Chapter

8

Using Scripting in a Report Design

BIRT provides a powerful scripting capability with which a report developer can provide custom code to control various aspects of report generation.

Overview of BIRT scripting

When developing a BIRT report using the Eclipse workbench, you can write custom event handlers in either Java or JavaScript. When developing a BIRT report using the Eclipse RCP, you can write only JavaScript event handlers. Whether you use Java or JavaScript, the set of event handlers that you can write is the same.

Choosing between Java and JavaScript

Both Java and JavaScript have advantages and disadvantages when writing an event handler. For a developer who is familiar with only one of the two languages, the advantage of using the familiar language is obvious but for all others, the decision depends on the report requirements.

The advantages of using JavaScript to write an event handler include:

- Ease of adding a simple script for a particular event handler

  Adding a JavaScript event handler to a report is less complicated than adding a Java event handler. When writing a JavaScript event handler, there is no need to create a Java environment in Eclipse or to learn the Eclipse Java development process. You are not required to specify a package, implement an interface, or know the parameters of the event handler you write.

To add a JavaScript event handler, you type the code for the event handler on the Script tab after selecting the name of the event handler from a drop-down list.

- Simpler language constructs, looser typing, and less strict language rules

  JavaScript is less demanding to code than Java due to simpler language constructs, looser typing, and less strict language rules.

The advantages of using Java to write an event handler include:

- Availability of the Eclipse Java development environment

  The Eclipse Java development environment is very powerful, and includes such features as autocompletion, context sensitive help, keyboard shortcuts, parameter hints, and much more.

- Ease of finding and viewing event handlers

  All the Java event handlers for a report exist in readily viewable Java files. By contrast, the JavaScript event handlers are embedded in the design and you can view only one handler at a time.

- Access to an integrated debugger

  The integrated debugger only supports Java event handlers, not JavaScript event handlers.

Using both Java and JavaScript to write event handlers

You are not limited to writing all event handlers in one language. You can write some in Java and others in JavaScript. If you have both a Java and a JavaScript event handler for the same event, BIRT uses the JavaScript handler.

Understanding the event handler execution sequence

This section explains the order of execution of the BIRT event handlers.

About event firing sequence dependency

The event firing sequence for ReportItem and ReportDesign events depends on whether the report is run in the BIRT Report Designer previewer or elsewhere, such as in the Web Viewer. When a report runs outside the previewer, the generation phase always completes before the presentation phase begins. When a report runs in the previewer, the generation and presentation phases are not distinctly separated. The onCreate event is a generation-time event and the onRender is a presentation-time event.

About the onCreate and onRender firing sequence dependencies

When a report runs in the BIRT Report Designer previewer, all ReportItem onRender events fire immediately after their corresponding onCreate events. When a report runs outside the previewer, all onCreate events fire as a part of the generation process, while the onRender events fire as a part of the presentation process.

About the ReportDesign firing sequence dependencies

When a report runs in the previewer, the ReportDesign initialize event fires only once, and is always the first event fired. When a report runs outside the previewer, the initialize event is fired twice, once at the beginning of the generation phase and once at the beginning of the presentation phase.

The ReportDesign beforeRender and afterRender events also fire at different times, depending on whether the report runs in the previewer. When a report runs in the previewer, beforeRender fires once near the start of the report, just after beforeFactory fires. The ReportDesign afterRender event fires once, near the completion of the report, just before the afterFactory event fires.

When the report runs outside the previewer, the ReportDesign beforeRender event fires once, immediately after the firing of the initialize event in the presentation phase. When the report runs outside the previewer, the ReportDesign afterRender event is the last event fired.

About the pageBreak event

Table and Text objects have event handlers for handling page break events. The pageBreak event is fired in the presentation phase whenever a page break occurs.

Analysis of the execution sequence phases

The following diagrams present a more detailed view of the event handler execution sequence. The diagrams reflect the processing sequence when a report is run inside the previewer. When the presentation phase is separate from the generation phase, as it is when a report is run outside the previewer, an additional rendering sequence occurs. The rendering sequence is identical to the generation sequence with the following exceptions:

- There are no onPrepare events.
- There are no onCreate events.
- There are no data source or data set events because data is retrieved from the report document rather than the database.
- There are no beforeFactory and afterFactory events.

