CHAPTER 30

COM Reporting Components

In this chapter

UNDERSTANDING THE REPORT DESIGNER COMPONENT

Crystal Decisions has long viewed the Component Object Model (COM) development platform as one of the key areas it needed to embrace to become successful. Although there were other popular developer platforms in the market, the trend for development projects concerning information delivery was to use Visual Basic. This was because of its good mix of power and simplicity. Now part of the Business Objects product line, Crystal Reports 10 mirrors these attributes and delivers a powerful yet productive reporting solution. This chapter covers Crystal Decisions reporting solutions for the COM platform, specifically, the Crystal Report Designer Component.

Although the chapters covering the Java and .NET components focused primarily on Web-based applications, this chapter concentrates on desktop applications because that is the focus of the Crystal Decisions COM Components. Desktop applications, although still popular today, were what started it all. These are standalone applications that run on a single tier and are installed locally on a user's machine. These applications are most commonly built using Visual Basic, but are also sometimes built using Visual C++ or Delphi.

→ For more information on Java, **see** "Overview of the Crystal Reports Java Reporting Component," **p. 654**

NOTE

All sample code in this chapter uses Visual Basic 6 syntax, but can easily be adapted to other languages that support COM. For sample code in other languages, visit the Business Objects support site at `http://support.businessobjects.com`.

Many development environments support Microsoft's COM technology. *COM (Component Object Model)* is a standard technology used for exposing Software Development Kits (SDKs) in the Windows world. It implies a set of objects with properties and methods. Much of Microsoft's own SDKs are based on COM. It follows that the recommended Crystal Reports SDK for desktop applications would also be based on COM. Its name is the Report Designer Component, and it consists of the following pieces:

- A report designer integrated into the Visual Basic environment
- An object model built around the report engine used for manipulation of the report
- A report viewer control used for displaying reports inside an application

The following sections describe each of these components in more detail.

BUILDING REPORTS WITH THE VISUAL BASIC REPORT DESIGNER

The Visual Basic report designer enables developers to create and edit reports from within the comfort of the Visual Basic environment. Figure 30.1 shows the report designer active inside Visual Basic.

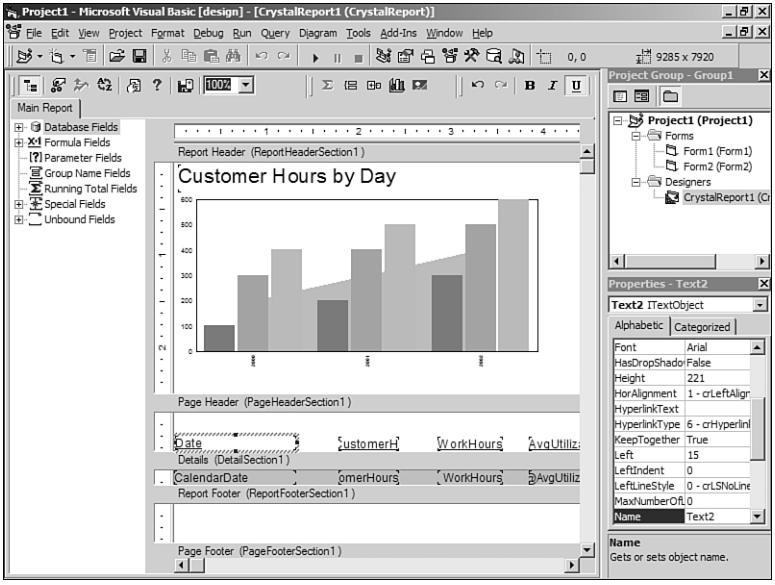


Figure 30.1
Here a report is shown being editing in the Visual Basic report designer.

To add a new report to a project, select Add Crystal Reports 10 from the Project menu inside Visual Basic.

NOTE

> If Add Crystal Reports 10 is not showing on the Project menu, go to the Project, Components menu, and on the Dialog tab, make sure Crystal Reports 10 has a check beside it. If you turn this on, it permanently appears on the Project menu.

From the dialog that opens, select Using the Report Wizard or As a Blank Report to create a new report from scratch. The From an Existing Report option provides you with the capability to import any existing Crystal Report file (.rpt) and use the Visual Basic report designer to make further modifications, a great way to leverage any existing work an organization has put into Crystal Reports. A report that is added to a Visual Basic project is saved as a .dsr file, which is a container for the actual .rpt file along with some other information. At any point, you can click the Save to Crystal Report File button on the designer's toolbar and save the report out to a standard RPT file, so in effect reports can easily go both ways: in and out of Visual Basic. Because the Visual Basic report designer is based primarily on the same code-base as the standalone Crystal Reports designer, the RPT file format is the same. You can also import existing reports from past versions into the Visual Basic report designer.

The Visual Basic report designer supports almost all the features of the Crystal Reports designer and can be used to create everything from simple tabular reports to highly formatted, professional reports. However, even though the capabilities of these two editions are

similar, there are some differences in the way the designer works. This is not meant to be inconsistent, but rather to adapt some of the Crystal Reports tasks to tasks with which Visual Basic developers are familiar. Ideally, the experience of designing a report with the Visual Basic report designer should be like designing a Visual Basic form. The following sections cover these differences.

UNDERSTANDING THE USER INTERFACE CONVENTIONS

Several user interface components work differently in the Visual Basic report designer. One of the first things you might notice is that the section names are shown above each section on a section band as opposed to being on the left side of the window. However, the same options are available when right-clicking on the section band. This tends to be more convenient anyway.

The Field Explorer resides to the left of the report page. Although it cannot be docked, it can be shown or hidden by clicking the Toggle Field View button on the designer toolbar. Other Explorer windows found in the standalone designer such as the Report Explorer and Repository Explorer are not available in the Visual Basic report designer.

NOTE

Reports that contain objects linked to the Crystal Repository are fully supported; however, no new repository objects can be added to the report without using the standalone designer.

The menus that you would normally find in the standalone Crystal Reports designer can be found by right-clicking on an empty spot on the designer surface. The pop-up menu provides the same functionality.

MODIFYING THE REPORT USING THE PROPERTY BROWSER

To change the formatting and settings for report objects in the standalone designer, users are familiar with right-clicking on a report object and selecting Format Field from the pop-up menu. This opens the Format Editor, which gives access to changing the font, color, style, and other formatting options. In the Visual Basic report designer this scenario is still available; however, there is an additional way to apply most of these formatting options: the Property Browser.

The Property Browser is a window that lives inside of the Visual Basic development environment. It should be very familiar to Visual Basic developers as a way to change the appearance and behavior of a selected object on a form or design surface. In the context of the report designer, the Property Browser is another way to change the settings (properties) for report objects. In general, any setting that is available in the Format Editor dialog is available from the property browser when that object is selected. To see which properties are available for a given object, click on it, and check out the Property Browser window shown in Figure 30.2.

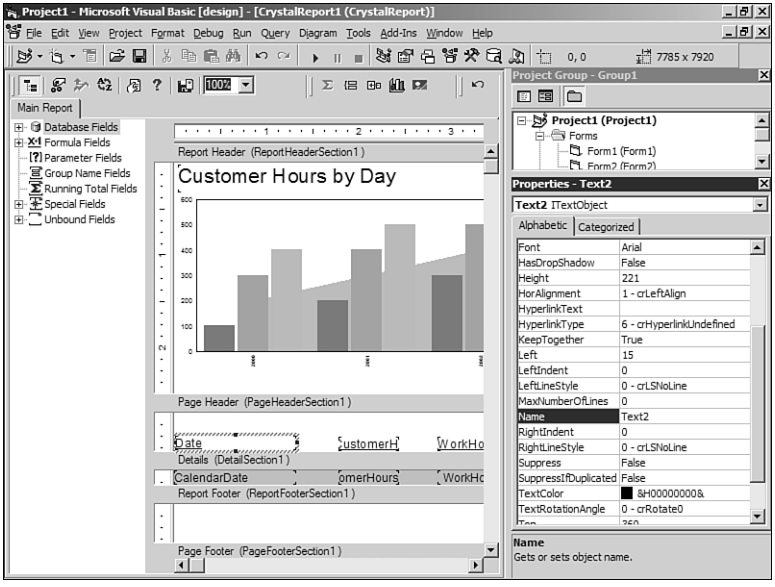


Figure 30.2
Changing a report object's settings via the Property Browser is shown here.

The property names are listed on the left and the current values are listed on the right. To choose a value, simply click on the current value and either type or select from the drop-down list.

One property to pay attention to is the `Name` property. This becomes relevant in the next section when you learn how to use the Report Engine Object Model to manipulate the report on the fly at runtime. This is the way to reference that object in code. Also of note is that the properties shown in the Property Browser map to the same properties that are available programmatically via the object model. If you see a property there, this means it is also available to be changed dynamically at runtime.

UNBOUND FIELDS

The Field Explorer in the Visual Basic report designer has an extra type field not found in the standalone report designer. These are called unbound fields. There is one type of unbound field for each data type. These fields are used to build dynamic reports. Because they do not have a predefined database field mapped to them, they provide a way to change the locations of fields on the report by using some application logic. The reason they each have their own data type is so that type-specific formatting can be applied such as the year format for a date object, or the thousands separator for a numeric object. Unbound fields are revisited later in this chapter.

NOTE

When you create an unbound field, it also shows up as a formula in the Formula Fields list. This is because a formula is used behind the scenes of an unbound field. The best practice is not to edit this as a formula field.

PROGRAMMING WITH THE REPORT ENGINE OBJECT MODEL

The object model is the main entry point to the Crystal Reports engine for desktop applications. As mentioned earlier, it is based on COM and can be used from any COM-compliant development environment. Although the main library's filename is craxdrt.dll, the more important thing to know is that it shows up in the Project References dialog as Crystal Reports ActiveX Designer Runtime Library 10.0, as shown in Figure 30.3. After a reference is added to this library, a new set of objects will be available to you. These objects are contained in a library called `CRAXDRT`. To avoid name collisions, it's probably a good idea to fully qualify all object declarations with the `CRAXDRT` name, for example, `Dim Param As CRAXDRT.ParameterField`.

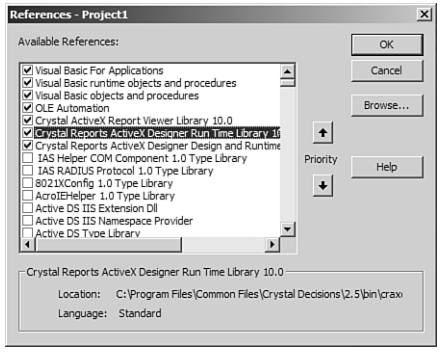


Figure 30.3 Reference the Report Designer Component's Object Model in the References dialog.

NOTE

When adding a Crystal Report to your project, a reference is automatically added to this library for you. You can use the `CRAXDRT` library right away.

In addition to many other features, the object model provides the capability to open, create, modify, save, print, and export reports. This section covers some of the more common scenarios a developer might encounter.

The main entry point to the object model is the Report object. This object is the programmatic representation of the report template and provides access to all the functions of the SDK. There are three ways to obtain a Report object:

- Load an existing RPT file from disk
- Create a new report from scratch
- Load an existing strongly typed report that is part of the Visual Basic project

The first two methods involve the `Application` object. Its two key methods are `NewReport` and `OpenReport`. As the name implies, the `NewReport` method is used to create a blank report

and the `OpenReport` method is used to open an existing report. Creating a new report is useful if the application needs to make a lot of dynamic changes to the report's layout on the fly. This way all report objects can be added dynamically.

NOTE

Although many purist developers are tempted to not use any predefined report templates (RPT files) and create all reports on the fly, this tends to be overkill for most projects. There is a lot of work in having to programmatically make every single addition to a report. It's usually a better plan to have some RPT files as part of the application and then make some small modifications at runtime.

The other option is to use the `OpenReport` method, which takes a filename to an RPT file as a parameter. This opens an existing report. When using this method, the RPT files must be distributed along with the application. The advantage of having these externals files is that you can update the reports without updating the application.

The last method is to use a strongly typed report. A strongly typed report is one that is added to the Visual Basic project and turned into a DSR file. Whichever name you give that report, a corresponding programmatic object exists with the same name. For example, if you save a report as BudgetReport.DSR, you can create that report programmatically with the following code:

```
Dim Report As New BudgetReport
```

There are several advantages to using strongly typed reports. First, all report files are bound into the project's resulting executable so no external files are available for users to modify and mess up. Second, not only is the name of the report strongly typed (BudgetReport in the previous example), but also the section and report objects. For example, if you have a text object acting as a column header and you want to modify this at runtime, it's very easy to access the object by name like this:

```
Report.ColumnHeader1.SetText "Some text"
```

The long-handed way of doing this would look something like this:

```
Dim field as CRAXDRT.FieldObject
Set field = Report.Sections("PH").ReportObjects(3)
Field.SetText "Some text"
```

Not only is this last method longer, you'd also have to refer to the report object by index instead of name, which can become problematic. The following sections discuss some of the common tasks that are performed after a Report object is obtained.

EXPORTING REPORTS TO OTHER FILE FORMATS

A very common requirement for application developers is to be able to export a report through their application. Not only do developers want a variety of formats, they want the export to happen in a variety of ways, for example, having the user select where to save the report, saving to a temp file and then opening it, e-mailing it to somebody else, and so on.

By being creative with exporting, you can create some very powerful applications. The Report Designer Component object model provides a very flexible API to meet these broad needs. This section covers the basics of exporting.

There are two components to exports: the format and the destination. The developer specifies both of these through the `ExportOptions` property of the `Report` object.

Setting the format options involves two steps. The first step is to choose which format you want to export to. Sometimes an application provides the user a list of export formats and lets him choose, other times the export type will be hardcoded. In any case, simply setting the `FormatType` property of the `ExportOptions` object specifies this. This property accepts a number. For example, to export to PDF, pass in 31. Remembering which number represents which format is tough so there are some enumerations with descriptive names that make this easier.

NOTE

For a full list of enumerations, consult the Crystal Reports Developers Help file and look at the `CRExportFormatType` enumeration in the Visual Basic object browser.

To help you get started, here are some of the more popular export format enumeration values:

- PDF: `crEFTPortableDocFormat`
- Word: `crEFTWordForWindows`
- Excel: `crEFTExcel97`
- HTML: `crEFTHTML40`
- XML: `crEFTXML`

Generally, all you need to do to set the format options is set the `FormatType` property. However, many of the format types have some additional options. For example, when exporting to Excel there is an option to indicate whether you want the grid lines shown. To handle these extra settings, there are some other properties off the `ExportOptions` object whose names begin with the format type. In the Excel grid lines example, the property is called `ExcelShowGridLines`. For PDF, there are `PDFFirstPageNumber` and `PDFLastPageNumber` properties that indicate which pages of the report you want exported to PDF. You can determine what options are available by checking out the Crystal Reports Developer Help file and looking at the `ExportOptions` object.

After the format is set up, you need to tell Crystal Reports where you want this report to be exported. This is called the *export destination*. The most common destination is simply a file on disk but there are destinations such as e-mail or Microsoft Exchange folders where reports can be automatically sent. The export destination is set via the `DestinationType` property of the `ExportOptions` object. Some example values are listed here:

- File: `crEDTDiskFile`
- E-mail: `crEDTMailMAPI`
- Exchange: `crEDTMicrosoftExchange`

Check out the `CRExportDestinationType` enumeration to see the other available options. Like the format, the destination has a set of additional options. The most obvious one is when setting the destination to a file (`crEDTDiskFile`), you would need to specify where you want this file and what its name should be. This is accomplished by setting the `DiskFileName` property. Other properties on the `ExportOptions` object are available such as the `MailToList` property, which is used to indicate who the report should be mailed to if the e-mail option is selected as the destination.

