WebObjects Developer's Guide

This manual describes WebObjects 4.0.

Writing: Greg Wilson and Jean Ostrem
With help from Eric Bailey, Craig Federighi, Patrice Gautier,
Francois Jouaux, and Charles Lloyd
Design and Illustration: Karin Stroud
Production: Gerri Gray

Contents

Table of Contents

Part II Special Tasks

Creating Reusable Components 133

Managing State 157

WebScript Programmer's Quick Reference to Foundation Classes 233

About This Book

This book describes concepts that you'll need to know when writing a WebObjects application. Programmers of all skill levels will find all or part of the information in this book useful. To help you find what you are looking for, this book is organized into three parts:

- Part 1, "WebObjects Essentials," is for programmers who are new to WebObjects.

  Part 1 covers basic concepts that are required knowledge for even the simplest of WebObjects programs. It describes what a WebObjects application is and what pieces a WebObjects application contains. It describes how to debug applications and provides a checklist of hard-to-find errors. The final chapter in Part 1, "WebObjects Viewed Through Its Classes," provides an in-depth description of the classes used in all WebObjects applications and explains how those classes process HTTP requests.

- Part 2, "Special Tasks," is for intermediate-to-advanced WebObjects programmers.

  Part 2 provides information that you can ignore until you understand the basic WebObjects concepts explained in Part 1. It describes how you can design components for reuse inside of other components, how a WebObjects application manages and stores state, how to create client-side Java components that behave like dynamic elements, and how to design an application for deployment and improved performance.

- Part 3, "WebScript," is for programmers who want to use a scripting language.

  WebScript is a scripting language provided with WebObjects for rapid application development. Use of WebScript is entirely voluntary—you can write applications using Java or Objective-C if you prefer. Part 3 describes WebScript's syntax and also describes how to use the Foundation framework when writing an application using WebScript.

There are no prerequisites for learning WebObjects; however, it does help if you understand object-oriented programming concepts and are familiar with either Java or Objective-C. If you aren't familiar with Java or Objective-C, you might want to pay special attention to Part 3 of this book. Part 3 provides a very brief introduction to object-oriented concepts—enough to get you started. Later, you'll want to round out your knowledge either by reading the book *Object-Oriented Programming and the Objective-C Language* or any other book on object-oriented programming.

Part I

WebObjects Essentials

Chapter 1

What Is a WebObjects Application?

What is WebObjects?

From a client perspective, WebObjects is a scalable, high-availability, high performance application server. From the viewpoint of a developer, though, WebObjects is an open platform upon which you can rapidly develop and deploy web applications which integrate existing data and systems. WebObjects is especially suited to two kinds of applications: those that do dynamic publishing, and those traditional client-server applications that will benefit from the increased connectivity that the World Wide Web provides.

The web was created to simplify access to electronically-published documents. Originally just static text pages with hyperlinks to other documents, web pages quickly evolved into highly-graphical animated presentations. Along the way, a degree of interactivity was introduced, allowing people browsing the web to fill out forms and thereby supply data to the server.

Dynamic HTML Publishing

The vast majority of web content falls into the category of web publishing. Much of it is still of the static variety: sites that contain textual or graphical material that doesn't change much over time. However, there is increasing demand for sites that publish ever-changing data: breaking news stories, up-to-the-minute stock quotes, or the current weather are good examples.

A typical web site looks something like that illustrated in Figure 1. Client requests, conforming to the HTTP protocol, originate at the user's web browser. These requests are sent over the network to the web server, which analyzes the request and selects the appropriate web page to return to the client browser. This web page is simply a text file that contains HTML markup. Using the HTML commands embedded within the file received from the web server, the browser renders the page.

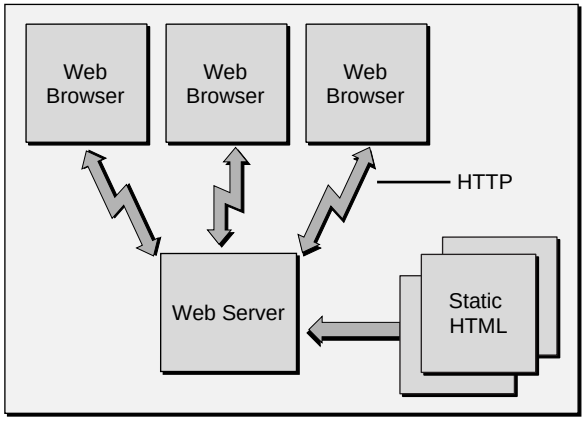

Figure 1. A Static Publishing Site

Static publishing sites are easy to maintain. There are a number of tools on the market that allow you to quickly and easily create HTML pages, and as long as the page content doesn't change too often, it isn't that difficult to keep them up-to-date. Dynamic publishing sites, however, are a different story altogether. It would take a small army to keep a breaking news site up to date, for instance.