Overview of the report execution process

Figure 8-1 shows an overview of the report execution process. Each box in the diagram refers to another diagram that appears later in the chapter.

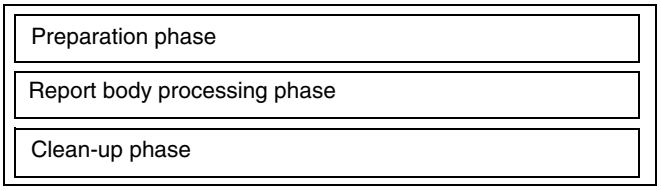

Figure 8-1 Method execution phases

Preparation phase

The preparation phase includes initialization and master page creation, followed by opening the data source. The preparation phase is identical for all reports. Figure 8-2 illustrates the method execution sequence for the preparation phase.

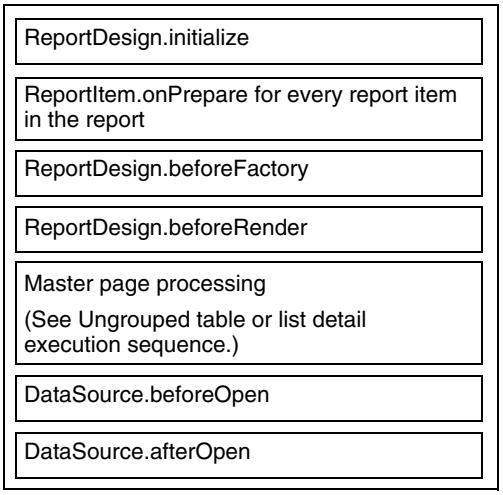

Figure 8-2 Preparation phase

In Figure 8-2, the master page processing sequence depends on the structure of the report. A master page typically consists of one or more header and footer grids with rows, their cells, and the cell contents. The execution sequence for a master page parallels that for creating an ungrouped table, except that the master page contains no detail rows.

Report body processing phase

BIRT processes a report body by processing all the report items that are not contained in other report items. BIRT processes the items, going from left to right and proceeding a row at a time toward the bottom right. A report item that

is not contained in another report item is called a top-level report item. Every report has at least one top-level report item, usually a grid, a list, or a table. If a report has more than one top-level report item, BIRT processes the top-level items in order, from left to right and top to bottom.

For each top-level item, BIRT processes all the second-level items before proceeding to the next top-level item. A second-level report item is a report item that is contained within a top-level item. For example, a table contained in a grid is a second-level report item.

There can be any number of levels of report items. To see the level of a particular report item, examine the structure of the report design in Outline, as shown in Figure 8-3.

BIRT processes all items at all levels in an iterative fashion, following the same process at each level as it does for the top-level items.

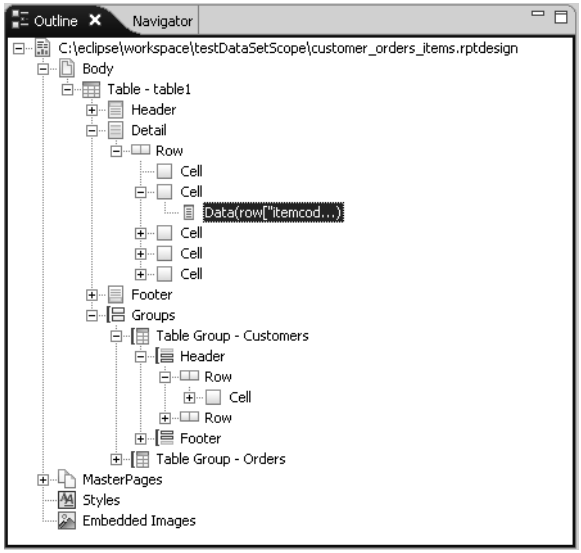

Figure 8-3 The Outline window, showing the level of a report item

Figure 8-4 illustrates the general report body processing phase.