The final step in exporting is to call the `Report` object's `Export` method. It takes a single parameter: `promptUser`. If this is set to true, any options previously set on the `ExportOptions` object are ignored and a dialog appears asking the user to select the format and destination. This can be useful if you want the user to have the capability to use any export format and any destination. If you would like a more controlled environment, you can set `promptUser` to false. When this is done the previously selected values from the `ExportOptions` object are respected and the export is done without any user interaction besides a progress dialog popping up while the export is happening. This progress dialog can also be suppressed by setting the `Report` object's `DisplayProgressDialog` property to false. Listing 30.1 provides an example of a report being exported to a PDF file without any user interaction.

LISTING 30.1 EXPORTING TO PDF

```
Dim Report As New CrystalReport1

' Set export format
Report.ExportOptions.FormatType = crEFTPortableDocFormat

' Set any applicable options for that format
' In this case, set to only export pages 1-2
Report.ExportOptions.PDFFirstPageNumber = 1
Report.ExportOptions.PDFLastPageNumber = 2

' Set export destination
Report.ExportOptions.DestinationType = crEDTDiskFile

' Set any applicable options for the destination
' In this case, the filename to be exported to
Report.ExportOptions.DiskFileName = "C:\MyReport.pdf"

' Turn all user interface dialogs off and perform the export
Report.DisplayProgressDialog = False
Report.Export False
```

PRINTING REPORTS TO A PRINTER DEVICE

Although it's helpful to view reports onscreen and save some paper, many times reports still need to be printed. To accomplish this, there is a collection of methods for printing reports available from the `Report` object. The simplest way to print a report is to call the `PrintOut` method passing in true for the `promptUser` parameter as shown here:

```
Report.PrintOut True
```


This opens the standard Print dialog that enables the user to select the page range and then click OK to confirm the print. The limitation to this is that the pop-up dialog does not enable the user to change the destination printer. Because this is a common scenario, this method isn't used very often. Instead, the `PrinterSetup` method is called. This method pops up a standard printer selection dialog that enables the user to change the paper orientation or printer.

Keep in mind that calling the `PrinterSetup` method does not actually initiate the print; it only collects the settings to be used for the print later on. Luckily it does indicate via a return value whether the user clicked the OK or Cancel button. Listing 30.2 shows an example of how to use the `PrinterSetup` method to set printer options.

LISTING 30.2 PRINTING A REPORT INTERACTIVELY

```
' Call PrinterSetup to set printer, paper orientation, and so on
If Report.PrinterSetupEx(Me.hWnd) = 0 Then
   ' If the return value is 0, the user did not click Cancel
   ' so go ahead with the print
   Report.PrintOut False
End If
```

To print a report without any user interaction, call the `PrintOut` method passing in false for the `promptUser` parameter. Options such as pages and collation can be set with the additional argument to the `PrintOut` method. To change the printer, call the `SelectPrinter` method. This accepts the printer driver, name, and port as parameters and performs the printer change without any user interaction. Listing 30.3 illustrates a silent print.

LISTING 30.3 PRINTING A REPORT SILENTLY

```
' Call PrinterSetup to set printer, paper orientation, and so on

' Set paper orientation
Report.PaperOrientation = crLandscape

' Set printer to print to
'   pDriver -- for example: winspool
'   pName -- for example: \\PRINTSERVER\PRINTER4
'   pPort -- for example: Ne00:
Report.SelectPrinter pDriver, pName, pPort

' Initiate the print
Report.PrintOut False
```

SETTING REPORT PARAMETERS

Often reports delivered through an application need to be dynamically generated based on a parameter value. If a report with parameters is viewed, exported, or printed, a Crystal parameter prompting dialog pops up and asks the user to enter the parameter values before the report is processed. This parameter prompting dialog requires no code. The use of the object model comes into play when a developer wants to set parameters without user interaction. This is done via the `ParameterFieldDefinitions` collection accessed via the `Report` object's `ParameterFields` property. If all parameter values are provided before the report is processed, the parameter dialog is suppressed.

Parameters can be referenced by name or by number. To reference by name, call the `ParameterFields` object's `GetItemByName` method passing in the name of the parameter you want to access. This returns a `ParameterField` object. Alternatively, use the indexer on the `ParameterFields` object; for example, `ParameterFields(1)`. When referencing by index, the parameters will be stored in the same order they appear in the Field Explorer window in the report designer. After a `ParameterField` object is obtained, simply call the `AddCurrentValue` method to set the parameter's value as shown in Listing 30.4.

LISTING 30.4 SETTING PARAMETERS

```
Dim Application As New CRAXDRT.Application
Dim Report As CRAXDRT.Report

' Open the report from a file
Set Report = Application.OpenReport("C:\MyReport.rpt")

Dim p1 as ParameterField
Set p1 = Report.ParameterFields.GetItemByName("Geography")
p1.AddCurrentValue("Europe")


Dim p2 as ParameterField
Set p2 = Report.ParameterFields(2)
p2.AddCurrentValue(1234)
```

If the parameter accepts multiple values, simply call the `AddCurrentValue` method multiple times. For range parameters where there is a start and an end value, use the `AddRangeValue` method.

Sometimes a developer wants to prompt the user to enter some or all of the parameters but they want to control the user interface. Much information about the parameter can be obtained by reading its properties:

- `ParameterFieldName`: Name of the parameter
- `ValueType`: The data type of the parameter (string, number, and so on)
- `Prompt`: The text to use to prompt for this parameter

Also, by using the `NumberOfDefaultValues` property and `GetNthDefaultValue` method, a developer can construct her own pick-list of default parameter values that is stored in the report.

NOTE

For more information on the other properties and methods available on the `ParameterField` object, consult the Crystal Reports Developer Help file and look for the `ParameterFieldDefinition` object.

SETTING DATA SOURCE CREDENTIALS

Although the sample reports that come with Crystal Reports 10 use an unsecured Microsoft Access database as their data source, most real-world reports are based on a data source that require credentials (username, password) to be passed. Also, it's very common to want to change data source information such as the server name or database instance name via code. This section covers these scenarios.

Unlike parameters, there is no default-prompting dialog for data source credentials. They must be passed via code. The server name, location, database name, and username are all stored in the report. However, the password is never saved. A report will fail to run if a password is not provided.

Most reports only have a single data source but because it is possible for reports to have multiple data sources that in turn would require multiple sets of credentials, setting credentials isn't something that's done on a global level. Credentials are set for each table in the report. Tables are represented by an object called a `DatabaseTable` inside the object model. The following code snippet illustrates the hierarchy required to get at the `DatabaseTable` object.

```
Report
   Database
      DatabaseTables
         DatabaseTable
```

Tables are accessed by their index, not their name. The indexes in the object model are all 1-based and are in the order you see them in the Field Explorer in the report designer. To access the first table in the report, you could do this:

```
Dim tbl as DatabaseTable
Set tbl = Report.Database.Tables(1)
```

After the correct `DatabaseTable` object is obtained, use the `ConnectionInfo` property bag to fill in valid credentials. If you do only have one data source in the report, but multiple tables from that data source, you need not set credentials for each one. The information is propagated across all tables. Listing 30.5 illustrates setting the server name, database name, username, and password for a report based off an OLEDB data source.

LISTING 30.5 SETTING DATA SOURCE CREDENTIALS

```
' Provide database logon credentials (in this case
' for an OLEDB connection to a SQL Server database)
Dim tbl as CRAXDRT.DatabaseTable
Set tbl = Report.Database.Tables(1)
tbl.ConnectionInfo("Data Source") = "MyServer"
tbl.ConnectionInfo("Initial Catalog") = "MyDB"
tbl.ConnectionInfo("User ID") = "User1"
tbl.ConnectionInfo("Password") = "abc"
```


Each type of data source has its own set of properties. OLEDB has a Data Source, which is the server name whereas the Microsoft Access driver has a Database Name, which is a file-name to the MDB file. The `ConnectionInfo` property bag is introspective so you can loop through and determine what properties are available.

MAPPING UNBOUND FIELDS AT RUNTIME

Earlier in this chapter you saw that a new type of field called an unbound field can be added to the report with the Visual Basic report designer. Using the object model, these unbound fields can be mapped to database fields in the report at runtime. This is done two different ways: manually or automatically.

The manual method is to use the `SetUnboundFieldSource` method of the `FieldObject`. This method takes a single parameter, which is the name of the database field to be mapped in the Crystal field syntax, such as {`Table.Field`}. If a strongly typed report is being used, that is, a report added to the Visual Basic project, the `UnboundField` objects can be referenced as properties of the `Report` object. For example, an unbound field object given the default name of `UnboundString1` can be referenced like this:

```
Report.UnboundString1.SetUnboundFieldSource "{Customer.Customer Name}"
```

If a report is loaded at runtime, there are no strongly typed properties so the `FieldObject` needs to be found under the Section and ReportObjects hierarchy. The following example gets a reference to the first unbound field in the details section:

```
Dim fld As FieldObject
Set fld = Report.Sections("D").ReportObjects(1)
fld.SetUnboundFieldSource "{Customer.Customer ID}"
```

The automatic method is to simply call the `Report` object's `AutoSetUnboundFieldSource` method. This assumes that any unbound fields to be mapped are named to match a database field. Initially this might seem strange because the whole point of an unbound field is that the developer doesn't know which database field it will be mapped to at design time. However, this automatic method is valuable when the database table doesn't exist at design time, and instead is added at runtime based on some dynamic data.

USING THE CODE-BEHIND EVENTS

One of the reasons that the report is saved as a DSR file instead of just an RPT file is that the DSR file contains some code that is attached to the report file. This code, often called

code-behind, is event-handing code for several events that the report engine fires. The following list describes events that are fired and their corresponding uses:

- `Initialize (Report)`: Fired when the report object is first created. This event can be useful for performing initialization-related tasks.
- `BeforeFormatPage/AfterFormatPage (Report)`: Fired before and after a page is processed; can be useful for indicating progress.
- `NoData (Report)`: Fired when a report is processed but no records were returned from the data source. Sometimes a report with no records is meaningless and thus should be skipped or the user should be warned; this event is a great way to handle that.
- `FieldMapping (Report)`: Fired when the database is verified and there has been a schema change; this event enables you to remap fields without user interaction.
- `Format (Section)`: Fired for the rendering of each section. This is useful for handling the detail section's event and performing conditional logic.

DELIVERING REPORTS USING THE REPORT VIEWER

In the previous section, only printing and exporting were mentioned as options for delivering reports. You might have been wondering how to view reports onscreen. This section will cover using the report viewer to view reports. This report viewer control is usually referred to as the ActiveX viewer, or the Crystal Reports Viewer Control. It is an ActiveX control, which means that in addition to being able to be dropped on to any Visual Basic form—like the other components of the Report Designer Component—it can be used in any COM-compliant development environment. Its filename is CRViewer.dll. Figure 30.4 depicts the ActiveX viewer displaying a report from a Visual Basic application.

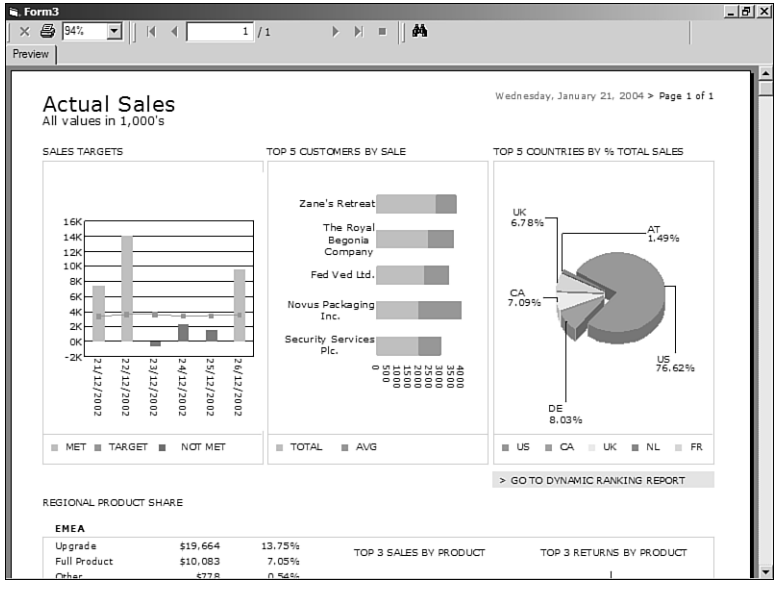


Figure 30.4
A Crystal Report is shown here being displayed in the ActiveX viewer.

The ActiveX viewer works in conjunction with the object model and report engine to render the report to the screen. The object model talks to the report engine to process the report, and then the ActiveX viewer asks the object model for the data for an individual page. After this data is received by the viewer, it displays the report page onscreen. The following code snippet illustrates how to view a report with the report viewer control:

```
Dim Application As New CRAXDRT.Application
Dim Report As CRAXDRT.Report

Set Report = Application.OpenReport("C:\MyReport.rpt")
CRViewer.ReportSource = Report
CRViewer.ViewReport
```


The ActiveX control has many properties and methods that enable you to customize its look and feel. To turn off the toolbar at the top of the viewer control, simply set the `DisplayToolbar` property to false. To turn off the group tree, set the `DisplayGroupTree` property to false. This can result in a very minimalist viewer. In addition, the control has a full event model that notifies you when certain actions are performed, such as a drill-down or page navigation. For more information on the ActiveX viewer control, consult the Crystal Reports 10 developer help file.

USING THE OBJECT MODEL TO BUILD BATCH REPORTING APPLICATIONS

So far this chapter has focused on on-demand reporting, meaning that reports are processed as they are requested and they generally go away when the viewing or printing is completed. One of the biggest uses of the Report Designer Component today is for batch reporting; that is, running a large number of reports at once. This section covers some features and best practices relevant to batch reporting.

WORKING WITH REPORTS WITH SAVED DATA

When using the standalone report designer, you might have noticed an option on the File menu called Save Data with Report. This enables a report to be saved with the last returned dataset so that it can be viewed again without connecting to the database. Reports with saved data are in effect an offline report.

Applications using the Report Designer Component can both create and view reports with saved data. This enables you to run a batch of reports and then be able to view them at any point later. This can be useful for reports based on queries that take a long time to run, or also for achieving an archiving process for reports.

Creating a report with saved data is very simple. You just export to the Crystal Reports format by using the `crEFTCrystalReport` identifier. All exported reports have saved data. You can control where this report is saved and archive it for later.

Viewing a report with saved data doesn't actually require any code at all. The logic of the report engine is: If the report has saved data, use it and only hit the database again if the

user clicks the Refresh button or the developer forces a refresh by calling the `DiscardSavedData` method off the `Report` object. You can always tell which copy of the data is being used from examining the `DataDate` property of the `Report` object.

Hopefully you can imagine how applying this principle to batch reporting would be powerful. A set of reports could be run overnight, producing another set of reports with saved data that can be viewed offline.

LOOPING THROUGH REPORTS

Another scenario that is relevant to batch reporting is looping through a set of reports. A common example is running either one report many times with different parameters (such as a bank statement) or running a large collection of reports all at once (such as financial statements).

These scenarios can be accomplished by using external report files and writing a loop that opens a report, prints or exports it, and then closes it. The best way to close a report is to set the Report object to `Nothing`:

```
Set Report = Nothing
```

This releases the COM object and releases the report job from memory.

Also, the `CRAXDRT` library is thread safe, which means that multiple threads can be calling into it at the same time. If a large number of reports need to be processing in a very small amount of time, you can spawn as many as five simultaneous threads that are all running reports at the same time.

TROUBLESHOOTING

ADD CRYSTAL REPORTS 10

"Add Crystal Reports 10" is not showing on my Project menu.

