>
   <monitor-name>Simple</monitor-name>
   <monitor-class>Simple</monitor-class>
   <description>A simple JMon monitor, used to show functionality of
      a new installation.</description>
</monitor>
```

Creating deployment descriptor files

Inserting annotations for deployment descriptor files

In your JMon source files, you can specify the following annotations:

- `@Application` — This annotation indicates the name of the application, as well as the author, version, company, and description of the application. Insert this annotation in any one, and only one, of your JMon source files. Each value is required. This annotation must be after any `import` statements and before the class definition statement. For example:

  ```
  @Application(
     name = "Simple",
     author = "Moray Grieve",
     version = "1.0",
     company = "Apama",
     description = "Deployment descriptor for a simple JMon monitor")
  ```

- `@MonitorType` — This annotation indicates the definition of a monitor. In each monitor class, insert this annotation immediately before the monitor class definition statement. You can specify a name and a description for the monitor. The name is the fully qualified EPL name for the monitor. If you do not specify a name, the name defaults to the fully qualified JMon class name of the class you are annotating.

  ```
  @MonitorType(description = "A simple JMon monitor, used to show
       functionality of a new installation.")
  ```

- `@EventType` — This annotation indicates the definition of an event type. In each event type definition class, insert this annotation immediately before the definition statement for the event type. You can specify a name and a description for the event. The name is the fully qualified EPL name for the event. If you do not specify a name, the name defaults to the fully qualified JMon class name of the class you are annotating. For example:

  ```
  @EventType(description = "Event that signals a stock trade")
  ```

- `@Wildcard` — This annotation indicates a wildcard event field. Insert it immediately before the field definition statement. You must have specified the `@EventType` annotation for the event type that defines this field. For example:

  ```
  import com.apama.jmon.*
  import com.apama.jmon.annotation.*

  @EventType
  public class EventWithWildcard extends Event {
     public long indexedField;
     @Wildcard
     public long wildcardField;
     public EventWithWildcard() {
        this(0, 0);
  ```

```
    }
    public EventWithWildcard(long iField, long wField) {
       this.indexedField = iField;
       this.wildcardField = wField;
    }
```

Creating deployment descriptor files

Sample source files with annotations

Following are two sample source files with annotations. These are the source files for the `simple` sample application provided with Apama. The lines with the annotations are in bold typeface for your convenience.

Here is the `Simple.java` file with comments removed:

```
import com.apama.jmon.*;
import com.apama.jmon.annotation.*;

@Application(name = "Simple",
    author = "Moray Grieve",
    version = "1.0",
    company = "Apama",
    description = "Deployment descriptor for the Simple JMon monitor")

@MonitorType(description = "A simple JMon monitor, used to show
    functionality of a new installation.")
public class Simple implements Monitor, MatchListener {

   public Simple() {}

   public void onLoad() {
      EventExpression eventExpr = new EventExpression(
         "all Tick(*, >10.0):t");
      eventExpr.addMatchListener(this);
   }
   public void match(MatchEvent event) {
      Tick tick = (Tick)event.getMatchingEvents().get("t");
      tick.emit();
   }
}
```

Here is the `Tick.java` file with comments removed:

```
import com.apama.jmon.Event;
import com.apama.jmon.annotation.*;

@EventType(description = "Event which signals a stock trade")
public class Tick extends Event {

   public String name;
   public double price;
   public Tick() {
      this("", 0);
   }

   public Tick(String name, double price){
      this.name = name;
      this.price = price;
   }
}
```

Creating deployment descriptor files

Generating deployment descriptor files from annotations

There are two utilities that you can use to generate the deployment descriptor file from annotations in your source files:

- `com.apama.jmon.annotation.DirectoryProcessor` — This utility processes a directory and generates the deployment descriptor file, which you must add to your application's `JAR` file.
- `com.apama.jmon.annotation.JarProcessor` — This utility processes an application's `JAR` file and adds the deployment descriptor file to that `JAR` file.

You can execute these utilities from the command line or from a Java build file.

The `DirectoryProcessor` utility takes three optional arguments:

- `-r` indicates that you want to recursively process the `.class` files in each directory and subdirectory in the specified directory. The default is that the utility processes only the `.class` files that are in the specified directory.
- `-d` specifies the directory that contains the `.class` files you want to process. The default is that the utility processes any `.class` files in the current working directory.
- `-o` specifies the file in which to store the output. The default is that output goes to `stdout`. In the JMon application `JAR` file, the name of the deployment descriptor file must always be `jmon-jar.xml`.

After you generate the deployment descriptor file, you must place it in the `META-INF` directory of your development directory. For example, you can execute the `DirectoryProcessor` utility from the command line as follows:

```
cd src
javac -classpath
$APAMA_CORRELATOR_HOME/lib/correlator_extension_api$APAMA_LIBRARY_VERSION.jar
*.java
java -DAPAMA_LOG_LEVEL=WARN -classpath
$APAMA_CORRELATOR_HOME/lib/correlator_extension_api$APAMA_LIBRARY_VERSION.jar
com.apama.jmon.annotation.DirectoryProcessor -r -d ./src -o
./src/META-INF/jmon-jar.xml
jar -cf ../simple-jmon.jar META-INF/jmon-jar.xml *.class
```

The `JarProcessor` utility takes one required argument, which is the name of the `JAR` file to operate on. To execute the `JarProcessor` utility from a Java build file, you can define something like the following:

```
<!--Target to process the annotations in the JMon application classes
    to produce jmon-jar.xml -- the deployment descriptor file.
-->
<target name="process-jar" depends="jar">
  <echo message=
    "Process annotations in jar file: ${process-jar-file}" />
  <java jvm="java"
    classname="com.apama.jmon.annotation.JarProcessor" dir="."
    fork="yes">
    <classpath>
      <fileset dir="${lib-dir}">
        <patternset refid="libs" />
      </fileset>
    </classpath>
    <jvmarg value="-DAPAMA_LOG_LEVEL=WARN" />
    <arg value="${process-jar-file}" />
  </java>
</target>

<target name="process" depends="jar">
```

```
  <antcall target="process-jar">
    <param name="process-jar-file" value="${jar-file}" />
  </antcall>
</target>
```

Creating deployment descriptor files

Package names and namespaces in JMon applications

There is no correlation between the correlator namespace defined for a named JMon event or monitor, and the Java package structure of the class file in which that event or monitor is implemented. Event expressions are based on the correlator namespace, not on the Java package of the implementation.

Consider the following example. An event type defined in a Java class `a.b.c.MyEvent` that is given the correlator name `x.MyEvent`. Also a monitor defined in a Java class `a.b.c.MyListener` that is given the correlator name `y.MyListener`. Now, although the two classes are in the same Java package and need not use import statements to see each other, their correlator names are in different namespaces. This means that an event expression in `y.MyMonitor` will need to use the fully qualified name `x.MyEvent` to refer the event.

Developing and Deploying JMon Applications

Sample JMon applications

The Apama distribution includes a number of complete sample applications. These applications are in the `samples` folder under `java_monitor`, and are called `simple`, `stockwatch`, `vwap`, `dos`, `context` and `complex`.

See the `README.txt` file included with each sample for complete instructions for how to compile and run the sample application.

Developing and Deploying JMon Applications