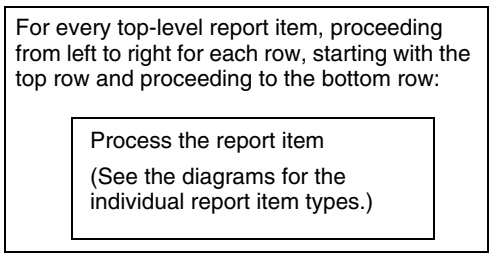

Figure 8-4 **Report body processing phase**

Clean-up processing phase

The clean-up phase consists of two methods that execute upon closing the data source, followed by a final method that executes after the generation phase. Figure 8-5 illustrates the method execution sequence for the clean-up phase.

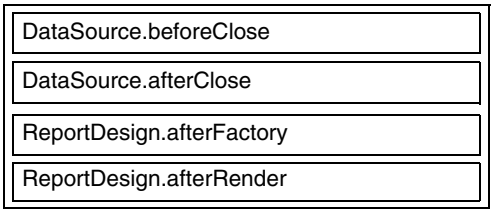

Figure 8-5 Clean-up phase

Row execution sequence

There are three kinds of rows: header, detail, and footer. Tables, lists, and groups have rows. BIRT processes all rows identically. Figure 8-6 illustrates the method execution sequence for a row.

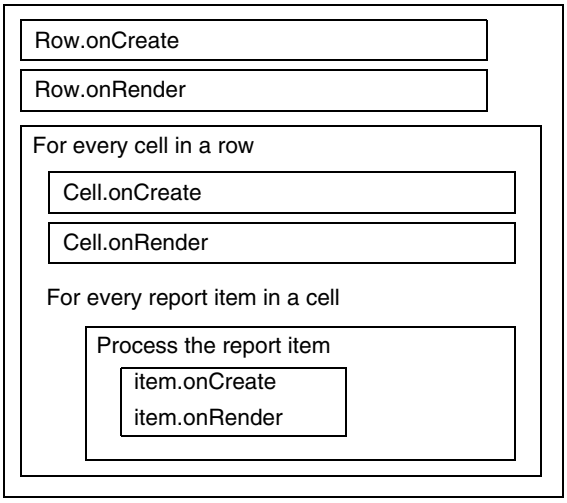

Figure 8-6 Row execution sequence

Table and list method execution sequence

A list is the same as a table, except it only has a single cell in every row. BIRT processes tables and lists identically except that for a list, BIRT does not iterate through multiple cells. BIRT processes tables in three phases, the setup phase, the detail processing phase, and the wrap-up processing phase, as shown in Figure 8-7.

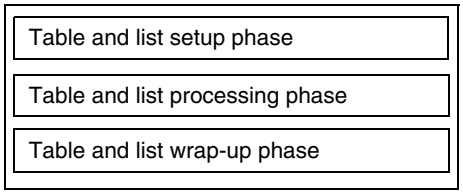

Figure 8-7 Table and list execution sequence

The following sections describe each of the three table and list execution sequence sections.

Table and list setup phase

The pre-table processing phase is the same for all tables, both grouped and ungrouped.

Figure 8-8 illustrates the method execution sequence for the pre-table processing phase.

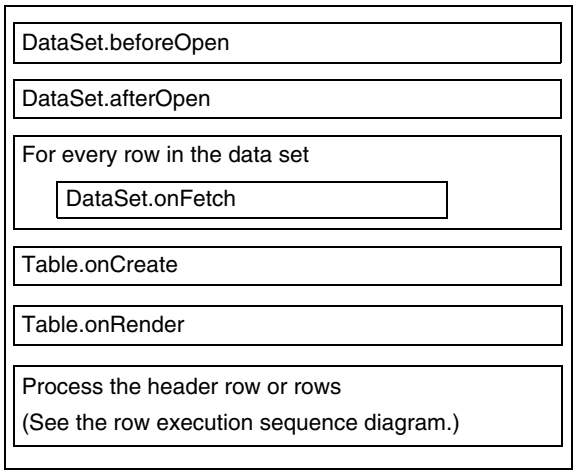

Figure 8-8 Table and list setup execution sequence

Table and list processing phase

The sequence for the table and list processing phase depends on whether the table or list is grouped. The diagram for an ungrouped table or list is shown in "Ungrouped table or list detail execution sequence," later in this chapter. The diagram for a grouped table or list is shown in "Grouped table or list execution sequence," later in this chapter.