If you don't see "Add Crystal Reports 10" on the Project menu, go to the Project, Components menu, and on the Dialog tab make sure Crystal Reports 10 has a check beside it. If you turn this on, it permanently appears on the Project menu.

PART VII

Customized Report Distribution—Using Crystal Enterprise Embedded Edition

CHAPTER 31

INTRODUCTION TO CRYSTAL ENTERPRISE EMBEDDED EDITION

In this chapter

INTRODUCTION TO CRYSTAL ENTERPRISE EMBEDDED EDITION

The Report Application Server (RAS) is a set of components that enable developers to take advantage of the report-design capabilities of the Crystal Reports engine. These components enable developers to build applications that include report design, modification, and viewing functionality and can be accessed through a Web browser. RAS is the replacement and evolution of previous single threaded object models and Crystal Products such as the Report Designer Component (RDC) and the Crystal Reports Print Engine (CRPE).

In version 10 of the Crystal Suite of Business Objects products, the standalone RAS is called the Crystal Enterprise Embedded edition. RAS is built using client/server technology. The server components consist of the application logic that interfaces with the reporting print engine. The client components consist of the RAS Software Developer's Kit (SDK) that communicates with the RAS via TCP/IP and the Viewers SDK that exposes a number of report viewers each suiting particular application needs. The RAS is a multithreaded server with both .NET/COM and Java object models.

The RAS and Viewers SDKs are discussed in greater detail in Chapter 32, "Crystal Enterprise—Viewing Reports," and Chapter 33, "Crystal Enterprise Embedded—Report Modification and Creation." This chapter introduces the sample applications provided around Crystal Enterprise Embedded and provides some configuration information.

UNDERSTANDING CRYSTAL ENTERPRISE EMBEDDED EDITION

Crystal Enterprise Embedded edition (the RAS by itself) is used in a standalone mode to deliver Crystal Report's creation, modification, and viewing functionality over the Web. In its simplest description, it can be thought of as an open Report Engine with a published object model and viewer controls. Crystal Enterprise Professional and Premium editions were introduced in Part V, "Web Report Distribution—Using Crystal Enterprise." Each of these advanced editions of Crystal Enterprise can also leverage the powerful report exploration (creation and modification) functionality of the RAS and object model. In these advanced editions, the RAS is effectively plugged into the Crystal Enterprise infrastructure or backbone and managed as any of its other services. Figure 31.1 displays the basic RAS standalone architecture.

In this standalone case, the installation is limited to a single RAS for the custom applications written to interact with. The RAS accesses reports on the server based on a central location specified in the RAS Configuration tool (see the next section for more detail). You can however have multiple installations of standalone RAS that share a central network location where the reports reside. Keep in mind that it is generally not a good idea to have the report located somewhere other than the RAS server—applications opening reports on this server

require the server components to load the involved report and to create a local copy of it. The network traffic associated with pulling the .rpt file from a location on a different server results in application performance degradation.

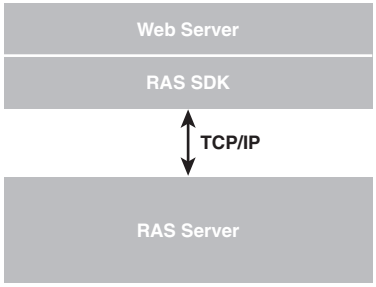


Figure 31.1
The RAS architecture provides programmatic access to report creation and modification.

USING THE CRYSTAL CONFIGURATION MANAGER

The Crystal Configuration Manager (CCM) provides a point of access for setting the different options around the Crystal Enterprise Embedded (or RAS) installation. It is accessed through the Microsoft Start, Programs, Crystal Enterprise menu path and is highlighted in Figure 31.2.

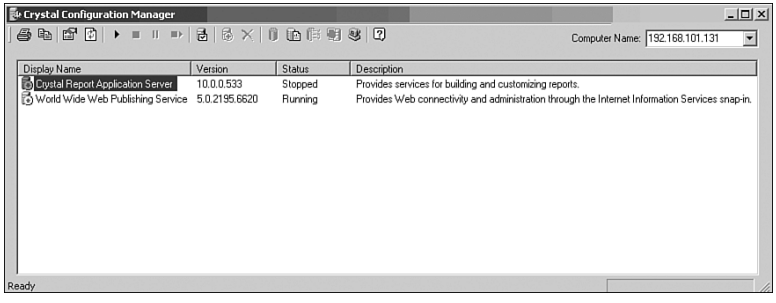


Figure 31.2
The CCM for the RAS provides access to key settings.

The default report location along with other RAS server settings can be accessed by stopping the RAS service in the CCM and then selecting Properties through the Properties button or the right-click menu on the service.

SETTING DATABASE PARAMETERS

After having accessed the Crystal RAS Properties dialog box, click on the Parameters tab and ensure the Option Type drop-down box has the Database option selected as shown in Figure 31.3. In this dialog, you can set the number of records that are brought back in reports by default or the number of records accessed in one batch. You can also set how many records are accessed and brought back when you expose the Browse Field functionality in your applicaton(s).

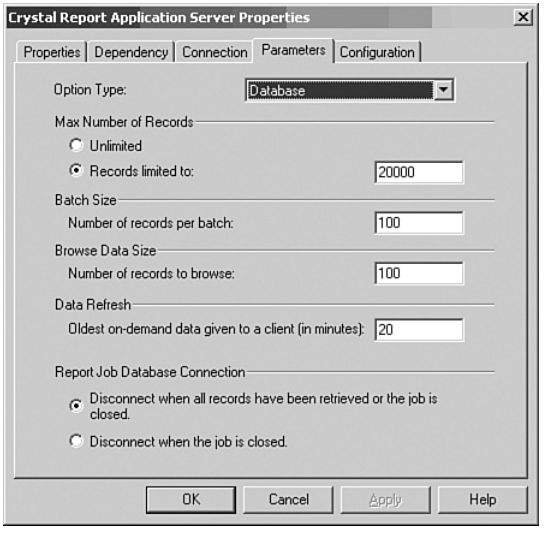


Figure 31.3
Setting the RAS Database options.

SETTING SERVER PARAMETERS

On the same Parameters dialog, after choosing Server for the option type you are shown the dialog displayed in Figure 31.4. Here you can set the location of your reports, the number of simultaneous jobs, and a number of minutes before an idle job is closed. Keep in mind as you change these settings you need to restart the RAS service for them to take effect.

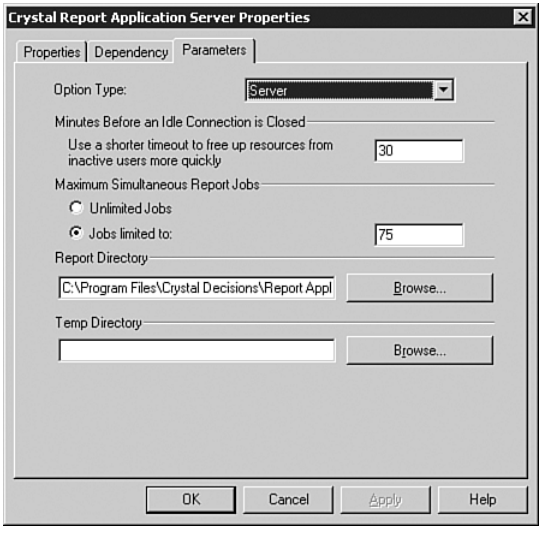


Figure 31.4
The Parameters tab of the RAS properties dialog enables you to set key RAS options such as report location, simultaneous job maximums, and user timeout.

NOTE

The RAS also exposes caching capabilities that enable multiple users to view the same copy of a cached report. This ultimately increases the number jobs the RAS can handle at any given point. Keep in mind however that if your reports contain subreports these are not cached.

CRYSTAL ENTERPRISE EMBEDDED EDITION SAMPLES

There are a number of sample applications that ship as part of Crystal Enterprise Embedded (RAS standalone). This section describes these sample applications. It is important to note that the purpose of these examples is to demonstrate the basic capabilities of RAS in action and to provide sample starting points for further application development. In some cases you will find these immediately useful for allowing your users to perform simple tasks like viewing reports, setting report data sources at runtime, changing the selection formula at runtime, or passing parameters to reports. In the majority of cases, it is expected you will be able to leverage the concepts and sample code as starting points into developing your own rich and more full-featured applications.

REPORT PREVIEW SAMPLE

The Report Preview Sample is an application that demonstrates report viewing using Crystal Enterprise Embedded's RAS (see Figure 31.5). The frame on the left represents the directory structure as it appears starting at the root of the default RAS reports directory. By default, this directory is c:\program files\crystal decisions\report application server10\reports. After a report is selected to view, it appears inside the bottom-right frame. The top-right frame lists report viewers that are available with RAS. You can toggle between the viewers available by selecting the desired option in this frame.

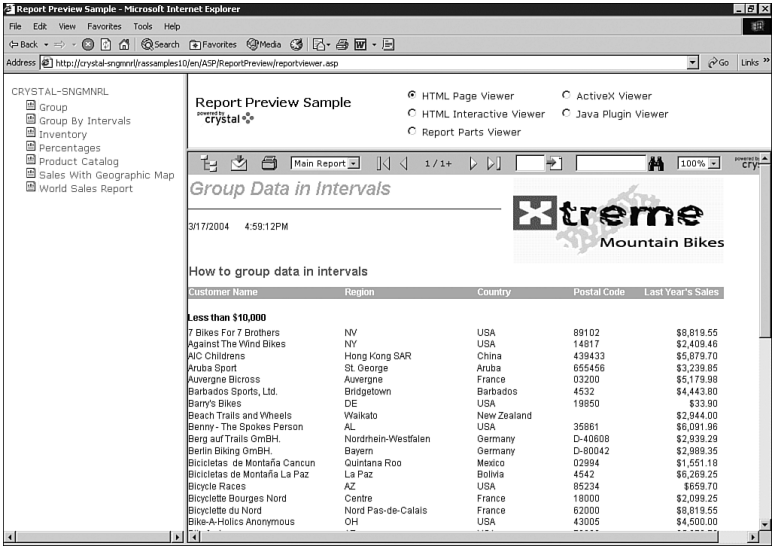


Figure 31.5
The Report Viewer Sample provides a good starting point for a report viewing application and also provides sample code.

Additional information on the report viewers for RAS and their programmatic hooks and controls is provided in the next chapter.

THE SIMPLE DISCRETE PARAMETER SAMPLE

The Simple Discrete Parameter example shown in Figure 31.6 provides a demonstration of a report running with parameters. As previously discussed in this book, there are often cases where reports are required to run with parameters against the database. Some of these parameters might be passed in without the report consumer even being aware of them (for example, global environment variables to set a user's preferred language for report viewing). Other types of parameters require user input where the user is presented with a list of values to choose from and a report is run using those selected values and limits the resultset displayed in the report.

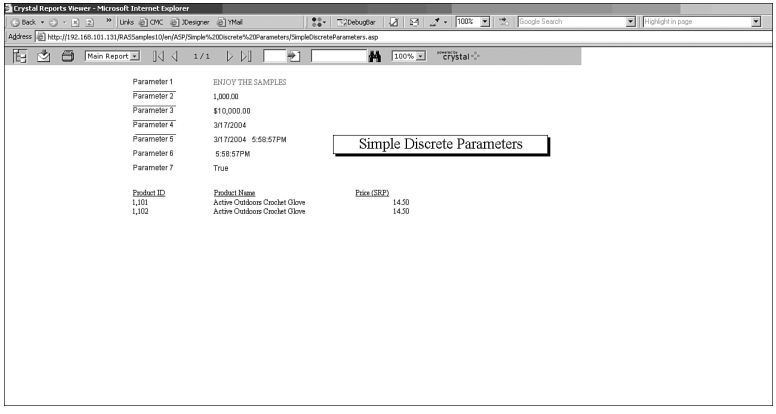


Figure 31.6
The Simple Discrete Parameters example provides a good starting point for a report viewing application using parameters.

THE DATABASE LOGON SAMPLE

The Database Logon example illustrates the capability for the end user to supply his own logon credentials to the report when it is retrieving data from the database. This scenario is commonly found in implementations where users should only be able to see data that pertains to them and user-level database security has been set up. An example of this might include a sales executive for the eastern region and his counterpart for the western region viewing the same pipeline report—each executive only needs to view information for his specific geographies. When either of them views the report, they are prompted to supply logon information (see Figure 31.7) to continue and this information is passed to the database and the appropriate user-level security is applied there.

THE DATA SOURCE LOCATION SAMPLE

The Set Data Source Location sample demonstrates the task of setting the location of the database to be used for the involved report at runtime. This example might be useful in scenarios where you need to run the same report against different environments (such as Development, Testing and Quality Assurance [QA]). Another practical example would be in

the scenario where different versions of identical databases are kept across an organization. You might want to enable the application users to decide what data source to dynamically run the report against (for example, different regional databases). In this case, a custom Web page could be developed to let the user make the desired choice and then use the RAS functionality to connect the report to the selected database.

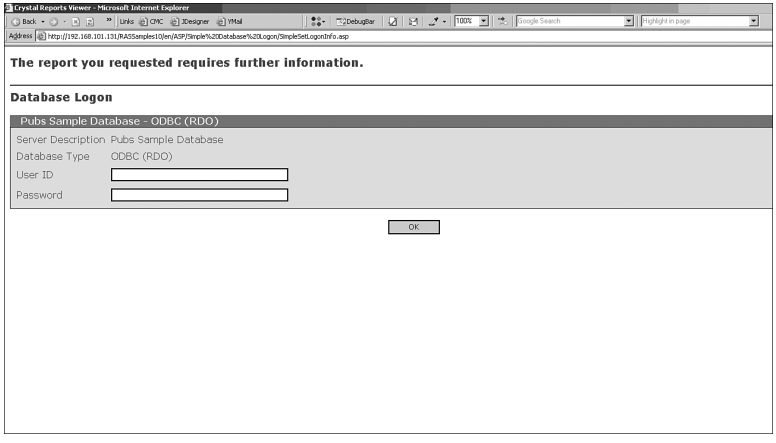


Figure 31.7
The Database Logon example provides a good starting point for understanding database logons and Crystal Reports.

Examples discussed in this section are included with the out-of-the box installation of the RAS (Crystal Enterprise Embedded in version 10) and can be leveraged by reusing the source code distributed as part of the installation.

DHTML Report Design Wizard

When RAS is installed and licensed as part of Crystal Enterprise Professional or Premium editions, it presents a DHTML wizard application through Crystal Enterprise that enables end users to create new or modify existing reports that are published in Crystal Enterprise. This integrated Crystal Enterprise action provides the end user with a series of dialogs that step through common report creation/modification tasks like adding fields, groups, filters, sorts, charts, and the application of report templates to existing reports stored in the Crystal Enterprise system.

Take a closer look at the wizard user interface. When you select the `modify` action on a given report, a dialog is displayed (see Figure 31.8) that enables you to select fields to be included in the newly modified report.

After you have selected the fields you want, click the Next button to open a Grouping dialog (see Figure 31.9) that enables the end user to specify groupings to be included in the report.

If at least one group has been selected for the newly modified report, a Summaries dialog is presented next (see Figure 31.10). This enables the addition of summaries to the new

report. To accomplish this, an end user defines the type of summary field (such as Sum, Average, Count, Max, or Min), the field to perform the summary on, and the group to calculate the involved summary on. Clearly, multiple summaries can be added to the newly created report in this dialog. Note that different types of summaries are presented based on the type of field selected (for example, Sum does not appear if a string field is selected).

Figure 31.8
The Field Selection dialog of the Modify Report Wizard enables user selection of fields from an existing Business View or Crystal Report.