WebObjects was designed from the beginning to allow you to quickly and easily publish dynamic data over the web. You create *templates* that indicate where on the web page the dynamic data is to be placed, and a simple WebObjects application—which in some cases can be completely generated using Wizards provided as part of WebObjects—fills in the content when your application is accessed. The information that your web pages publish can reside in a database, or it can reside elsewhere (in files, perhaps), or it can even be calculated or generated at the time a page is accessed.

Figure 2 shows a WebObjects-based dynamic publishing site. Again, the HTTP request (in the form of a URL) originates with a client browser. If the web server detects that the request is to be handled by WebObjects, it passes that request off to a WebObjects adaptor. The adaptor packages the incoming HTTP request in a form the WebObjects application can understand and forwards it to the application. Based upon templates you've defined, and the relevant data from the data store, the application

generates an HTML page which it passes back through the adaptor to the web server. The web server sends the page to the client browser, which renders it.

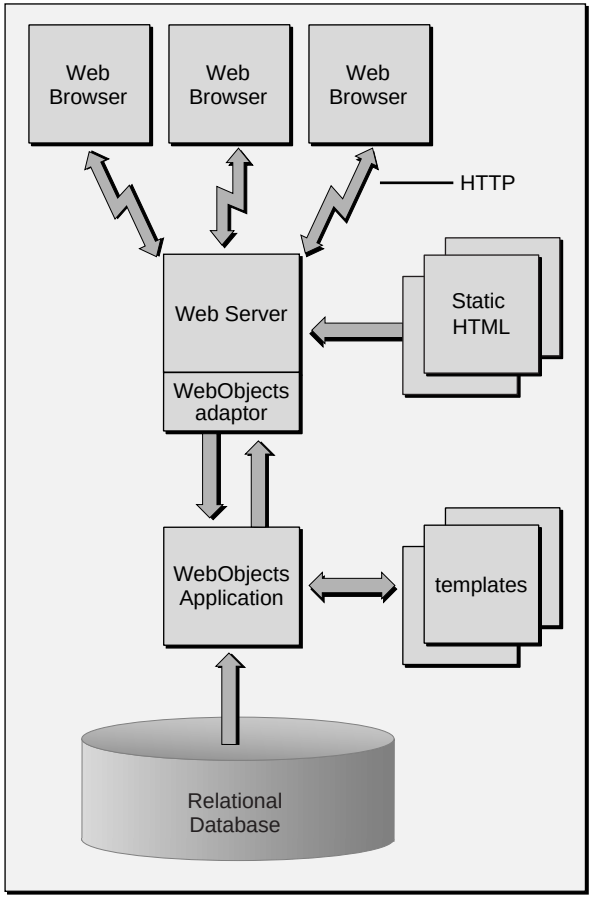

Figure 2. A Dynamic Publishing Site

Web-Enabled Client-Server Applications

Although the majority of web sites primarily publish data, the number of web-enabled applications is growing rapidly. Many corporations use intranets, the internet, or both to provide easy access to internal applications and data. An “online store” selling books, music, or even computers is one example of a web-enabled client-server application.

Web-enabled applications can have huge advantages over traditional apps. For one, clients don't have to install the application. This not only saves client disk space but ensures that the user always has the most up-to-date version of the application. As well, the client computers can be a heterogeneous mix. Macintosh computers, PCs, workstations; anything that can run a web browser with the necessary capabilities.

WebObjects' development tools allow you to quickly and easily create web applications. WebObjects supplies a large number of pre-built *components*—web pages, or portions of web pages, from which you can build up your web application's interface. These components range from simple UI widgets (such as drop-down list boxes, tables, and the like) to complex ones that, for instance, present a tool bar. And the set of components that you can use with WebObjects is extensible, so you can create components that can then be re-used across all of your web applications.

Your application isn't entirely built of components, however. You create WebObjects applications from a combination of components and classes, some of which you write, some of which are provided by WebObjects. The classes you write can either be written using a compiled language (Java or Objective-C) or using WebObjects' own interpreted language, WebScript.