Table and list wrap-up phase

The post-table processing phase is the same for all tables, both grouped and ungrouped. Figure 8-9 illustrates the method execution sequence for the post-table processing phase.

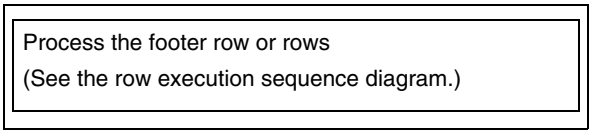

Figure 8-9 Table and list wrap-up execution sequence

Ungrouped table or list detail execution sequence

A table or list with no grouping has a different sequence than one with grouping.

Figure 8-10 illustrates the execution sequence for a table or list without grouping.

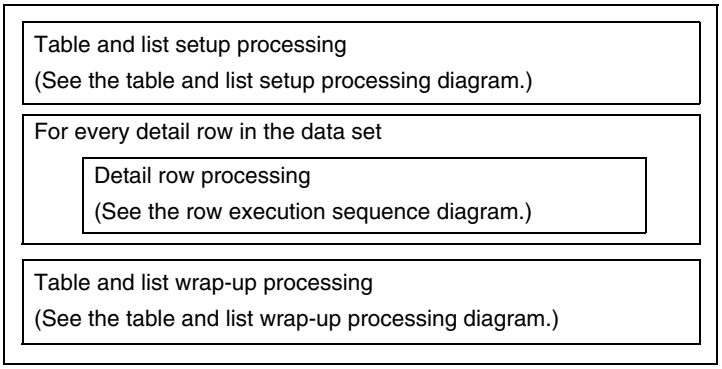

Figure 8-10 Ungrouped table or list detail execution sequence

Grouped table or list execution sequence

One of the differences between the processing sequence for a table or list with grouping and a table or list without grouping is that for a table with grouping, BIRT creates one ListingGroup item per group.

The ListingGroup element has three methods, onCreate, onRow, and onFinish, all of which are called one or more times when processing a grouped table or list. A ListingGroup is very similar to a table because it has one or more header rows, one or more detail rows, and one or more footer rows. BIRT processes grouping rows in the same way that it processes a table row.

Figure 8-11 illustrates the method execution sequence for a table that has groups.

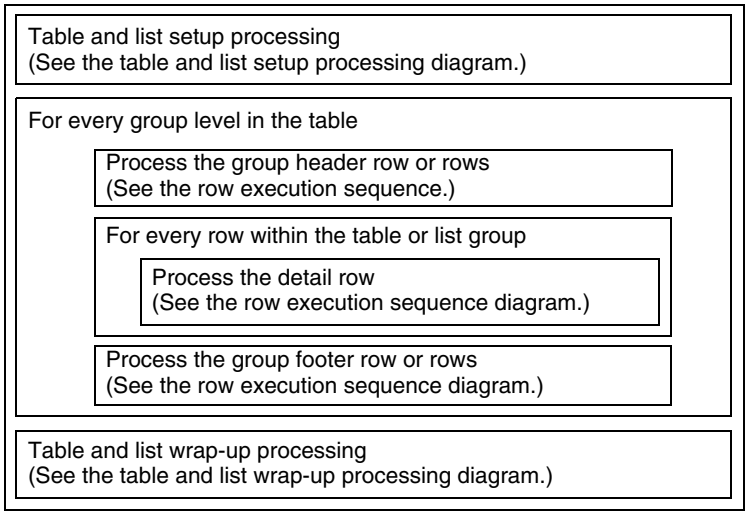

Figure 8-11 Grouped table execution sequence

If you need to verify the execution sequence of event handlers for a specific report, you can add logging code to your event handlers. For information about adding logging code, see the section on determining method execution sequence in the chapter on using JavaScript.

About a report item event handler

You can write event handlers for all report item elements, such as Label and List. Table 8-1 describes the report item event handler methods.