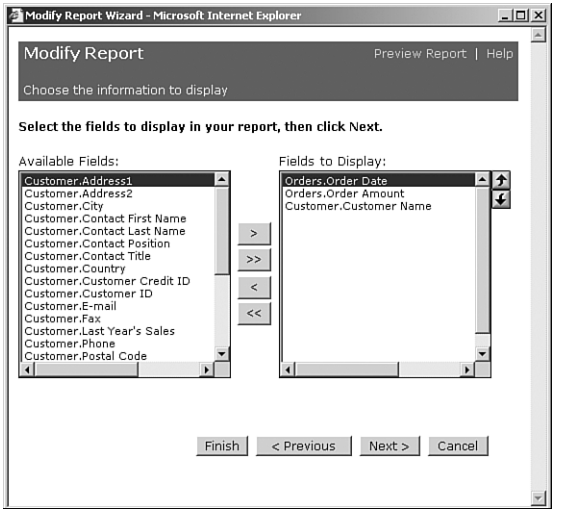
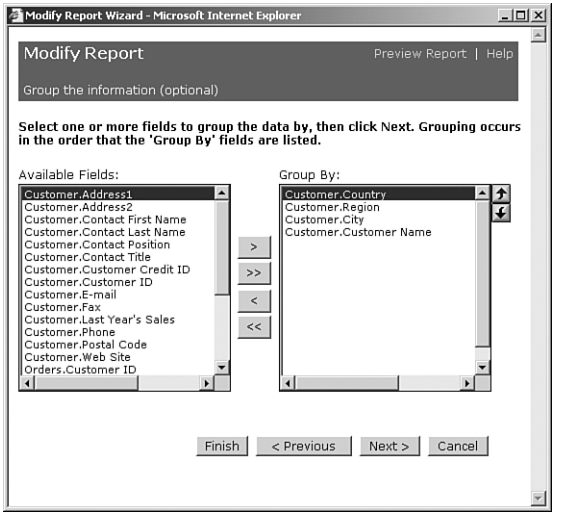


Figure 31.9
The Group Selection dialog of the Modify Report Wizard enables end users to add or change report groupings.



After you click the Next button, you are presented with a choice to override the default group sorting that is added when grouping the data in the report. Group sorting can be based on the actual Group names or the Summary fields that are calculated for that group.

When performing the sort on the latter, you can specify Top and Bottom N sorts or an All Records sort through the drop-down boxes in the dialog. Also in this dialog, you can specify the sort order for the detail level fields included in the detail section of the report (see Figure 31.11).

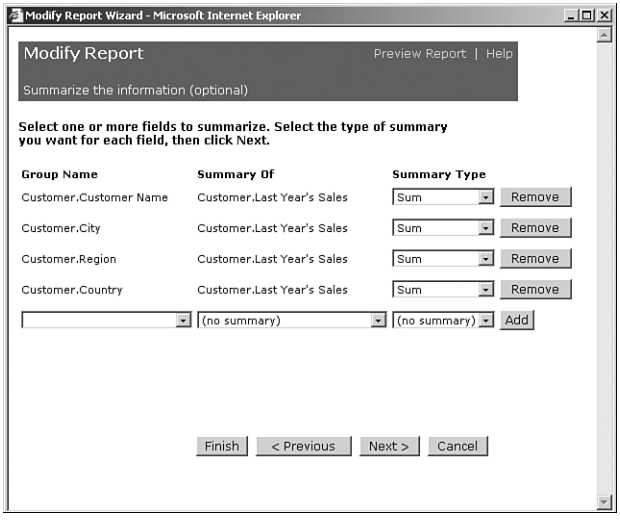


Figure 31.10
The Summary Selection dialog of the Modify Report Wizard enables the end user to add or change field summaries to reports.

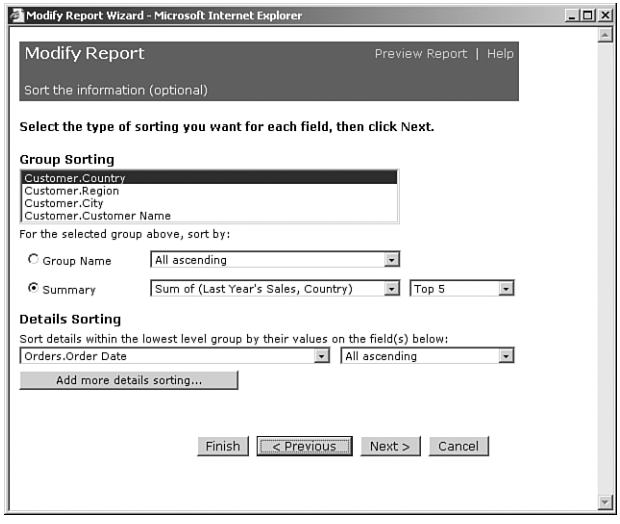


Figure 31.11
The Sorting Selection dialog of the Modify Report Wizard enables end users to add or change field sorting on the report.

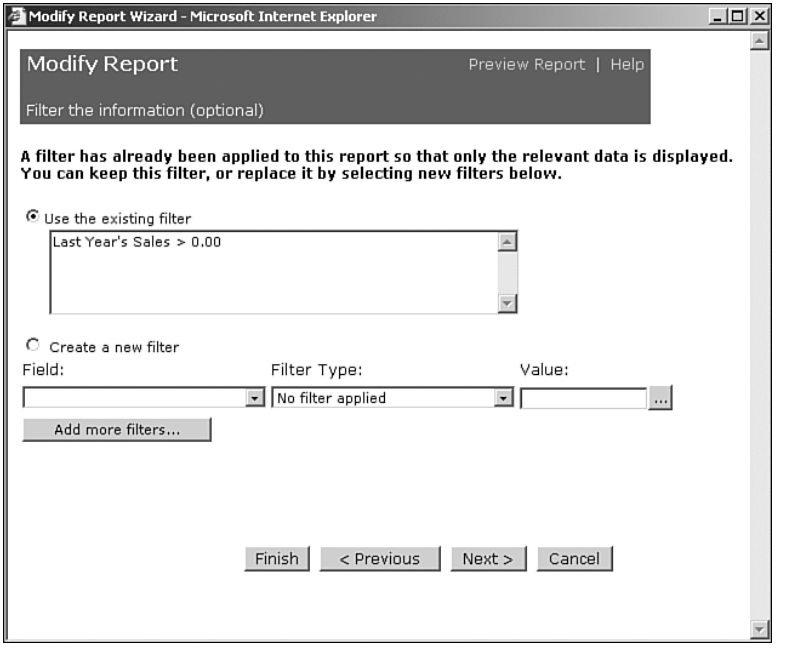
The next step is the option to apply record filters. *Filters* are used to limit the data that the report displays. This dialog enables you to see any filters that are already defined and append new filters through the provided text box or the provided drop-down boxes.

NOTE

It is important to note that although the default wizard uses the AND operator to join multiple filters, a small customization to this wizard would enable the end user to specify the operator on multiple joins. As you move through the following two chapters, it should become more clear how this type of customization would take place.

The RAS-based DHTML wizard also enables the end user to place a chart in the report header or footer section. The Chart Type is selected in the dialog presented in Figure 31.12 and the end user can customize the chart by adding a title to the chart and specifying the summary data that is actually charted (see Figure 31.13).

Figure 31.12
The Filter Selection dialog of the Modify Report Wizard enables the end user to select filters for the modified report.



One last screen that is presented for optional use is the report template specification screen. In this dialog, you can apply an existing report template to the newly created report. Report templates were covered in Chapter 14, "Designing Effective Report Templates."

After the report is defined to fit your needs, you can preview it immediately using the Preview Report link on any of the DHTML wizard screens. If the newly modified report is of sufficient value to keep as a new report, you can save the report in any Crystal Enterprise folder that you have been granted access to within the Crystal Enterprise Professional/Premium security model.

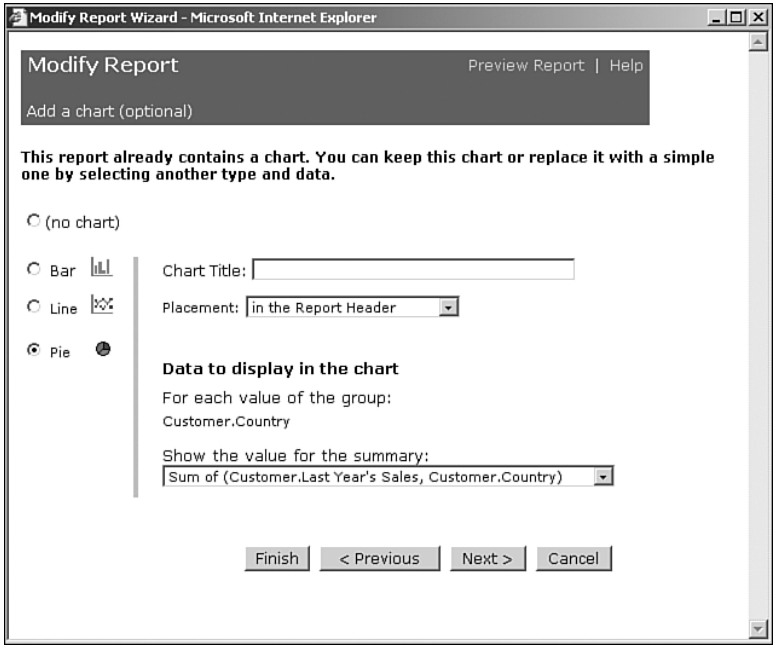


Figure 31.13
The Charting Selection dialog of the Modify Report Wizard.

LEVERAGING THE OPEN SOURCE NATURE OF THE SAMPLE APPLICATIONS

It is worth pointing out as a reminder that all the functionality that is provided through the step-wise DHTML Report Design Wizard just described is provided through a single application called the Crystal Reports Explorer in version 10. (In previous versions it was called the Crystal Ad-Hoc Application and is often still referenced as such.) This application is provided with Crystal Enterprise Premium or Crystal Enterprise Professional with the Report Explorer Add-in.

For more information, see Chapter 21, "Ad-Hoc Application and Excel Plug-in for Ad-Hoc and Analytic Reporting."

It is also instructive to understand that both the DHTML Report Design Wizard and the Crystal Reports Explorer introduced in Chapter 21 are simply working examples of the Crystal Enterprise Embedded (or RAS) functionality in action. These applications are provided open-source and provide great starting points for rapid application development and customization. The next two chapters introduce the RAS object models and their capability to integrate report creation and modify functionality into either Java or .NET/COM-based applications.

TROUBLESHOOTING

DISTRIBUTING THROUGH THE WEB

I want to distribute Crystal Reports through the Web but I need to scale beyond the built-in simultaneous request limit of 3 provided with Crystal Reports Advanced.

Install Crystal Enterprise Embedded and either use one of the sample Web reporting applications or include programmatic access to the object model in your applications.

In cases where there have been legacy applications written in previous versions of the Crystal product set that make use of the Report Designer Component (RDC) or the Crystal Print engine (CRPE), Crystal Enterprise Embedded edition should be considered the logical migration path. This is particularly true when you're developing applications that require Web access. This version of the product leverages the advantages of the client/server architected components in RAS that are specifically written for the Web. In the following two chapters the common Crystal Viewers SDK and the functionality in the RAS SDK are presented in increasing detail.

CHAPTER 32

CRYSTAL ENTERPRISE—VIEWING REPORTS

In this chapter

VIEWING REPORTS OVER THE WEB

This chapter introduces programmatic access to viewing reports over the Web through the Crystal Enterprise SDKs. It is important to note that these viewers and the means to programmatically access them have been made consistent across the Business Objects Crystal product line—including Crystal Reports, Crystal Enterprise Embedded, and Crystal Enterprise Professional and Premium editions. This consistency across products enables a seamless and rapid migration through the different versions of the Crystal product suite as developer's application requirements grow. These SDKs are provided in Java, .NET, and COM flavors and provide rich functionality that can be integrated into both intranet and extranet targeted applications.

This chapter introduces the different Crystal Report viewer components and explains how to set them up for inclusion in your custom applications. The following topics are covered:

- Introduction to the Crystal Report viewers
- Understanding the report source
- Implementing the Page viewer and toolbar buttons
- Implementing the Part viewer
- Implementing the Interactive viewer and toolbar buttons
- Implementing the Grid viewer and toolbar buttons
- Using the Export Control

INTRODUCTION TO THE CRYSTAL REPORT VIEWERS

The Crystal Report viewers that ship with Report Application Server break into four different categories to suit the need of a variety of applications: the Page viewer, Part viewer, Interactive viewer, and Grid viewer. Although all four viewers offer unique capabilities, they share a common API and set of basic features. Each viewer allows the developer to indicate which report to display, supply database logon credentials, apply report parameters, and export the report. All four viewers are exposed as server-side controls and as a result, output dynamic HTML that is rendered in any Web browser. No special software is required on the client's machines to view reports using any of the viewers.

Listed below is a short description about each report viewer:

- Page viewer: The standard report viewer component. It displays reports in a paginated fashion. A toolbar along the top allows access functions like page navigation, printing, exporting, zooming, and text searching.
- Part viewer: A report viewer component that renders just individual elements of a report. This is useful for portal-style applications where only a small portion of the screen is reserved for report viewing.

- Interactive viewer: Looks and acts identical to the Page viewer but exposes an extra toolbar button that provides an additional user interface for doing data-level searching within the report.
- Grid viewer: A viewer component that just displays the data from the report in a grid without any layout or formatting applied.

The means with which all of these viewers interact with the reports themselves is a mechanism called the report source. The following section describes the report source in detail.

UNDERSTANDING THE REPORT SOURCE

Because the Crystal Report viewer components are shared across both the Crystal Enterprise Embedded Edition and the Crystal Enterprise Professional/Premium editions, there must be a common interface defined so the viewer can display reports generated using both types of report processing engines. This interface is called the *report source*. The report source is an object that both the Embedded edition and Professional/Premium editions supply that the viewer in turn communicates with to render the reports to the various forms of HTML.

There are three types of report sources:

- Standalone Report Application Server: This is packaged as the Crystal Enterprise Embedded Edition.
- Clustered Report Application Server: This is packaged as part of Crystal Enterprise Professional/Premium edition. This is the Report Application Server running as a service on the Crystal Enterprise framework.
- Page Server: This is the primary report processing service as part of the Crystal Enterprise framework.

NOTE

Because of the special functionality of the Interactive and Grid viewers, they do not work with a Page Server–based report source.

The Java code in Listing 32.1 illustrates the first scenario where a report source object is obtained from the standalone Report Application Server.

LISTING 32.1 OBTAINING A REPORT SOURCE FROM A REPORT FILE

```
//First you must create a new ReportClientDocument object
ReportClientDocument reportClientDoc = new ReportClientDocument();

//After the ReportClientDocument is created, you then need to
//specify the report file that is to be used as the report
```

continues

LISTING 32.1 CONTINUED

```
//source:
String path = "C:\\Program Files\\Crystal Decisions\\Report Application Server" +
              " 10\\Reports\\Sample.rpt";
reportClientDoc.open(path, openReportOptions._openAsReadOnly);

//Finally use the openReportSource method to return the report source object
IReportSource reportSource = reportClientDoc.getReportSource();
```

NOTE

All the code listings provided in this chapter are provided in JSP/Java. Although the .NET/COM and the Java flavors of the RAS SDK share identical functionality, there are obviously language nuances associated with each of them. Code samples for additional language flavors are available for download from the `www.usingcrystal.com` Web site.

Listing 32.2 illustrates obtaining a report source when using the Report Application Server as part of Crystal Enterprise. Notice that the same `ReportClientDocument` object is used. The difference is in how the `ReportClientDocument` object is obtained.