Accessing Legacy Data

Much of the data that is (or could be) presented on the web already exists in electronic form. Not only can it be a challenge to create a web site or web app to present your data using conventional tools, it can also be a challenge just to access the data itself. WebObjects works hand-in-hand with set of frameworks designed to allow your objects access to data stored in relational databases and other legacy data stores.
These frameworks, collectively known as Enterprise Objects Framework, maps database contents into objects using a model you create. Freed from having to know SQL or any of the specific techniques involved in accessing data from your particular third-party databases, you can simply work with the data as objects, letting the Enterprise Objects Framework keep your objects in sync with the underlying data stores.

State Management

The HTTP protocol used on the web is inherently stateless. Most applications of consequence—as well as some of the more interesting dynamic publishing sites—need to keep state information associated with each user session, such as login information or a shopping basket.

WebObjects provides objects that allow you to maintain information for the life of a particular client session, or longer. This makes it particularly easy to implement an application like a web-based online store; you don't have to do anything special in order to maintain the contents of the user's shopping cart over the life of the session, for instance. And your online store could even monitor individual customer buying patterns and then highlight items they're more likely to be interested in the next time they visit your site.

Your applications can, of course, use more traditional means to keep track of state information—URLs, cookies, hidden fields, and so on—if you prefer.

Performance

Static web sites and traditional client-server applications have one strong suit: they both leverage the power of the client platform, minimizing the load on the server. It doesn't take all that much processing power to serve up a set of static web pages, for instance. Dynamic web applications, although a tremendous advance over static pages, requires additional server power to access the dynamic data and construct the web pages "on the fly."

The WebObjects application server is both efficient and scalable. With WebObjects, if more power, reliability, or failover protection is needed you can run multiple instances of your application, either on one or on multiple application servers. You can specify how WebObjects decides which instance each new client should be connected to. And, either locally or from a remote location, you can analyze site loads and usage patterns and then start or stop additional application instances as necessary. Load balancing is a very powerful feature of WebObjects that allows you to simply add more server capacity as the need arises.

Java Client

For some applications HTML is simply too restrictive for efficient client-server communications. Although dynamically-generated HTML pages can make for effective displays, using HTML forms for data entry into your application can be awkward at times. Because of this, WebObjects' Java Client capabilities allow you to partition your application so that a portion of it—including all or part of the user interface—runs in Java directly on the client. Client-server communication is handled by WebObjects.

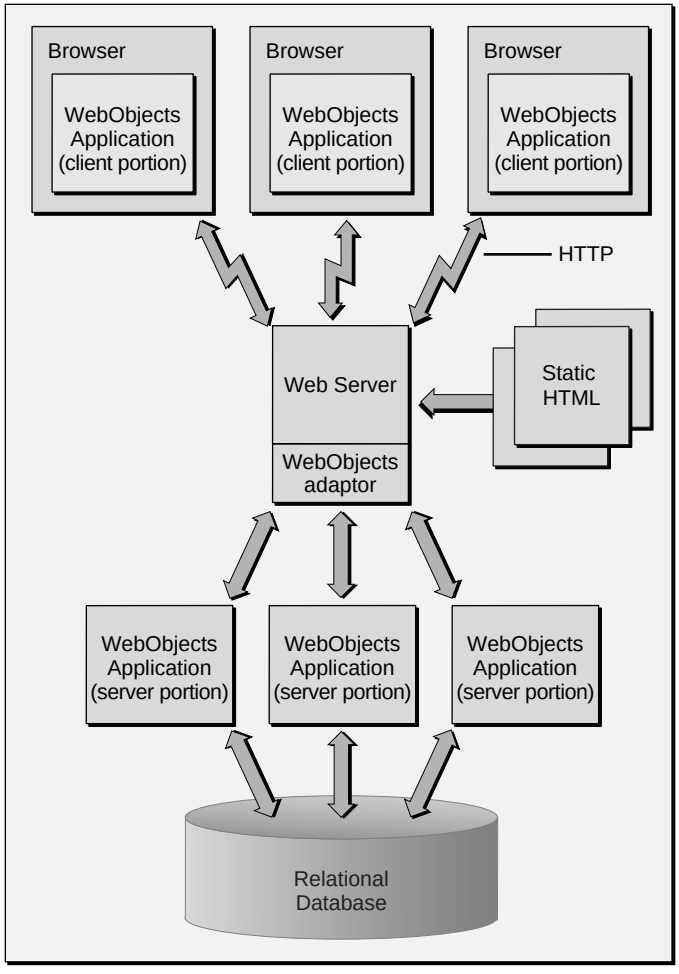

Figure 3. A Web Site Running Java Client Applications

Figure 3 illustrates a Java Client application. As before, the browser can still communicate with your application using HTTP and HTML. In addition, Java Client passes objects between a portion of your application (written in Java) residing within the browser and the portion of your application that remains on the application server. Your WebObjects applications can therefore be a mix of Java Client and HTML-based pages.