Table 8-1 Report item event handler methods

Method	Description
onPrepare( )	The onPrepare event fires at the beginning of the generation phase, before data binding or expression evaluation occurs. This event is useful for changing the design prior to data binding or expression evaluation.
onCreate( )	The onCreate event fires at the time the element is created in the generation phase, after it is bound to data. This event is useful for operations that depend on the data content of the element.
onRender( )	The onRender event fires in the presentation phase. This event is useful for operations that depend on the type or format of the output document.

About data source and data set event handlers

There are two kinds of data source elements and two kinds of data set elements. The data source elements are DataSource and ScriptedDataSource. The data set elements are DataSet and ScriptedDataSet. ScriptedDataSource and ScriptedDataSet elements are for non-ODA data sources. The events that BIRT fires for the ODA data sources are different from the events that it fires for non-ODA data sources.

ODA data source events

You use the ODA DataSource events, afterClose, afterOpen, beforeClose, and beforeOpen, to perform operations that are not directly related to managing the data source. There is no requirement to implement the ODA data source event handler methods.

Scripted data source events

You use the ScriptedDataSource events, open and close, to perform the actions of opening and closing the data source. You must implement the ScriptedDataSource event handlers.

ODA data set events

As with the ODA data source events, you are not required to provide event handlers for the ODA data set events. The ODA data set events include afterClose, afterOpen, beforeClose, beforeOpen, and onFetch.

Scripted data set events

You must handle the open, close, and fetch events of a ScriptedDataSet element. You use these events to open the data set, close the data set, and to fetch a data set row. In addition to the three ScriptedDataSet events for which you must provide handlers, there is one optional event, the describe event. You use the describe event handler to define dynamically generated columns.

About ReportDesign event handlers

There are several events associated with the ReportDesign element. There are five ReportDesign element events that fire during the report generation process. These five events are not associated with a specific report item, data source, or data set. The ReportDesign events are initialize, beforeFactory, afterFactory, beforeRender, and afterRender.

The initialize event fires before any other event and the initialize event handler is therefore the most logical place to include initialization code.

The beforeFactory event fires before the generation phase. The afterFactory events fire after the generation phase. The beforeRender event fires before the presentation phase.

The afterRender events fire after the presentation phase. There are no specific guidelines for what kind of code to include in these event handlers. They are available for whatever purpose you have for them.

Writing event handlers for charts

While a chart is a report item, it is a much more complex report item than any other. The set of events for a chart is much greater than for any other report item. As with all report items, chart scripting is supported in both Java and JavaScript.

Chart events

All chart Java event handlers receive a chart script context object, IChartScriptContext. The chart script context object has methods to get the chart instance, the locale, and the external context. Some chart event handler methods have arguments of the type Chart, Series, Block, MarkerRange, ISeriesRenderer, GeneratedChartState, DataSet, and IDataSetProcessor.

The chart has a different set of events for which you can write event handlers than the other report items. Table 8-2 lists the chart event handler methods and describes when they are called.

Table 8-2 Chart event handler methods

Method	Called
afterDataSetFilled(Series series, DataSet dataSet, IChartScriptContext icsc)	After populating the series data set
afterDrawAxisLabel(Axis axis, Label label, IChartScriptContext icsc)	After rendering each label on a given axis
afterDrawAxisTitle(Axis axis, Label label, IChartScriptContext icsc)	After rendering the title of an axis
afterDrawBlock(Block block, IChartScriptContext icsc)	After drawing each block
afterDrawDataPoint(DataPointHints dph, Fill fill, IChartScriptContext icsc)	After drawing each data point graphical representation or marker
afterDrawDataPointLabel(DataPointHints dph, Label label, IChartScriptContext icsc)	After rendering the label for each data point

(continues)