→ For more information on using the `IEnterpriseSession` as associated Crystal Enterprise objects, **see "Establishing a Crystal Enterprise Session," p. 767**

LISTING 32.2 OBTAINING AN `EnterpriseSession` OBJECT

```
//Retrieve the IEnterpriseSession object previously stored in the user's session.
IEnterpriseSession enterpriseSession =
        (IEnterpriseSession) session.getAttribute("EnterpriseSession");

//Use enterpriseSession object to retrieve the reportAppFactory object
IReportAppFactory reportAppFactory =
        (IReportAppFactory) enterpriseSession.getService("", "RASReportFactory");

//Open the report document by specifying the report ID
ReportClientDocument reportClientDoc = reportAppFactory.openDocument(reportID,
   0, Locale.ENGLISH);

//Finally use the openReportSource method to return the report source object
IReportSource reportSource = reportClientDoc.getReportSource();
```

An alternative way to do this is shown in Listing 32.3.

LISTING 32.3 ALTERNATIVE METHOD TO OBTAIN AN `EnterpriseSession` OBJECT

```
//Retrieve the IEnterpriseSession object.
IEnterpriseSession enterpriseSession =
        (IEnterpriseSession) session.getAttribute("EnterpriseSession");

//Use the IEnterpriseSession object's getService method to get
```

```
//an IReportAppFactory object.
IReportSourceFactory reportFactory =
        (IReportSourceFactory) enterpriseSession.getService("",
            "RASReportFactory");

//Use IReportAppFactory object's openReportSource method, passing it
//the report ID to return the reportSource object
IReportSource reportSource = reportFactory.openReportSource(reportID,
        Locale.ENGLISH);
```

Listing 32.4 illustrates obtaining a report source object from the Page Server service from Crystal Enterprise Professional/Premium.

LISTING 32.4 UTILIZING THE PAGE SERVER TO OPEN A REPORT

```
//Retrieve the IEnterpriseSession object previously stored in the user's session.
IEnterpriseSession enterpriseSession =
        (IEnterpriseSession) session.getAttribute("EnterpriseSession");

//Use the getService method of the EnterpriseSession object to obtain an
//IReportAppFactory object:
IReportSourceFactory reportFactory =
        (IReportSourceFactory) enterpriseSession.getService ("",
"PSReportFactory");

//Finally use the openReportSource method to return the report source object
IReportSource reportSource = reportFactory.openReportSource(reportID,
        Locale.ENGLISH) ;
```


IMPLEMENTING THE PAGE VIEWER

The first viewer component to be covered is the Page viewer, as illustrated in Listing 32.5. To use this viewer, you will create its `CrystalReportViewer` object. It, along with all the other viewers, exposes a method called `setReportSource` that accepts a valid report source object as obtained from the description in the previous section. Finally, again like the other viewers, it has a `processHttpRequest` method that accepts references to the current servlet context. This method does the actual rendering to HTML.

LISTING 32.5 VIEWING A REPORT OVER THE WEB

```
//To create a Java report viewer you need to instantiate a CrystalReportViewer
//object. To create a CrystalReportViewer object:
CrystalReportPartsViewer viewer = new CrystalReportViewer();

//Obtain a ReportSource object. Set the viewer's report source by calling its
//setReportSource method.
viewer.setReportSource(reportSource);

//When you have created and initialized a Java report page viewer, you call
//its processHttpRequest method to launch it in a Web //browser.
viewer.processHttpRequest(request, response, getServletContext(), null);
```

Figure 32.1 shows the output of this code.

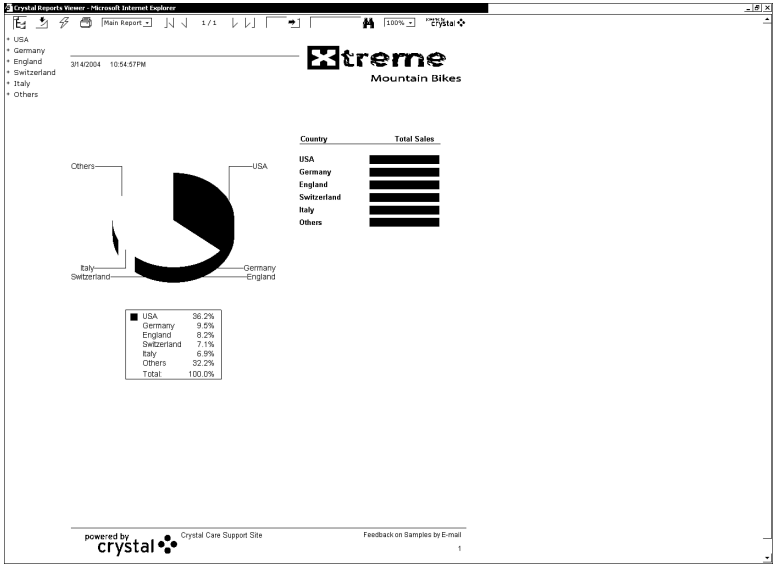


Figure 32.1
A report being displayed in an HTML page viewer.

All viewers including the Page viewer share a number of toolbar elements. These properties can be programmatically toggled and are displayed in Table 32.1. All the viewer properties must be set before calling the `ProcessHTTPRequest` method that displays the selected report. For example, to ensure the Crystal logo is displayed when the involved report is viewed, the code line

```
Viewer.HasLogo(true);
```

needs to be included in the code before the `processHTTPRequest` method is called.

As the different viewers are introduced and discussed later in this chapter, some additional elements pertinent to the viewer being discussed will be displayed in that section's table.

TABLE 32.1 TOOLBAR ELEMENTS (PAGE VIEWER)

Property	Property Description
`HasLogo`	Includes or excludes the "Powered by Crystal" logo when rendering the report.
`HasExportButton`	Includes or excludes the export button when rendering the report.
`HasGotoPageButton`	Includes or excludes the Go to Page button when rendering the report.
`HasPageNavigationButtons`	Includes or excludes the page navigation buttons when rendering the report.

Property	Property Description
HasPrintButton	Includes or excludes the Print button when rendering the report.
HasRefreshButton	Includes or excludes the Refresh button when rendering the report.
HasSearchButton	Includes or excludes the Search button when rendering the report.
HasToggleGroupTreeButton	Includes or excludes the Group Tree toggle button when rendering the report.
HasViewList	Specifies whether the viewer should display a list of previous views of the report.
SetPrintMode	Set printing to use PDF or Active X printing (0=pdf, 1=actx).
HasZoomFactorList	Specifies zoom factor for displayed report.

IMPLEMENTING THE PART VIEWER

The Part viewer works much the same way as the Page viewer—in fact, much of the code is exactly the same, except for the type of viewer object that is created. Listing 32.6 assumes that the report to be displayed has an initial report part defined in the report itself.

LISTING 32.6 VIEWING A REPORT USING THE REPORT PART VIEWER

```
//To create a Java report part viewer you need to instantiate a
//CrystalReportPartsViewer object:
CrystalReportPartsViewer viewer = new CrystalReportPartsViewer();

//Obtain a ReportSource object. Set the viewer's report source by calling its
//setReportSource method
viewer.setReportSource(reportSource);

//After you have created and initialized a Java report part viewer, you
//call its processHttpRequest method to launch it in a Web browser.
viewer.processHttpRequest(request, response, getServletContext(), null);
```

If a report part is not defined for a report, or if the default part needs to be overridden, Listing 32.7 provides code that can be used to manipulate the `ReportParts` collection. Figure 32.2 shows the output of this page being displayed in a Web browser.

LISTING 32.7 SPECIFYING REPORT PART NODES

```
//To create a Java report part viewer you need to instantiate a
//CrystalReportPartsViewer object:
CrystalReportPartsViewer viewer = new CrystalReportPartsViewer();

//After you have created the CrystalReportPartsViewer object,
//you must specify the report parts that you want to display when the
```

continues

LISTING 32.7 CONTINUED

```
//viewer is launched.  To specify the report parts that you want the
//viewer to display Create a ReportPartsDefinition object.
ReportPartsDefinition partsDefinition = new ReportPartsDefinition();

//Get the collection of ReportPartNodes that belong to the
//ReportPartsDefinition.
ReportPartNodes reportPartNodes = partsDefinition.getReportPartNodes();

//Create a corresponding ReportPartNode object for each report part that
//you would like the viewer to display. Add these objects to the
//ReportPartNodes collection. Part1 is being used here as the default
//Report Part to display
ReportPartNode node0 = new ReportPartNode();
node0.setName("Part1");
partsDefinition.getReportPartNodes().add(node0);

//Obtain a ReportSource object. Set the viewer's report source by calling
//its setReportSource method
viewer.setReportSource(reportSource);

//Call the viewer's setReportParts method, passing it the ReportPartsDefinition.
viewer.setReportParts(partsDefinition);

//After you have created and initialized a Java report part viewer,
//you call its processHttpRequest method to launch it in a Web browser.
viewer.processHttpRequest(request, response, getServletContext(), null) ;
```

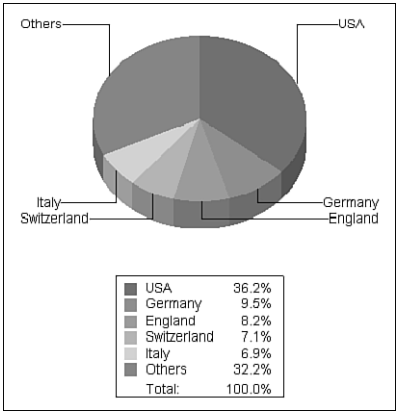



Figure 32.2
The Report Part viewer displaying a report part.

IMPLEMENTING THE INTERACTIVE VIEWER

The Interactive viewer works almost exactly like the Page viewer. In fact, the Interactive viewer component derives from the Page viewer component, so it inherits all the base functionality. What it adds is a new toolbar button that enables an advanced searching User Interface inside the viewer. This is useful for larger reports and for end users requiring advanced searches where simple text string searching is not suitable. The Interactive viewer allows the report to be filtered using a specified record selection criteria.

Listing 32.8 shows a report being viewed by the Interactive viewer. Note that the `setOwnPage` method is called to indicate that the viewer owns the entire page, which is generally a good thing to do when using this viewer.

LISTING 32.8 USING THE REPORT PART VIEWER IN CODE

```
//To create a Java interactive viewer you instantiate a
//CrystalReportInteractiveViewer object:
CrystalReportInteractiveViewer viewer = new CrystalReportInteractiveViewer();

//Set the viewer's report source by calling its setReportSource method
viewer.setReportSource(reportSource);

//Enable the Advanced Search Wizard.
viewer.setEnableBooleanSearch(true);

//Set the setOwnPage property to true. The setOwnPage property should always
//be set to true for the interactive viewer.
viewer.setOwnPage(true);

//After you have created and initialized a Java interactive viewer,
//you call its processHttpRequest method to launch it in a Web browser.

viewer.processHttpRequest(request, response, getServletContext(), null);
```

Figure 32.3 shows a report being displayed in the Interactive viewer and the advanced searching UI being used.

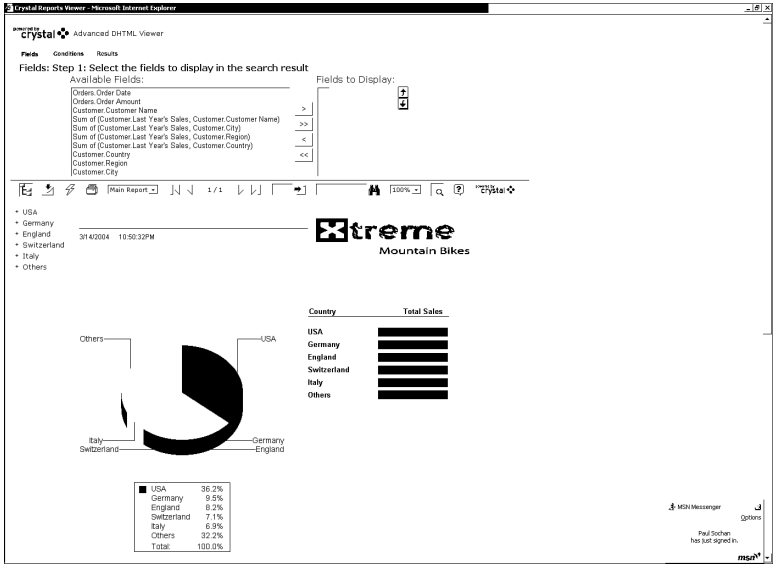


Figure 32.3
The Interactive viewer in action.

All viewers including the Interactive viewer share a number of toolbar elements. These properties can be programmatically toggled and are displayed in Table 32.2. All the viewer

properties must be set before calling the `ProcessHTTPRequest` method that will display the selected report. For example, to ensure the Crystal logo is displayed when the involved report is viewed, the code line

```
Viewer.HasLogo(true);
```

needs to be included in the code before the `processHTTPRequest` method is called.

TABLE 32.2 TOOLBAR ELEMENTS (INTERACTIVE VIEWER)

Property	Property Description
`HasLogo`	Includes or excludes the "Powered by Crystal" logo when rendering the report.
`HasExportButton`	Includes or excludes the Export button when rendering the report.
`HasGotoPageButton`	Includes or excludes the Go to Page button when rendering the report.
`HasPageNavigationButtons`	Includes or excludes the page navigation buttons when rendering the report.
`HasPrintButton`	Includes or excludes the Print button when rendering the report.
`HasRefreshButton`	Includes or excludes the Refresh button when rendering the report.
`HasSearchButton`	Includes or excludes the Search button when rendering the report.
`HasToggleGroupTreeButton`	Includes or excludes the Group Tree toggle button when rendering the report.
`HasViewList`	Specifies whether the viewer should display a list of previous views of the report.
`SetPrintMode`	Set printing to use PDF or Active S printing (0=pdf, 1=actx).
`HasZoomFactorList`	Specifies zoom factor for displayed report.
`HasBooleanSearchButton`	Includes or excludes the toggle Boolean search button when rendering the report. Unique to Interactive viewer.
`HasHeaderArea`	Includes or excludes the header area when rendering the report. Unique to Interactive viewer.
`HasPageBottomToolbar`	Includes or excludes the page bottom toolbar. Unique to Interactive viewer.

IMPLEMENTING THE GRID VIEWER

The final viewer to be covered in this chapter is the Grid viewer. The Grid viewer (shown in Figure 32.4) differs more from the other viewers in that it does not render the

report's presentation onscreen. Instead it looks at the dataset associated with the report (that is, the query result after the report engine has done its magic) and displays that data in a tabular fashion. This opens up some very interesting scenarios if you use your imagination.

NOTE

> You can override the style of the grid table by defining a stylesheet that maps to the styles used by the grid object. Consult the documentation for more information on this.

Listing 32.9 shows a report being displayed using the Grid viewer.

LISTING 32.9 DISPLAYING A REPORT IN THE GRID VIEWER

```
//To create a Java grid viewer you need to instantiate a GridViewer object.
//To create a GridViewer object:
GridViewer viewer = new GridViewer();

//Set the viewer's report source by calling its setReportSource method
viewer.setReportSource(reportSource);

//After you have created and initialized a Java grid viewer object, you call
//its processHttpRequest method to display the results in the Web page
viewer.processHttpRequest(request, response, getServletContext(), null);
```

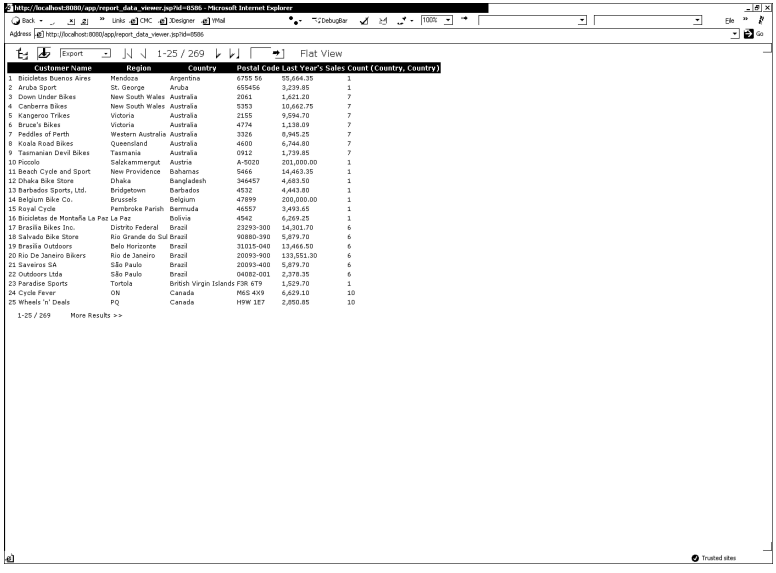



Figure 32.4
The Grid viewer in action.