How WebObjects Applications Work

WebObjects applications come in two flavors, depending on how much processing you plan to do on the client's computer. Server-based WebObjects applications run entirely on the server, handling HTTP requests and generating HTML pages. Applications that are based on WebObjects' Java Client technology pass objects directly between the server and the client. Although you can construct hybrid applications that use both technologies, for purposes of explanation it's simpler to keep the two technologies separate.

Server-Based WebObjects Applications

When you run a WebObjects application, it communicates with the web browser through the chain of processes shown in Figure 4.

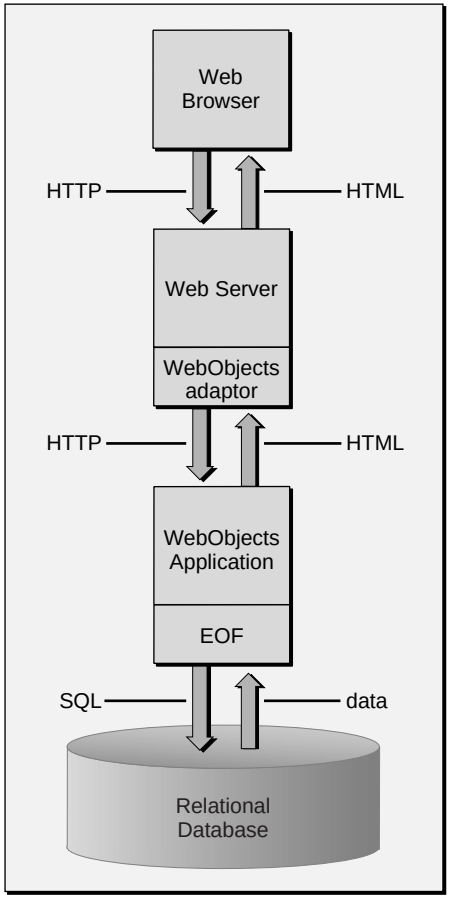

Figure 4. Chain of Communication Between the Browser and an HTML-based WebObjects Application

Here is a brief description of these processes:

- **A Web server**. Any HTTP server that uses the Common Gateway Interface (CGI), the Netscape Server API (NSAPI), the Internet Server API (ISAPI), or the Apache API. Although (usually) necessary for deployment, you don't actually need a web server while you develop your WebObjects applications.

- **A WebObjects adaptor**. A WebObjects adaptor connects WebObjects applications to the web by acting as an intermediary between web applications and HTTP servers.

- **A WebObjects application process**. The application process receives incoming requests and responds to them, usually by returning a dynamically-generated HTML page. You can run multiple instances of this process if one instance is insufficient to handle the application load.

Two of these, WebObjects adaptors and the WebObjects application process, are described in greater detail beginning on page 29.

Java Client-Based WebObjects Applications

Figure 5 shows a client browser communicating with a WebObjects server while running a Java Client application. The portion of your application that runs in the browser is linked to an EOJavaClient object. The portion that runs on the server is similarly linked to an EODistribution object. These two objects handle communications between the client and the server, allowing both portions of your application to focus on implementing your application's business logic.

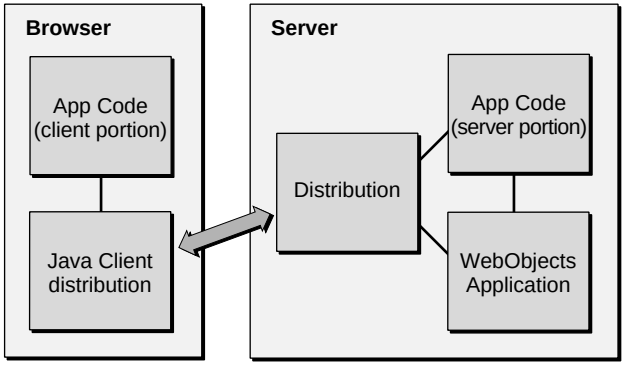

Figure 5. Chain of Communication Between the Browser and a Java Client-based WebObjects Application

Note: The techniques used in creating the Java Client portions of your WebObjects applications are documented in greater detail in the *Enterprise Objects Framework Developer's Guide*. The remainder of this

book deals with those aspects of WebObjects that are common to all WebObjects applications.