Table 8-2 Chart event handler methods *(continued)*

Method	Called
afterDrawFittingCurve(CurveFitting cf, IChartScriptContext icsc)	After rendering curve fitting
afterDrawLegendEntry(Label label, IChartScriptContext icsc)	After drawing each entry in the legend
afterDrawMarkerLine(Axis axis, MarkerLine mLine, IChartScriptContext icsc)	After drawing each marker line in an axis
afterDrawMarkerRange(Axis axis, MarkerRange mRange, IChartScriptContext icsc)	After drawing each marker range in an axis
afterDrawSeries(Series series, ISeriesRenderer isr, IChartScriptContext icsc)	After rendering the series
afterDrawSeriesTitle(Series series, Label label, IChartScriptContext icsc)	After rendering the title of a series
afterGeneration(GeneratedChartState gcs, IChartScriptContext icsc)	After generation of a chart model to GeneratedChartState
afterRendering(GeneratedChartState gcs, IChartScriptContext icsc)	After the chart is rendered
beforeDataSetFilled(Series series, IDataSetProcessor idsp, IChartScriptContext icsc)	Before populating the series data set using the DataSetProcessor
beforeDrawAxisLabel(Axis axis, Label label, IChartScriptContext icsc)	Before rendering each label on a given axis
beforeDrawAxisTitle(Axis axis, Label label, IChartScriptContext icsc)	Before rendering the title of an axis
beforeDrawBlock(Block block, IChartScriptContext icsc)	Before drawing each block
beforeDrawDataPoint(DataPointHints dph, Fill fill, IChartScriptContext icsc)	Before drawing each datapoint graphical representation or marker
beforeDrawDataPointLabel(DataPointHints dph, Label label, IChartScriptContext icsc)	Before rendering the label for each datapoint
beforeDrawFittingCurve(CurveFitting cf, IChartScriptContext icsc)	Before rendering curve fitting
beforeDrawLegendEntry(Label label, IChartScriptContext icsc)	Before drawing each entry in the legend
beforeDrawMarkerLine(Axis axis, MarkerLine mLine, IChartScriptContext icsc)	Before drawing each marker line in an axis
beforeDrawMarkerRange(Axis axis, MarkerRange mRange, IChartScriptContext icsc)	Before drawing each marker range in an axis

Table 8-2 Chart event handler methods *(continued)*

Method	Called
beforeDrawSeries(Series series, ISeriesRenderer isr, IChartScriptContext icsc)	Before rendering the series
beforeDrawSeriesTitle(Series series, Label label, IChartScriptContext icsc)	Before rendering the title of a series
beforeGeneration(Chart cm, IChartScriptContext icsc)	Before generation of a chart model to GeneratedChartState
beforeRendering(GeneratedChartState gcs, IChartScriptContext icsc)	Before the chart is rendered

Chart script context

All chart event handler methods for both Java and JavaScript receive a chart script context argument in the form of a ChartScriptContext object. The chart script context object provides access to the chart instance object, the Locale and ULocale objects, a logging object, and an external context object. You can also use the ChartScriptContext object to set the external context, the chart instance, and the ULocale.

Table 8-3 lists the methods of the chart script context object and their functions.

Table 8-3 Chart script context event handler methods

Method	Function
getChartInstance( )	Returns the chart instance object.
getExternalContext( )	Returns the IExternalContext object that provides access to a scriptable external object. External scriptable objects are defined in the user application.
getLocale( )	Returns the Locale object for the locale currently in use.
getLogger( )	Returns the Logger object, which can be used for logging messages and errors.
getULocale( )	Returns the ULocale object for the locale currently in use.
setChartInstance(Chart)	Sets the chart instance.
setExternalContext(IEXternalContext)	Sets the external context.
setLogger(ILogger)	Sets the logger.
setULocale(ULocale)	Sets the ULocale.

Chart instance object

As explained in the previous section, you get a chart instance object from the chart script context object. The chart instance object contains methods that provide chart modification functionality. Use the chart instance object to get properties, change properties, and test properties.

Chart instance getter methods

The chart instance getter methods allow you to get various properties of a chart.

Table 8-4 lists the chart instance getter methods and the property values they return.