All viewers including the Grid viewer share a number of toolbar elements. These properties can be programmatically toggled and are displayed in Table 32.3. All the viewer properties must be set before calling the `ProcessHTTPRequest` method that will display the selected

report. For example, to ensure the Crystal logo is displayed when the involved report is viewed, the code line

```
Viewer.HasLogo(true);
```

needs to be included in the code before the `processHTTPRequest` method is called.

TABLE 32.3 TOOLBAR ELEMENTS (GRID VIEWER)

Property	Property Description
`HasLogo`	Includes or excludes the "Powered by Crystal" logo when rendering the report.
`HasExportButton`	Includes or excludes the Export button when rendering the report.
`HasGotoPageButton`	Includes or excludes the Go to Page button when rendering the report.
`HasPageNavigationButtons`	Includes or excludes the page navigation buttons when rendering the report.
`HasPrintButton`	Includes or excludes the Print button when rendering the report.
`HasRefreshButton`	Includes or excludes the Refresh button when rendering the report.
`HasSearchButton`	Includes or excludes the Search button when rendering the report.
`HasToggleGroupTreeButton`	Includes or excludes the Group Tree toggle button when rendering the report.
`HasViewList`	Specifies whether the viewer should display a list of previous views of the report.
`SetPrintMode`	Set printing to use PDF or Active X printing (0=pdf, 1=actx).
`HasZoomFactorList`	Specifies zoom factor for the displayed report.
`DisplayNavigationBar`	Specifies whether the viewer should display the navigation bar at the bottom of the grid. Unique to Grid viewer.
`DisplayRowNumberColumn`	Specifies whether to display the row number column. Unique to Grid viewer.

Property	Property Description
`DisplayToolbarFindRowButton`	Includes or excludes the Find Row button when rendering the toolbar. Unique to Grid viewer.
`DisplayToolbarGroupViewList`	Specifies whether the viewer should display the view list. Unique to Grid viewer.
`DisplayToolarSwitchViewButton`	Includes or excludes the Toggle Grid View button. Unique to Grid viewer.
`EnableGridToGrow`	Specifies whether the viewer should enable the Grid to Grow. Unique to Grid viewer.
`GridViewMode`	Specifies the viewer View mode. Unique to Grid viewer.
`MatchGridandToolbarWidth`	Specifies whether the table should align with the toolbar. Unique to Grid viewer.
`TableStyle`	Specifies the style class of the table. You can apply a css style class to the grid table that shows records. You do so by stating: `Gridviewer.TableStyle="cssclass";` Unique to Grid viewer.
ToolbarStyle	Specifies the style class of the toolbar. You can apply a css style class to the grid toolbar. You do so by stating: `Gridviewer.ToolbarStyle="cssclass";` Unique to Grid viewer.

USING THE EXPORT CONTROL TO DELIVER REPORTS IN OTHER FORMATS

So far all the scenarios that have been discussed in this chapter have involved displaying reports in dynamic HTML format. Although this is a great report delivery method for most scenarios, there are times when reports need to be exported to various other file formats. Although this can be accomplished by using the `ReportClientDocument` object model, there is an easier way to do this: using the Export control.

Listing 32.10 shows how the Export control would be used to export a report to PDF. Notice that the Export control has the concept of the report source of the `processHttpRequest` method.

LISTING 32.10 EXPORTING A REPORT VIA CODE

```
//Instantiate a ReportExportControl object
ReportExportControl exportControl = new ReportExportControl();

//After you have created the ReportExportControl object, you must
//specify the export format that you want to export the report to. To
//specify the export format create an ExportOptions object:

ExportOptions exportOptions = new ExportOptions();

//Specify the export format by calling the ExportOptions object's
//setExportFormatType method, passing it the integer constant that
//represents the chosen format:
exportOptions.setExportFormatType(ReportExportFormat.PDF);

//To initialize an Export control in a Crystal Enterprise environment set
//the control's report source by calling its setReportSource
method.exportControl.setReportSource(reportSource);

//Call the control's setExportOptions method, passing it an ExportOptions object
exportControl.setExportOptions(exportOptions);

//You may also want to call the setExportAsAttachment method, passing it the
//Boolean value true. The Export control will then display a dialog box
//that allows users of your Web application to save the exported report before
//they open it: exportControl.setExportAsAttachment(true);

//To initialize an Export control in an unmanaged RAS environment set the
//control's report source by calling its setReportSource method and passing
//the method a reference to a report source object.
exportControl.setReportSource(reportSource);

//Call the control's setExportOptions method, passing it an ExportOptions object
exportControl.setExportOptions(exportOptions);

//You may also want to call the setExportAsAttachment method, passing it the
//Boolean value true. The Export control will then display a dialog box
//that allows users of your Web application to save the exported report before
//they open it:
exportControl.setExportAsAttachment(true);

//After you have created an export control, you call its processHttpRequest
//method to complete the export.
exportControl.processHttpRequest(request, response, getServletContext(), null) ;
```

Figure 32.5 shows a report being exported to PDF.

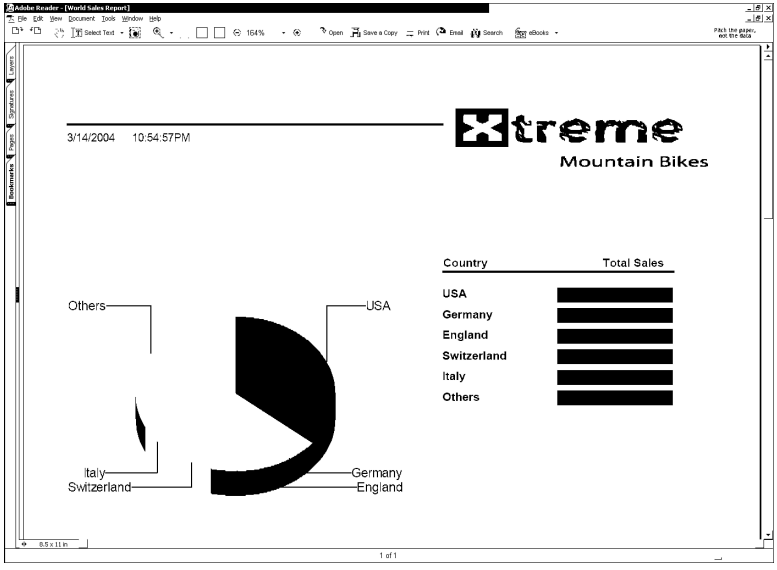


Figure 32.5
Using the Export control to export a report to PDF.

TROUBLESHOOTING

REPORT VIEWING PERFORMANCE IS SLOW

What efficiencies can I add to increase the performance of my application?

Caching a report source in the session variable allows it to be used multiple times efficiently. When a report source is not cached, the process of creating a new report source multiple times becomes fairly expensive. Furthermore, caching a report source allows reports with or without saved data to be refreshed.

Listing 32.11 shows how to store and retrieve the report source object from session state.

LISTING 32.11 CACHING A REPORT SOURCE OBJECT

```
//To store the report source in a session variable
request.getSession().settAttribute("RptSrc",reportSource);

//To retrieve the report source from a session variable
rptSrc = request.getSession().getAttribute("RptSrc");
```

THE VIEWER NEEDS TO WORK IN A PAGE WITHOUT A FORM ELEMENT

How can I control how the viewer interacts with a surrounding form?

If your Web page contains only the viewer and nothing else, several things can be done that can simplify the report viewing implementation. The viewer is capable of generating

complete HTML pages and can set the appropriate page properties depending on the viewing context. Setting the `setOwnPage` property to true provides several benefits. Allowing the viewer to handle the surrounding HTML content reduces the amount of code you need to add to your Web page and allows the viewer to automatically determine certain settings. It correctly sets the content-type and `charset` information for the page. This ensures that pages containing international characters will be displayed correctly. When `setOwnPage` is set to true, you must use the `processHttpRequest` method to display the report instead of `getHtmlContent`. The `processHttpRequest` method must be used because using `getHtmlContent` has the same effect as setting `setOwnPage` to false, negating any of the benefits gained from setting `setOwnPage` to true. If your Web page does not contain any controls that require post back, you should set the `setOwnForm` method to true. Doing so allows the viewer to handle the view state information automatically. The view state is used to perform client-side caching of information about the current state of the report. If you have other controls on the page, you must set `setOwnForm` to false and handle the view state information manually.

THE CHARACTER SET IS DISPLAYING INCORRECTLY

How can I indicate which unicode character to set should be used for the report viewing session?

To send characters from a Web page to a Web browser, you must use the correct encoding. Always specify the correct content-type and character set for all your Web pages. If your Web page returns content to a standard HTML browser, the following lines will ensure that the correct character set is defined. The `contentType` and `charset` directives let the browser know how the returned HTML page is encoded. UTF-8 is the recommended standard character set if it is available for your target client browser. For more information, consult the Release Notes or the vendor for your target client browser.

CHAPTER 33

Crystal Enterprise Embedded—Report Modification and Creation

In this chapter

INTRODUCTION

This chapter covers the capability of the Report Application Server (RAS) SDK to create and modify reports. Topics include

- RAS environments
- Loading report files
- RAS component locations
- Installing the RAS SDK
- Exception handling
- Programming with the RAS SDK

DEPLOYING RAS ENVIRONMENTS

The RAS functions in two different environments—either as a standalone report processing and modification server or as a service of the Crystal Enterprise framework.

USING RAS IN A CRYSTAL ENTERPRISE ENVIRONMENT

Crystal Enterprise provides a framework for delivering enterprise reporting. RAS adds the capability for users to modify reports stored in Crystal Enterprise. In this scenario, the Crystal Enterprise framework manages the RAS. Multiple instances of the RAS can be added and Crystal Enterprise will load balance between them.

→ For additional information on the Crystal Enterprise Framework, **see** "The Server Tier: Introduction to the Crystal Enterprise Framework," **p. 510**

USING RAS IN A STANDALONE ENVIRONMENT

Even without Crystal Enterprise, the capabilities of RAS can be leveraged. In this scenario the RAS SDK modifies unmanaged reports (reports stored on the file system rather than in Crystal Enterprise) so you refer to the RAS as an unmanaged server.

NOTE

The core functionality of the RAS is the same regardless of the environment it is running in. The rest of this chapter assumes the RAS is running in unmanaged mode as a standalone server.

LOADING REPORT FILES

Crystal Report (.rpt) files must first be placed in a folder designated as the Report Directory for the RAS. The RAS interacts only with reports in this folder and its subfolders. An "access denied" error results from attempting to access reports outside of this folder.

The RAS Configuration Manager sets the Report Directory. In a default installation, the Report Directory points to the location of the sample reports folder.

After setting the report directory, there are several ways to designate a report file.

For example, the `ras` prefix can be used:

```
ras://D:\directory\reportname.rpt
```

The location on the RAS machine of `reportname.rpt` is given by this string.

It is not necessary to always key in the `ras` prefix because it is assumed by default; for example, `c:\reportname.rpt` is assumed to be a path on the RAS machine.

The `rassdk` prefix also can be used like this:

```
rassdk://c:\directory\reportname.rpt
```

Where RAS SDK is running on the machine, generally the Web application server, this string gives the location of `reportname.rpt`.

Because reports must be serialized and sent to the RAS for processing each time they are accessed, reports stored outside the RAS slow down performance. Therefore, to improve performance it is highly recommended that a local folder is used on the RAS machine for the Report Directory.

LOCATING RAS COMPONENTS IN A NETWORK ARCHITECTURE

Though the default installation places both the RAS SDK and RAS on the local machine, they can be installed on separate computers.

SPECIFYING SEPARATE RASS

The server attribute of the clientSDKOptions.xml file defines the location of the RAS. This file is created in the jar folder under `Program Files\Common Files\Crystal Decisions` during the RAS installation. This file can be modified to indicate the location of the RAS when installed separately. This can also be done programmatically using the RAS SDK, which will be illustrated by examples later in the chapter.

The location of the clientSDKOptions.xml file can be specified either statically, by setting a classpath that points to the file, or dynamically, by specifying the location of this file. Load balancing can be enabled where there are multiple RASs by specifying the location of all the RASs on the network in the clientSDKOptions.xml file.

SETTING A STATIC LOCATION

Adding the file path of the file clientSDKOptions.xml to the Web application server's `CLASSPATH` environment variable specifies the location of the file statically. The file path might also need to be added to the `CLASSPATH` on the local system. More information

regarding adding classpaths might be found on the Web server's documentation information.

DEPLOYING RAS IN A DYNAMIC LOCATION

The location of the clientSDKOptions.xml file can be specified at runtime. From the JSP or Java files use the Java method `setProperty` from the System class. Set the system property indicated by the ras.config key to the specified directory as follows:

```
system.setProperty("ras.config","c:/temp")
```

This specifies that to locate RAS servers, the clientSDKOptions.xml file in `c:/temp` will be used. Using the web.xml file (located by default in the `\WEB-INF\` directory of your Web application) to specify the location of the clientSDKOptions.xml will avoid hard-coding the location for the clientSDKOptions.xml file throughout your program.

INSTALLING THE RAS SDK

As mentioned previously, the RAS SDK JAR files can be found in the jar folder by default. Copy the RAS and Crystal Enterprise .jar files to the appropriate folder on the application server being used. If you are using Apache Tomcat, for example, move the .jar files to the Web application's `WEB-INF\lib` folder. Configuring a Web server to access the SDK JAR files might take additional steps, detailed in the installation help files provided with the RAS.

BEST PRACTICES IN RAS EXCEPTION HANDLING

Options for displaying and logging exception information can also be specified. These tasks can be performed by modifying the web.xml file (located by default in the `\WEB-INF\` directory of your Web application) as follows.

DISPLAYING EXCEPTIONS

Three options exist for displaying exception information to the user. Setting the `crystal_exception_info` parameter to one of the following values determines how exceptions are handled:

- `short`—The exception information is displayed without the accompanying stack trace.
- `long`—The exception information is displayed with the accompanying stack trace.
- `disable`—The exception information is not displayed; the user must handle the exception.

The following code shows an example of the exception display configuration:

```
<context-param>
    <param-name>crystal_exception_info</param-name>
        <param-value>long</param-value>
    <description>
        Options for displaying exception information.
```

```
        If this parameter is not set, the default value is short.
        It can be one of the following values: short, long, disable.
    </description>
</context-param>
```

The `crystal_exception_info` parameter is short by default. Modifying exception.css specifies the style and formatting of short messages.

LOGGING EXCEPTIONS

The option to turn exception logging either on or off can be set with the `crystal_exception_log_file` parameter. The exception information output to the log file will be in the long format regardless of the setting of the `crystal_exception_info` parameter. The following code shows an example of the exception logging configuration:

```
<context-param>
    <param-name>crystal_exception_log_file</param-name>
        <param-value>c:\temp\webreportingexception.log</param-value>
    <description>
        Set this parameter to log the exception in long form
        to the file specified.
        The value is the full path of the log file.
    </description>
</context-param>
```

When setting the parameter to the desired path of the log file, by default, exceptions are not logged.