WebObjects Adaptors

An initial request to a WebObjects application consists of a URL like that shown in Figure 6.

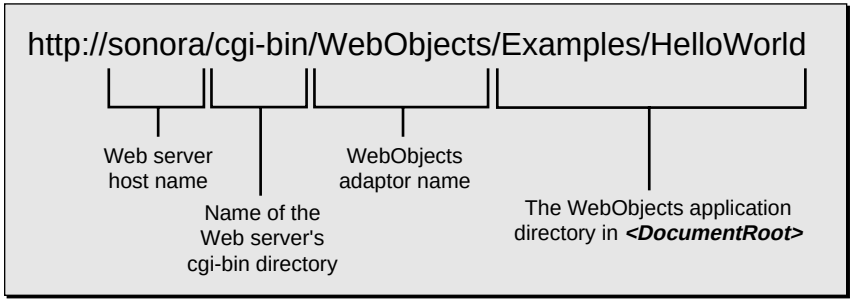

Figure 6. URL to Access a WebObjects Application

When the named server receives the request, it passes the request to the WebObjects adaptor identified in the URL. The adaptor takes the request from the server, repackages it in a standard format, and forwards it to the appropriate WebObjects application (see Figure 7).

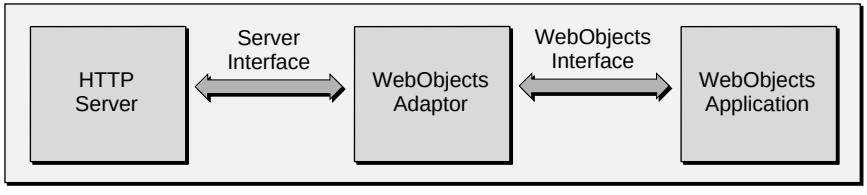

Figure 7. The Role of a WebObjects Adaptor

All WebObjects adaptors communicate with WebObjects applications in the same way, but they communicate with HTTP servers using whatever interface is provided by a particular server. For example, the WebObjects CGI adaptor uses the Common Gateway Interface, while the Netscape Interface adaptor uses the Netscape Server API (NSAPI). This allows WebObjects adaptors to take advantage of server-specific interfaces but still provide server independence.

By default, WebObjects uses the WebObjects CGI adaptor. The Common Gateway Interface is supported by all HTTP servers, so you can use the CGI adaptor with any server. As demands on performance increase, you can switch to another adaptor with a server that supports the corresponding API (Netscape Server API, Internet Server API, or Apache API). Such servers are capable of dynamically loading the adaptor and eliminating the overhead of starting a new process for each request.

The Request-Response Loop

WebObjects applications are event driven, but instead of responding to mouse and keyboard events, they respond to HTTP requests. The main loop of a WebObjects application is called the *request-response loop*. During each cycle of the request-response loop, the application receives an HTTP request for an action, responds to it, and then waits for the next request. The application continues to respond to requests until it terminates.

The basic request-response loop is illustrated in Figure 8, below.