Table 8-4 Chart instance getter methods

Method	Gets
getBlock( )	The value of the Block containment reference
getDescription( )	The value of the Description containment reference
getDimension( )	The value of the Dimension attribute
getExtendedProperties( )	The value of the Extended Properties containment reference list
getGridColumnCount( )	The value of the Grid Column Count attribute
getInteractivity( )	The value of the Interactivity containment reference
getLegend( )	The Legend block
getPlot( )	The Plot block
getSampleData( )	The value of the Sample Data containment reference
getScript( )	The value of the Script attribute
getSeriesForLegend( )	An array of series whose captions or markers are rendered in the Legend
getSeriesThickness( )	The value of the Series Thickness attribute
getStyles( )	The value of the Styles containment reference list
getSubType( )	The value of the Sub Type attribute
getTitle( )	The Title block for the chart
getType( )	The value of the Type attribute
getUnits( )	The value of the Units attribute
getVersion( )	The value of the Version attribute

Chart instance setter methods

The chart instance setter methods allow you to set various properties of a chart. Table 8-5 lists the chart instance setter methods and the values they set.

Table 8-5 Chart instance setter methods

Method	Sets
setBlock(Block value)	The value of the Block containment reference
setDescription(Text value)	The value of the Description containment reference
setDimension(Chart Dimension value)	The value of the Dimension attribute
setGridColumnCount(int value)	The value of the GridColumnCount attribute
setInteractivity(Interactivity value)	The value of the Interactivity containment reference
setSampleData( )	The value of the Sample Data containment reference
setScript( )	The value of the Script attribute
setSeriesThickness( )	The value of the Series Thickness attribute
setSubType( )	The value of the Sub Type attribute
setType( )	The value of the Type attribute
setUnits( )	The value of the Units attribute
setVersion( )	The value of the Version attribute

Writing a Java chart event handler

Writing a Java chart event handler is not different from writing a Java event handler for any other kind of report item. For more information about writing Java event handlers, see Chapter 10, "Using Java to Write an Event Handler."

Writing a JavaScript chart event handler

The process of writing a JavaScript chart event handler differs from the process of writing an event handler for other report items. The primary difference is that the Script tab does not contain a selectable list of chart events. For chart events, you must include every event handler script for the chart in one place.

The Script tab of the BIRT Report Designer for a chart contains a set of function stubs to assist you in writing a chart event handler. The set of stubs is a subset of the chart events, consisting only of the before events, such as beforeDataSetFilled( ). Figure 8-12 shows the Script tab as it appears before any event handlers have been typed.

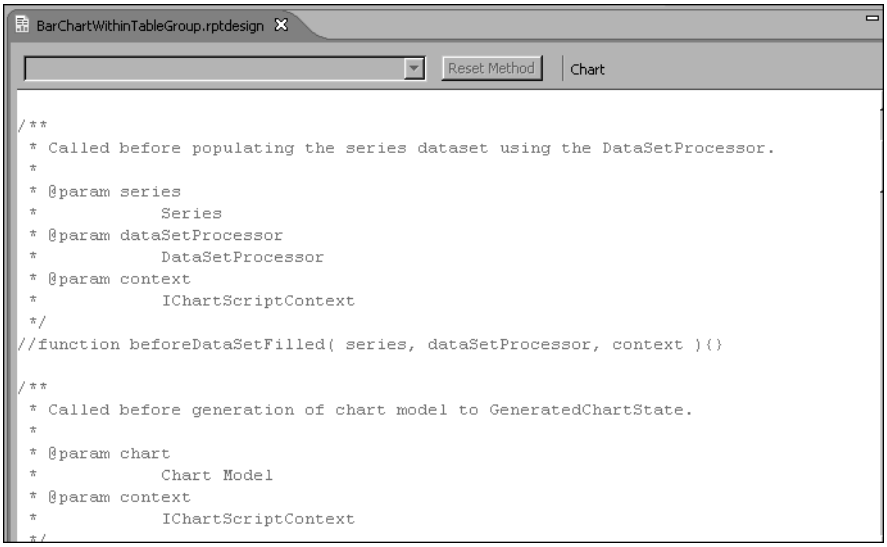

Figure 8-12 Script tab

The before events are the most common events to script. If you need to write an after event handler, such as afterDataSetFilled( ), you can find the signature of the event handler earlier in this chapter or by viewing the interface IChartEventHandler in the Chart Engine API Reference in the BIRT online help.

To write handler code for one of the before chart events, uncomment the appropriate function statement in the Script tab and type the code between the parentheses, as shown in Figure 8-13.