THE RAS SDK IN ACTION

This section covers the common programming tasks associated with the RAS SDK. Although the SDK provides many capabilities, some of the following tasks are common to most programming exercises and are central to the SDK.

INITIALIZING AN RAS SESSION

Initiating a session with the RAS is the first step in programming with the RAS SDK. In this step, a specific RAS can be specified for use; otherwise, the system selects one from the RASs listed in the clientSDKOptions.xml file using a round-robin method. Initializing a RAS session by specifying a machine name at runtime is shown in the following code:

```
//Create a new Report Application Session
ReportAppSession reportAppSession = new ReportAppSession();
//Create a Report Application Server Service
reportAppSession.createService("com.crystaldecisions.sdk.occa.report.
➥application.ReportClientDocument");
//Set the RAS server to be used for the service. You can also use "localhost"
➥if the RAS server is running on
your local machine.
reportAppSession.setReportAppServer("MACHINE_NAME");
//Initialize RAS
reportAppSession.initialize();
//Create the report client document object
```

```
ReportClientDocument clientDoc = new ReportClientDocument();
//Set the RAS Server to be used for the Client Document
clientDoc.setReportAppServer(reportAppSession.getReportAppServer() );
```

All ReportClientDocument objects created from the same `ReportAppSession` communicate with the same RAS.

OPENING A REPORT

A report can be opened first by creating a new ReportClientDocument object and specifying the `ReportAppServer`. Then the open method can be used to open a report. This method takes two parameters:

- The absolute path and filename of the report
- A flag indicating how the file will be opened

See the `OpenReportOptions` class for valid report options.

NOTE

Reports are loaded from the report folder found at `\Program Files\Crystal Decisions\Report Application Server 10\Reports\` by default.

The following code opens a report:

```
try
{
    reportClientDocument.open("C:\MyReports\GlobalSales.rpt", 0);
}
catch (ReportSDKException e)
{
    // Handle the case where the report does not open properly.;
}
```


The previous chapter explained how to view reports using RAS. Creating and modifying those reports using the RAS SDK will be the focus of the remainder of this chapter.

ADDING FIELDS TO THE REPORT

A report can be modified after creating and opening a `ReportClientDocument` by using the report's controllers. The only way to modify reports and ensure that the changes are synchronized with the server is to use controllers. Although the report's fields can be accessed directly through the `DataDefinition` property, any changes made will not be committed. This section explains how to add a field to a report.

IDENTIFYING THE FIELD TO ADD

A field is usually selected by name. The `DatabaseController` can be used to retrieve the object that represents this field given a database field's name or its table's name. Another method of accessing a table's fields is using the `ReportClientDocument`'s `Database` property. Here you use the `DatabaseController`.

The `DatabaseController` contains a collection of database tables that are available to the report and might be accessed using the `getDatabaseController` method of ReportClientDocument. Each table contains a collection of `DBField` objects.

NOTE

All tables and fields that are listed by `DatabaseController.getDatabase()` are not retrieved when the report is refreshed; that is, they are available for report design but might not actually be part of the report's data definition.

A method called `findFieldByName` is shown in the following sample code snippet. This method returns a field given its fully qualified field name in the form: `<TableAlias>.<FieldName>`. The table alias is used as a qualifier and it is assumed that a period is used to separate the table alias from the field name.

```
IField findFieldByName(String nameOfFieldToFind, ReportClientDocument
➥reportClientDocument)
{
    //Extracts the field name and the table name.
    int dotPosition = nameOfFieldToFind.indexOf(".");
    String tablePartName = nameOfFieldToFind.substring(0, dotPosition);
    String fieldPartName =
       nameOfFieldToFind.substring(dotPosition + 1,
➥ nameOfFieldToFind.length());

    ITable table = null;

    // Uses the DatabaseController to search for the field.
    try
    {
        Tables retreivedTables =
            reportClientDocument.getDatabaseController().getDatabase().
➥getTables();
        int tableIndex = retreivedTables.findByAlias(tablePartName);
        table = retreivedTables.getTable(tableIndex);
    }
    catch (ReportSDKException e)
    {
        return null;
    }

    // Finds the field in the table.
    int fieldIndex =
        table.getDataFields().find(fieldPartName,
➥FieldDisplayNameType.fieldName, Locale.ENGLISH);
    if (fieldIndex == -1)
    {
        return null;
    }
    IField field = table.getDataFields().getField(fieldIndex);

    return field;
}
```

This method uses the following key methods:

- `Tables.findByAlias` finds the index of a particular table when given its alias. Given the index, the desired `Table` object can be retrieved from the collection.
- `Fields.find` finds the index of a field in a table's `Fields` collection when given the name of the field.

ADDING A FIELD TO THE REPORT DOCUMENT

After you obtain the `Field` object that you want to add, the field can be added to the report so that it is processed and displayed when the report is run. This is done via the `DataDefController`, which is used to modify the report's data definition and contains a subcontroller called the `ResultFieldController`. This subcontroller is used for modifying fields that have been placed on the report and that are processed at runtime. The fields that are shown on the report belong to the `ResultFields` collection. A new database field will be added to the `ResultFields` collection in this step.

NOTE

> The `ResultFields` collection can contain other types of field objects such as parameter fields, formula fields, and summary fields in addition to `DBField` objects. Like `DBFields`, the `ResultFieldController` can add these fields to a report. Unlike `DBFields`, only the `DatabaseDefController`'s `DataDefinition` property, and not the `DatabaseDefController`'s `Database` property, can retrieve these fields.

A field being added to the `ResultFields` collection is shown by the following code:

```
/*  * Because all modifications to a report must be made with a controller,
  * the resulting field controller is used to add and remove each field.
  */
ResultFieldController resultFieldController =
    reportClientDocument.getDataDefController().getResultFieldController();
// Adds fieldToAdd. -1 indicates the end of the collection.
resultFieldController.add(-1, fieldToAdd);
```

The parameter `-1` indicates that the field is to be placed at the end of the collection. As a result of this code, the new field displays on the report and is processed when the report is refreshed.

DETERMINING ALL FIELDS USED IN THE REPORT

The fields that have been added to a report are stored in the `ResultFields` collection and can be retrieved using the following sample method:

```
Fields getUsedDatabaseFields(ReportClientDocument reportClientDocument)
{
    Fields usedFields = new Fields();

    /*
```

```
     * The DataDefinition's ResultFields collection
     * contains all the fields that have been placed
     * on the report and which will be processed
     * when the report is refreshed.
     */
    Fields resultFields = null;
    try
    {
        resultFields = reportClientDocument.getDataDefinition().
➥getResultFields();
    }
    catch (ReportSDKException e)
    {
        return null;
    }

    /*
     * Because the ResultFields collection contains
     * many different kinds of fields, all fields except
     * for database fields are filtered out.
     */
    for (int i = 0; i < resultFields.size() - 1; i++)
    {
        if (resultFields.getField(i).getKind() == FieldKind.DBField)
        {
            // Adds the database field to the collection.
            usedFields.addElement(resultFields.getField(i));
        }
    }

    return usedFields;
}
```

With the full name of the field, you can use a `DatabaseController` to retrieve the `DBField` object.

REMOVING A FIELD FROM THE REPORT

When you've found the field you want to remove, use the `ResultFieldController` to remove it as follows:

```
// Removes fieldToDelete.
resultFieldController.remove(fieldToDelete);
```

In this code, `fieldToDelete` is a `DBField` object. After the field is removed from the result fields using this method, the report ceases to display the field.

CREATING A NEW REPORT

A new report can be created by first creating an empty ReportClientDocument as shown:

```
ReportClientDocument reportClientDocument =
    reportAppFactory.newDocument(Locale.ENGLISH);
```

NOTE

Because the `newDocument` method of `ReportClientDocument` is provided for deployments that use an unmanaged RAS to access report (.rpt) files, it should not be deployed when using a Crystal Enterprise RAS. Instead, when deploying with Crystal Enterprise, the `IReportAppFactory.newDocument` method should be used as in the previous code.

Because the report is not actually created until tables are added, after creating an empty `ReportClientDocument`, details such as the new report's tables and the fields used to link them should be added.

RETRIEVING A REPORT'S TABLES

However, before adding the tables to the new report, the table objects must first be retrieved from the source report. This can be accomplished in two ways: using the `DatabaseController` object and using the `Database` object, both of which are available from the `ReportClientDocument` object. The ensuing code iterates through all the tables in an open report and prints the tables' aliases:

```
Tables tables = reportClientDocument.getDatabase().getTables()
for (int i = 0; i < tables.size(); i++)
{
    ITable table = tables.getTable(i);
    out.println(table.getAlias());
}
```

ADDING TABLES TO THE REPORT

Because controllers are the only objects that can modify the report's object model, a controller must be used to add tables to a report. The following code retrieves the report's `DatabaseController` and adds a table.

```
DatabaseController databaseController;
try
{
    databaseController = reportClientDocument.getDatabaseController();
    databaseController.addTable(sourceTable, new TableLinks());
    databaseController.addTable(targetTable, new TableLinks());
}
catch(ReportSDKException e)
{
    throw new Exception("Error while adding tables.");
}
```

The `addTable` method of the `DatabaseController` adds a table to the report. The `addTable` method takes two parameters:

- The `Table` object you want to add
- A `TableLinks` object that defines how the table being added is linked with other tables

LINKING TABLES

Tables must be linked after they have been added to the report. To link two tables, first create a new `TableLink` object, set the properties of the `TableLink`, and then add the `TableLink` to the report definition.

Linking two tables using an equal join is illustrated by the following code:

```
// Create the new link that will connect the two tables.
TableLink tableLink = new TableLink();

/*
 * Add the source field name and the target field name to the SourceFieldNames
 * and TargetFieldNames collection of the TableLink object.
 */
tableLink.getSourceFieldNames().add(sourceFieldName);
tableLink.getTargetFieldNames().add(targetFieldName);

/*
 * Specify which tables are to be linked by setting table aliases
 * for the TableLink object.
 */
tableLink.setSourceTableAlias(sourceTable.getAlias());
tableLink.setTargetTableAlias(targetTable.getAlias());

// Add the link to the report. Doing so effectively links the two tables.
try
{
    databaseController.addTableLink(tableLink);
}
catch(ReportSDKException e)
{
    throw new TutorialException("Error while linking tables.");
}
```

These newly linked tables can be used as the report's data source. However there have been no visible objects added to the report, so when the report is refreshed, it will be blank.

ADDING GROUPS

To add a group, you must know which field is being grouped on. For information on working with fields, see the "Adding a Field to the Report Document" section earlier in this chapter. Because not all fields can be used to define a group, use the `canGroupOn` method of `GroupController` to check whether a field can be used for grouping. If `canGroupOn` returns true, the field is an acceptable field to use for grouping. The next example demonstrates a function that adds a new group to a report:

```
// Uses the sort controller to remove all the report's sorts.
Sorts sorts = dataDefController.getDataDefinition().getSorts();
SortController sortController = dataDefController.getSortController();
for (int i = 0; i < sorts.size(); i++)
{
    sortController.remove(0);
}
```

Here the group was added to the end of the Groups collection by setting the index to `-1`, which means that the new group becomes the innermost group. When a new group is added, a new sorting definition is also added which will sort the records according to the group's condition field and group options. An additional reflection of adding the new group is the group name field appearing on the group's header. Fields added to the group header are not added to the `ResultFields` collection. When the group is removed, the group name field is also removed.

ADDING SORTING TO THE REPORT

Using the `SortController` adds a new sorting definition to a report. The `SortController` can add any kind of sorting definition, including a Top N sort. Adding a Top N sort requires that a summary has first been added.

Next you demonstrate how to add a sort to the report by taking a `Fields` collection and adding a sorting definition based on each field in the collection:

```
void addNewGroup(ReportClientDocument reportClientDocument,
➥ Fields newGroupFields)
throws ExampleException
{
    try
    {
        // Create a new, empty group.
        IGroup group = new Group();

        // Iterate through every field in the given Fields collection.
        for (int i = 0; i < newGroupFields.size(); i++)
        {
            IField field = newGroupFields.getField(i);

            // Set the field that will define how data is grouped.
            group.setConditionField(field);


            GroupController groupController =
                reportClientDocument.getDataDefController().
➥getGroupController();
            groupController.add(-1, group);
        }
    }

    // If any part of the above procedure failed, redirect the user
➥to an error page.
    catch (ReportSDKException e)
    {
        throw new ExampleException("Error while adding new groups.");
    }
}
```

When the new Sort object is added, it is added to the end of the collection, indicated by the `-1` argument, which designates that the records will be sorted on this field after all other

sorting definitions. The `SortDirection` class indicates the direction of the sort. The static objects `SortDirection.ascendingOrder` and `SortDirection.descendingOrder` are the only values that can be used for a normal sort. The other values are used for a Top N or Bottom N sort. See Adding a Top N sorting definition in the SDK documentation for additional details.

ADDING SUMMARIES TO THE REPORT

The `SummaryFieldController` adds a new summary field. To determine if a field can produce a summary, the `SummaryFieldControllers` method `canSummarizeOn` is called. Here you add a summary to a group:

```
void setSorting(ReportClientDocument reportClientDocument, Fields
➥fieldsToSortOn)
throws ExampleException
{
    try
    {
        DataDefController dataDefController = reportClientDocument.
➥getDataDefController();

        // Create a new Sort object
        ISort sort = new Sort();

        // Iterate through the fields
        for (int i = 0; i < fieldsToSortOn.size(); i++)
        {
            IField field = fieldsToSortOn.getField(i);

            // Add the current field to the result fields.
            dataDefController.getResultFieldController().add(-1, field);

            // Set the field to sort on.
            sort.setSortField(field);

            // Define the type of sorting. Ascending here.
            sort.setDirection(SortDirection.ascendingOrder);

            //Get Sort Controller.
            SortController sortController =
                dataDefController.getSortController();

            sortController.add(-1, sort);
        }
    }

    // If any part of the above procedure failed, redirect the user to
➥an error page.
    catch (ReportSDKException e)
    {
        throw new TutorialException("Error while setting sort.");
    }
}
```

After creating a summary field, set the following properties before adding it:

- `SummarizedField`—The field used to calculate the summary.
- `Group`—The group for which the summary will be calculated.
- `Operation`—The operation used to calculate the summary. One of the static objects defined in the `SummaryOperation` class.

WORKING WITH FILTERS

Filters are used in record selection and group selection. The filter is initially a string written in Crystal formula syntax. The record selection formula is then parsed into an array of `FilterItems` stored in the Filter object's `FilterItems` property. The string is broken up into data components and operator components that act on the data. These components are stored as `FieldRangeFilterItem` and `OperatorFilterItem` objects respectively, which are stored in the `FilterItems` collection in the same order that they appear in the formula string. Re-ordering the objects in the array changes the functionality of the formula. In summary, the `FieldRangeFilterItem` is an expression that is joined with other expressions using an `OperatorFilterItem`.

For instance, consider a simple record selection formula such as

```
{Customer.Name} = "Bashka Futbol" and {Customer.Country} = "USA".
```

This results in only the records that have a name equal to `"Bashka Futbol"` and a country of the `USA`. The result is stored in the `FreeEditingText` property. After this string is parsed, the `FieldRangeItems` collection contains two `FieldRangeFilterItem` objects because there are two data items used to filter the records. The `OperatorFilterItem` is used to indicate how two primitive expressions are combined, so it is now equal to `and`.