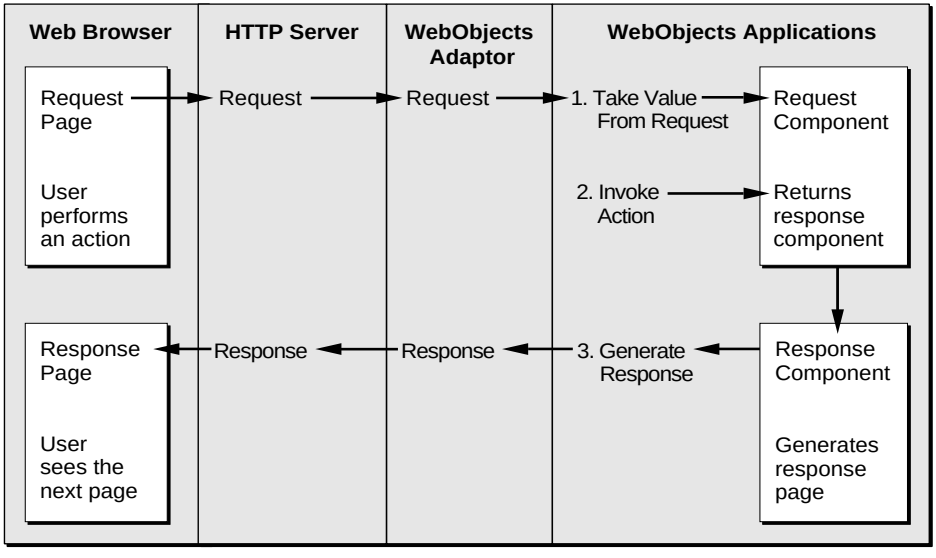

Figure 8. The Request-Response Loop

WebObjects Applications

Figure 8 shows that WebObjects applications are made up of components. Components consist of HTML templates, of user-written scripted or compiled class files, and of *bindings*, which connect the dynamic portions of your components to your action methods and to your business objects.

These components are combined with the WebObjects framework to produce an application executable. When executed, the application process receives incoming requests from the adaptor and responds to them, usually by returning a dynamically-generated HTML page.

The Parts of a WebObjects Application

Project Builder—the development tool you use to manage the various parts of your WebObjects application—groups the various parts of a WebObjects application into "suitcases" by type. Five that are particularly interesting to the WebObjects developer are Classes, WebComponents, Resources, Web Server Resources, and Frameworks.

Classes

The "Classes" suitcase contains the various compiled or scripted custom classes for the objects that make up your application. As well, Classes contains an Application class and a Session class.

The Application object is responsible for receiving requests from the adaptor and forwarding them on to a *request handler*—a dispatcher that passes requests to the appropriate session and component and receives responses back. When a response comes back from the request handler, the Application object passes the response back to the adaptor. The Application object also manages adaptors, sessions, application resources, and components.

Sessions are periods during which a particular user is accessing your application. Because users on different clients may be accessing your application at the same time, a single application may host more than one session at a time. Session objects encapsulate the state of a single session.

These objects persist between the cycles of the request-response loop, and store (and restore) the pages of a session, the values of session variables, and any other state that components need to persist throughout a session. As well, each session has its own copy of the components that its user has requested.

If your application makes use of the Enterprise Objects Framework, your application's enterprise object classes appear in the Classes suitcase. The Enterprise Objects Framework keeps the state of these objects in sync with your external data store (typically, a relational database), freeing you to concentrate on writing your business logic.

Many of the remaining classes that fall into Project Builder's Classes suitcase correspond to your Web Components; these contain the code that defines each component's behavior.

With WebObjects, you can write your code in one of three programming languages: Java (*myClass*.**java**), Objective-C (*myClass*.**m**), and WebScript (*myClass*.**wos**). Because Java and Objective-C require compilation, they aren't as well suited to rapid prototyping as an interpreted language. For this reason WebObjects provides a scripting language named WebScript; it is described in the chapter "The WebScript Language" (page 201).

You can mix and match languages. It's common to use WebScript to write your presentation logic and use Java or Objective-C to write your business logic. Many simple applications are written entirely in WebScript. Some programmers prototype using WebScript and then create a compiled version of the same application to improve performance.

Note: Framework class names are the same regardless of which language you use, but method names sometimes differ between Java and WebScript. (Objective-C uses the same method names as WebScript.) This book generally uses the WebScript names for methods. Usually, you can discern the Java name from the WebScript name, and vice versa. The following table tells you how to do so.

Where the mapping is not obvious, this book notes both the Java and WebScript/Objective-C names.

	WebScript/Objective-C Name	Java Name
Methods with no arguments	*method*	*method*
Single-argument methods	*method***:**	*method*
Multiple-argument methods	*methodWithArg1:arg2:*	*methodWithArg1*

Web Components

A *component* is a web page, or a portion of one, that has both content and behavior. Usually a component represents an entire page, so the word "page" is used interchangeably with the word "component." Components don't always represent an entire page, however. For example, a component might represent only a header or footer of a page; you can nest it inside of a component that represents the rest of the page.