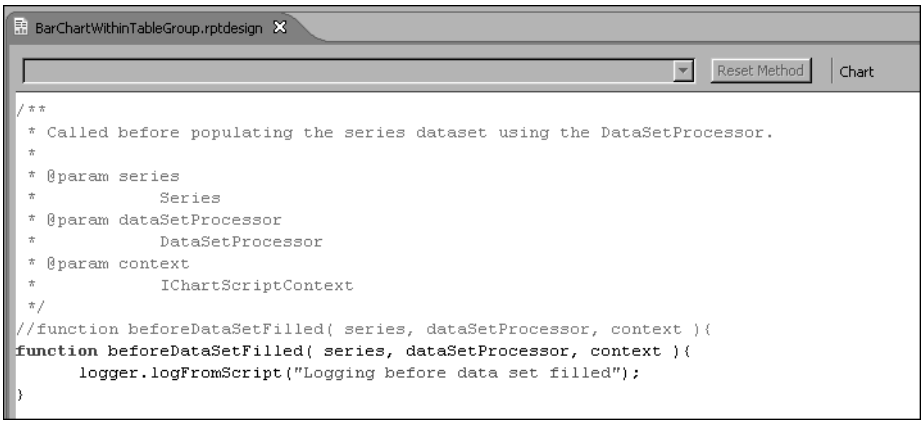

Figure 8-13 Chart event handler code with a standard function declaration

If you want to write an event handler that does not have a stub in the Script tab, you must type the function declaration yourself. When typing a new function declaration, follow the format of the function declarations in the stubs.

Figure 8-14 shows the entry of an afterDrawElement( ) script.

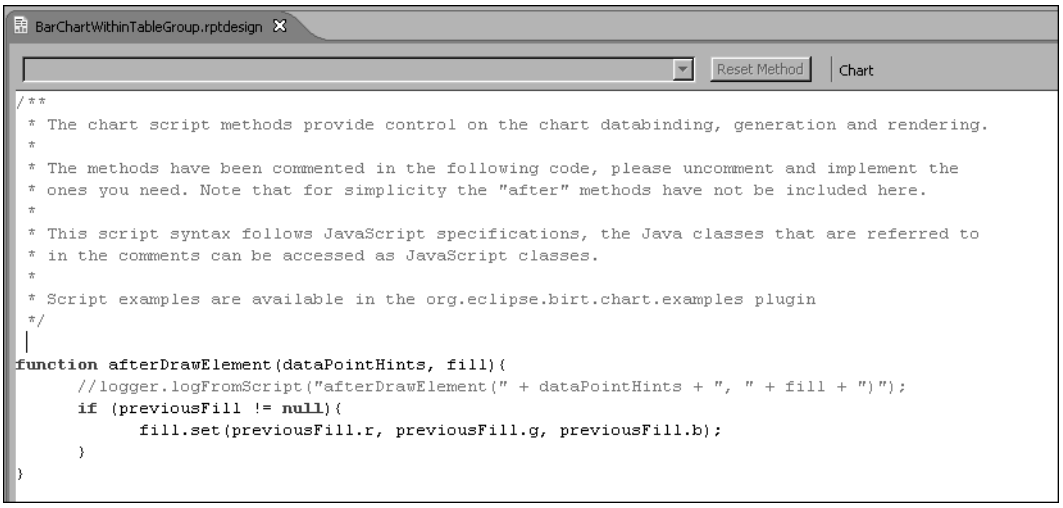

Figure 8-14 Chart event handler code with a custom function declaration

Getting a dynamic image from a Microsoft Access database

Microsoft Access stores an image as an array of image bytes preceded by 78 bytes of header information. BIRT does not use the header information. To get a dynamic image from an Access database, you must copy the image data from the database field into a Java array of type byte. You perform this copy operation in the dynamic image expression.

The following script is an example of how to copy the image data from the Access image field into a byte array that BIRT can use:

```
var picBytes = row["Picture"];
var offset = 78;
var lengthOfImage = picBytes.length - offset;
var imgBytes =
   java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE,
   lengthOfImage);
java.lang.System.arraycopy(picBytes, offset, imgBytes, 0,
   lengthOfImage);
imgBytes;
```