The `FieldRangeFilterItem` contains three properties:

- `Operation`—This property indicates the operation performed in the primitive expression; in both cases, it is the equals operator.
- `RangeField`—The `RangeField` property indicates the comparator field used in the expression. Because not all fields are suitable to filter records and groups, use the `canFilterOn` method in the `RecordFilterController` and the `GroupFilterController` to determine whether a field can be used for a particular filter.
- `Values`—The `Values` property indicates the comparison values in the expression. In this example, it is the strings "USA" and "Bashka Futbol". This property has one `ConstantValue` object that stores "USA".

After the file is opened, and the filters parsed, the `FreeEditingText` property that stores these strings is cleared and the `FilterItems` populated. Conversely, if the formula is too complex, the `FilterItems` collection property remains empty and the `FreeEditingText` property populated. When altering a filter, you have two options: modify the `FreeEditingText` property or the `FilterItems` property.

If you use only one property to modify the filter, the other will not be automatically updated, however. For instance, you would modify the `FreeEditingText` property, but this will not necessarily be parsed again to repopulate the `FilterItems`. You should use only one of these properties per session.

Use a controller to ensure that modifications are saved. The `GroupFilterController` and the `RecordFilterController` modify the group formula and record formula respectively.

CREATING A `FieldRangeFilterItem`

A `FieldRangeFilterItem` contains a primitive comparison expression. Its most relevant properties are

- `Operation`
- `RangeField`
- `Values`

The `Operation` and `RangeField` properties usually contain a constant. However, the `Values` property stores either `ConstantValue` objects, which don't need evaluation (such as 1, 5, or "Stringiethingie"), and `ExpressionValue` objects, which do need evaluation (such as "WeekToDateSinceSun," 4/2, and so on).

The following section of code defines the expression `{Customer.ID > 2}`. Note how it creates a new `ConstantValue` object for the number `2` and adds it to the `Values` collection:

```
//  Create a new range filter item.
FieldRangeFilterItem fieldRangeFilterItem = new FieldRangeFilterItem();

// Assume the customerDBField has been retrieved from a table
fieldRangeFilterItem.setRangeField(customerDBField);

// Set the operation to >
fieldRangeFilterItem.setOperation(SelectionOperation.greaterThan);
fieldRangeFilterItem.setInclusive(false);

// Create a constant value and add it to the range filter item
ConstantValue constantValue = new ConstantValue();
constantValue.setValue(2);
fieldRangeFilterItem.getValues().addElement(constantValue);

//  Create a filter and add the field range filter item
IFilter filter = new Filter();
filter.getFilterItems().addElement(fieldRangeFilterItem);
```

All fields cannot be used in a filter formula (for example, you can't use BLOB fields). Use the `canFilterOn` method, which is located in either the `RecordFilterController` or the `GroupFilterController`, to verify that a field can be filtered on. You must also verify that the constant data type is the same as the field. In the previous example, `constantValue` must not be a variant and corresponds to the data type used in the comparison.

CREATING A OperatorFilterItem

The following example assumes the same expression as defined in the preceding example, but concatenates to the filter using the OR operator. Assume the filter would look like this: {Customer.ID} > 2 OR {Customer.name} = "Arsel". To the code above you would add:

```
OperatorFilterItem operatorFilterItem = new OperatorFilterItem();
operatorFilterItem.setOperator("OR");

filter.getFilterItems().addElement(operatorFilterItem);
filter.getFilterItems().addElement(fieldRangeFilterItem);
```

The filterItems parameter is a FilterItems collection. It stores FilterItem objects. In the two examples, both a FieldRangeFilterItem object and an OperatorFilterItem object were added to this collection. Both of these objects inherit from FilterItem, making this possible.

ADDING A FILTER TO THE REPORT

After defining the filter, you add it to the report. Filters can be used in two places: group selection and record selection. The GroupFilterController and RecordFilterController, which can be accessed via the DataDefController object, modify their respective filters.

You can also obtain the filters from the GroupFilter and RecordFilter properties in the DataDefinition, although they can only be modified with a controller.

FilterController provides these methods for modifying a filter:

- addItem—This method adds an expression or an operator to the existing filter.
- modify—This method replaces the current filter with a new or modified one.
- modifyItem—This method modifies a filter element.
- moveItem—This method moves the filter element around the filter array.
- removeItem—This method deletes a filter element.

In the following code, the modify method is used because a new filter has already been defined. Assume that there is a ReportClientDocument object and that you have opened a report already:

```
FilterController groupFilterController =
reportClientDocument.getDataDefController.getGroupFilterController();
groupFilterController.modify(filter) ;
```

WORKING WITH PARAMETERS

Parameters enable end users to enter information to define the report behavior. Parameters have specific data types just like any other field: string, number, date, and so on. Parameters also are divided into two basic types: discrete and ranged. A discrete parameter value is one that represents a singular value such as 9, "Nur", 1863, True, and so on. Ranged values represent a particular span of values from one point to another such as [9..95], [4..6], ["Alpha","Omega"]. The lower bound value of the range must be smaller than the upper

bound. Some parameters support more than one value: They effectively contain an array containing many values.

Parameters have default values and the user can be forced to select from them. You can also provide default parameters but allow users to enter their own values. Default values are stored in the `ParameterField.DefaultValues` property. Selected values are stored in the `ParameterField.CurrentValues` property.

Parameters support many more features than those covered here. For a complete list of features, see the `ParameterField` class in the SDK documentation.

READING PARAMETERS AND THEIR VALUES

The parameters are exposed in the SDK by the `DataDefinition`'s `ParameterFields` class. The `ParameterFields` class inherits from the `Fields` class. For example the name of a parameter is obtained using this method:

```
Fields parameterFields = reportClientDocument.getDataDefinition().
➥getParameterFields();
ParameterField parameterField = (ParameterField)parameterFields.getField(0);
parameterField.getDisplayName(FieldDisplayNameType.fieldName, Locale.ENGLISH);
```

The `getDisplayName` method is used for UI purposes and so is not a unique identifier. The `getFormulaForm` method can be used to retrieve a unique identifier. `getDisplayName` and `getFormulaForm` are not documented under the `ParameterField` class because they are inherited from `Field`.

Because parameter values might be either discrete or ranged, and default values might only be discrete, there are two different objects to represent these: `ParameterFieldDiscreteValue` and `ParameterFieldRangeValue`. Both of these objects inherit from `ParameterField`. You must understand the type of the parameter to know what kind of parameter values it contains. For example, the following code determines if the parameter is of a ranged or discrete type:

```
// Check to see if the value is range or discrete
IValue firstValue = parameterField.getCurrentValues().getValue(0);
if (firstValue instanceof IParameterFieldRangeValue)
{
    IParameterFieldRangeValue rangeValue =
➥ (IParameterFieldRangeValue)firstValue;
    toValueText = rangeValue.getEndValue().toString();
    fromValueText = rangeValue.getEndValue().toString();
}
else
{
    IParameterFieldDiscreteValue discreteValue =
➥ (IParameterFieldDiscreteValue)firstValue;
    discreteValueText = discreteValue.getValue().toString();
}
```

Check the parameter's type before you try to print the parameter's values. You must determine the type so you can retrieve the correct field. Trying to access the `EndValue` of a discrete value will cause a runtime error because no `EndValue` exists. The previous code

example determines whether the parameter value is an instance of `IParameterFieldRangeValue` to determine what kind of values it will have. For parameters that support both discrete and ranged values, however, you must verify the type of the parameter by using `getValueRangeKind` method. The following code checks the parameter type and calls a secondary function to handle the correct type and build a table of parameters:

```
ParameterValueRangeKind kinda = parameterField.getValueRangeKind();
if (kinda == ParameterValueRangeKind.discrete)
{
    table += createDiscreteParameterTableData(parameterField, key);
    key += 1;
}
else if (kinda == ParameterValueRangeKind.range)
{
    table += createRangeParameterTableData(parameterField, key);
    key += 2;
}
    else if (kinda ==  ParameterValueRangeKind.discreteAndRange)
{
    table += createDiscreteRangeParameterTableData(parameterField, key);
    key += 3;
}
else
{
    table += "<td>Parameter kind not known</td>";
}
```

CHANGING PARAMETER VALUES

The `ParameterFieldController`, which can be found in the `DataDefController`, enables you to change parameters. To modify a parameter field in the report, you copy the field, modify the copy, and then have the controller modify the original based on changes made to the copy. For instance here you demonstrate this by changing a default discrete value:


```
ParameterField newParamField = new ParameterField();
parameterField.copyTo(newParamField, true);
newParamField.getCurrentValues().removeAllElements();

// Check the type of the parameter
ParameterValueRangeKind kinda = parameterField.getValueRangeKind();

// If it is discrete
if (kinda == ParameterValueRangeKind.discrete)
{
    // Get the parameter's value
    String textFieldText = request.getParameter("textField" + key);

    // Convert this value to the right format
    String discreteValueText =
(String)convertToValidValue(newParamField, textFieldText);

    // Modify the copy of the parameter field with the value above.
    ParameterFieldDiscreteValue discreteValue =
➥new ParameterFieldDiscreteValue();
    discreteValue.setValue(discreteValueText);
```

```
    newParamField.getCurrentValues().add(discreteValue);

    key += 1;
}
```

ADDING A PARAMETER

Use the `ParameterFieldController` to add new parameters. You do this the same way as adding any other fields to the report: A new field is created, its fields are set, and it is added using a controller. Here you define a new, discrete, string parameter and add it using the controller:

```
IParameterField paramField = new ParameterField();

paramField.setAllowCustomCurrentValues(false);
paramField.setAllowMultiValue(false);
paramField.setAllowNullValue(false);
paramField.setDescription("Here we go dude!");
paramField.setParameterType(ParameterFieldType.queryParameter);
paramField.setValueRangeKind(ParameterValueRangeKind.discrete);
paramField.setType(FieldValueType.numberField);
paramField.setName("YourNewParameter");

reportClientDoc.getDataDefController().getParameterFieldController().
➥add(parameterField);
```

Adding a parameter using the Parameter field controller does not place the parameter on the report, so the user is not prompted for the parameter when the report is refreshed. To prompt the user, either use it in a filter, or add it by using the `ResultFieldController`.

TIPS AND TRICKS FOR PARAMETER HANDLING

Handling parameters involves many important details. When using parameters keep the following points in mind:

- Parameter values must match the type of the parameter.
- Any values for the parameter should respect the parameter mask.
- Ensure that you know what type of values you are reading: Are they discrete or ranged?
- Set the bound type on a range value before adding it to the parameter.
- Ensure that the upper bound of a range value is greater than the lower bound.

Failing these tests results in a runtime error.

CHARTING OVERVIEW

The `ChartObject`, which represents a chart in the RAS SDK, inherits variables and methods from the `ReportObject`. Remember that the report that you open is represented by the `ReportClientDocument`, not the `ReportObject`.

The `ChartObject`'s properties determine the chart's appearance and where it shows on the report.

Here you focus on three `ChartObject` properties:

- `ChartDefinition` indicates the chart type and the fields charted. The chart type can be a Group or Details type.
- `ChartStyle` specifies the chart style type (such as a bar chart or a pie chart) and the text that appears on the chart (such as the chart title).
- `ChartReportArea` is where the chart is located (for example, the report footer).

The following two sections show how you can use these `ChartObject` properties to create a chart. You must first specify the fields on which you want your chart to be based on. To do this, create a `ChartDefinition` object, which will then be added to the `ChartObject` with the `ChartDefinition` property.

DEFINING THE FIELDS IN A CHART

The `ChartDefinition` object determines the type of chart that appears in the report and sets the fields to be displayed. A simple, two-dimensional chart displays two types of fields:

- `ConditionFields`—The fields that determine where to plot a point on the x-axis.
- `DataFields`—The fields that determine what to plot on the y-axis.

Below the chart added is a Group type (see the `ChartType` class), so the `ConditionFields` and `DataFields` that are being charted on are group fields and summary fields respectively.

ADDING `ConditionFields`

Add the first group field in the `Groups` collection to a `Fields` collection. This field is retrieved with the `ChartDefinition`'s `getConditionFields` method.


```
ReportClientDocument's DataDefinition:
// Create a new ChartDefinition and set its type to Group
ChartDefinition chartDef = new ChartDefinition();
chartDef.setChartType(ChartType.group);

Fields conditionFields = new Fields();
if (!dataDefinition.getGroups().isEmpty())
{
    IField field = dataDefinition.getGroups().getGroup(0).getConditionField();
    conditionFields.addElement(field);
}
chartDef.setConditionFields(conditionFields);
```

NOTE

Adding two groups as `ConditionFields` enables you to create a 3D chart. Because one value is required for the x values, the next value drives the z-axis.

ADDING DataFields

After you have added ConditionFields to the ChartDefinition, add the DataFields. In a Group type chart, the DataFields are summaries for the group fields that you added as ConditionFields.

Adding DataFields is similar to how you added ConditionFields. For example, you use the name of the summary field that the user has selected to locate the desired field in the SummaryFields collection and add this field to a Fields collection. You then accessed the summary field with the ChartDefinition's DataFields property:

```
Fields dataFields = new Fields();
for (int i = 0; i < dataDefinition.getSummaryFields().size(); i++)
{
    IField summaryField = dataDefinition.getSummaryFields().getField(i);
    if (summaryField.getLongName(Locale.ENGLISH).equals(summaryFieldName))
    {
        dataFields.addElement(summaryField);
    }
}
chartDef.setDataFields(dataFields);
```

Here you use the LongName of the summary field. The LongName contains the type of summary, for example, a sum or a count, and the group field that it applies to. For example

```
Sum of (Customer.Last Year's Sales, Customer.Country)
```

In general you will want to use a field's LongName instead of its ShortName or Name to avoid confusion as the ShortName or Name might be the same for several fields.

CREATING A ChartObject

After the fields are defined, they are added to the ChartObject with the ChartDefinition property. The following code uses the ChartObject's ChartStyle property and ChartReportArea to specify the chart style type, the chart title, and the location of the chart:

```
ChartObject chartObject = new ChartObject();
chartObject.setChartDefinition(chartDefinition);

String chartTypeString = request.getParameter("type");
String chartPlacementString = request.getParameter("placement");
String chartTitle = request.getParameter("title");
if (chartTitle.equals(""))
{
    chartTitle = "no title at all!";
}

ChartStyleType chartStyleType = ChartStyleType.from_string(chartTypeString);
AreaSectionKind chartPlacement =
➥AreaSectionKind.from_string(chartPlacementString);

// Set the chart type, chart placement, and chart title
chartObject.getChartStyle().setType(chartStyleType);
chartObject.setChartReportArea(chartPlacement);
chartObject.getChartStyle().getTextOptions().setTitle(chartTitle);
```

```
// Set the width, height, and top
chartObject.setHeight(5000);
chartObject.setWidth(5000);
chartObject.setTop(1000);
```

In this example, the first chart that you add will appear 50 points below the top of the report area in which the chart is located. (These fields are measured in twips, and 20 twips = 1 font point, so 1000/20 = 50 points.) Adding another chart to the same report area places it over the first chart because the formatting for report objects is absolute. The first chart remains hidden until the second chart is removed.

ADDING A CHART TO THE REPORT

Now add the chart using the `ReportObjectController`'s `add` method. This method takes three parameters: the `ChartObject`, the section to place it in, and the position in the `ReportObjectController` collection where you want to add the chart. An option of `1` for the index adds the chart to the end of the array. Return the `ReportObjectController` by the `ReportDefController`'s `getReportObjectController` method:

```
reportDefController.getReportObjectController().add(chartObject,
➥chartSection, 1);
```

NOTE

If you want to modify an existing chart, you can use the clone method to copy the chart, make the desired changes, and then call the `modifyObject` method using the original chart and the newly modified chart as parameters.