Components are made up of:

- A template that specifies how the component looks
- Code that specifies how the component acts
- Bindings that associate the component's template with its code

Typically, components consist of some form of these three files, but any given component might contain more or fewer parts. For example, a component may not need a code file at all; it may need only a template file and a set of bindings. Another component might have a code file but no template file or bindings. Plus, if you create a component using Project Builder or WebObjects Builder, you'll get a fourth file, *Component***.api**, which contains API that should be made public to other components.

Note that the various parts of a component are actually located in various Project Builder suitcases: the template and bindings files are located under WebComponents, the code files are located under Classes, and API files (if you have any) wind up under a suitcase named Resources.

Template

You use a *template* to specify how the page you're creating should look. This file typically contains static HTML elements (such as `<H1>` or `<P>`) along with some *dynamic elements*. Dynamic elements are the basic building blocks of a WebObjects application. They link an application's behavior with the HTML page shown in the web browser, and their contents are defined at run-time. Some of the more commonly-used dynamic elements are listed in the following table:

Element Name	Description
WOActionURL	Enables the creation of URLs to invoke methods or specify pages to return.
WOApplet	Generates HTML to specify a Java applet.
WOBody	Specifies the background image to display for the HTML page.
WOBrowser	A selection list that displays multiple items at a time.
WOCheckBox	A check-box user interface control.
WOConditional	Controls whether a portion of the HTML page will be generated.
WOForm	A container element that generates a fill-in form.
WOHyperlink	Generates a hypertext link.
WOImage	Displays an image.
WOImageButton	A graphical submit button.
WOJavaScript	Lets you embed a script written in JavaScript in a dynamically-generated page.
WORadioButton	Represents an on-off switch.
WORepetition	A container element that repeats its contents (that is, everything between the <WEBOBJECT...> and </WEBOBJECT...> tags in the template file) a given number of times.
WOResetButton	A reset button.
WOString	A dynamically generated string.
WOSubmitButton	A submit button.
WOText	A multi-line field for text input and display.
WOTextField	A text input field.

For a complete list of dynamic elements, along with their parameters, see the *Dynamic Elements Reference*.

An HTML template can also contain a reference to another component (called a *reusable component* or *subcomponent*) that represents a portion of an HTML page. This reference behaves just like a reference to a dynamic element.

Code or Script File

You use the *code file* to define your component's attributes and actions. The attributes are called *instance variables*, and the actions are called *methods*. Files containing your code can be found in Project Builder's Classes suitcase, as previously discussed.

Action Methods

Your component code contains *action methods*. An action method is a method you associate with a user action—for instance, clicking a submit button or a hyperlink. There are two types of action methods you can implement: those for *component action* requests and those for *direct action* requests.

- Elements bound to a component action pass the action message to the session object and on to the request component. State is restored by the session object.
- Elements bound to a direct action send their action message directly to the object that can handle it. Direct actions aren't required to use session objects and thus can be stateless.

Your components can contain a mix of direct and component actions.

For a more detailed discussion of action methods, see "Action Methods" on page 63.

Bindings

Bindings associate a component's template with its code. You use a *declarations file* (*myComponent*.**wod**) to define the bindings or mapping between the methods and variables you defined in your code and the

<WebObject> tags in your template that specify dynamic elements and shared components.

Resources

The Resources suitcase is used primarily to hold two types of files: API files, which declare any methods and instance variables that your custom components need to make public; and model files. A model file is used by the Enterprise Objects Framework to store the mapping between a relational database schema and your enterprise objects. As well, the model file stores information needed to connect to the database server.

Web Server Resources

Project Builder's Web Server Resources suitcase holds those resources used by the application in the client interface. This includes Java classes for applets, images, sounds, and other resources that need to be vended by the web server.

Frameworks

Project Builder's Frameworks suitcase contains any custom frameworks you may have developed (perhaps containing business-specific classes that you've packaged into a framework for re-use) along with the standard frameworks supplied with WebObjects.

Nearly all WebObjects applications employ the frameworks listed in the following table. Applications that don't make use of the Enterprise Objects Framework don't need EOAccess and EOControl.

Framework	Description
WebObjects	Used by all WebObjects applications, this framework contains the application, session, component, and adaptor classes, as well as classes that perform request handling and state management.
WOExtensions	This framework serves as a repository of reusable components that you can use to enhance your application's user interface. This framework also contains components that display application statistics and notify you when exceptions are raised within your application. Source code for this framework is also supplied as an example.

Framework	Description
DirectToWeb	A framework for dynamically generating user interfaces (pages or parts of pages) based upon the structure of your database as specified in your model.
EOAccess	Allows your application to interact with database servers at a high level of abstraction. Provides a fine level of control over database operations while allowing to you work with database records as objects.
EOControl	These classes allow you to write enterprise objects that have no dependencies on the user interface and the storage mechanism being used.
Foundation	An operating system-independence layer that provides memory management, fundamental object behavior, access to lower-level services, and classes that implement important features such as collections, strings, and threads.

In Java, WebObjects classes are in the package **com.apple.yellow.webobjects**. Similar packages corresponding to the other frameworks listed above can be found under **com.apple.yellow**.

Developing a WebObjects Application

Developing a WebObjects application is a matter of creating your templates, bindings, and code files and, if your application uses Java Client, your interface files. Although these files are text based and thus could be created using a text editor, WebObjects provides graphical tools that simplify the entire process. The primary tools you use when developing WebObjects applications are:

- **Project Builder**. As its name implies, Project Builder manages all of the constituent parts of your application, including source code files, WebObjects components, frameworks, makefiles, graphics and sound files, and the like. It is from within Project Builder that you edit your code files, compile, debug, and launch your application for development testing. Project Builder's wizards help you to create new WebObjects components. You also can launch the other development tools from within Project Builder.

- **WebObjects Builder**. You use WebObjects Builder to edit your application's components. WebObjects Builder allows you to graphically edit a component's HTML template. If you prefer, you can switch to "raw mode," from which you can edit the template as an HTML text file. WebObjects Builder also allows you to graphically bind the dynamic elements on your template to variables and methods within your code.

- **EOModeler**. EOModeler maintains the mapping between your enterprise objects and your database schema. It presents you with the tables and columns within your database and allows you to establish relationships between them. EOModeler maintains an EOModel file, which your application then uses to determine how enterprise data is to be stored and retrieved to the data store.

- **Interface Builder**. This is a graphical tool that you use to create Java Client interfaces. It not only allows you to visually lay out your interface, it also allows you to graphically make connections between your code and your interface elements.

Where To Go From Here

If you're new to WebObjects programming, begin by reading the book *Getting Started With WebObjects*. It contains tutorials that teach the mechanics of creating a WebObjects application as well as the basic concepts behind WebObjects.

If you want to write an application that accesses a database, you'll need to use enterprise objects in conjunction with WebObjects. Although database access is covered in the tutorials in *Getting Started With WebObjects*, you'll want to consult the *Enterprise Objects Framework Developer's Guide* for more in-depth information.

Other valuable information can be found online. To access online documentation, use the WebObjects Info Center. You can access the WebObjects Info Center on Windows NT systems by selecting Start ►Programs ►WebObjects ►Info Center.

Among the books that are available online are:

- *WebObjects Tools and Techniques* describes WebObjects Builder and Project Builder and shows how to use them to create WebObjects applications.
- *Topics in WebObjects Programming* consists of a set of single-subject articles that each focus on a topic of interest to a WebObjects programmer. Many of these articles include code samples or are accompanied by a fully-working example.
- *Serving WebObjects* describes how to administer and deploy WebObjects applications after you've written them.
- The *WebObjects Framework Reference* provides a complete reference to the classes in the WebObjects framework. Reference material is provided for both the Java and Objective-C languages.
- The *Dynamic Elements Reference* documents the dynamic elements provided with WebObjects and shows examples of how to use them.

Chapter 2

Dynamic Elements

In the previous chapter, you learned that a WebObjects application is made up of components, which in turn are made up of dynamic elements. Dynamic elements are the basic building blocks of a WebObjects application. They link an application's behavior with the HTML page shown in the web browser, and their contents are defined at run-time.

There are two types of dynamic elements that you can place in a component:

- Server-side dynamic elements
- Client-side Java components

This chapter describes each of these types and tells you how to decide when to use them. Before reading it, you should be familiar with the concepts presented in the previous chapter. To learn the mechanics of using dynamic elements, see the online book *WebObjects Tools and Techniques*.

Server-Side Dynamic Elements

Server-side dynamic elements are the simplest type of element to create and are supported by all web browsers. Needless to say, they are the most commonly used.

Server-side dynamic elements produce HTML at run-time. This HTML is composed of the same HTML elements you use when you're creating a static web page. Like static elements, dynamic elements display formatted text, images, forms, hyperlinks, and active images. WebObjects provides several dynamic elements. For a complete list, see the online book *Dynamic Elements Reference*.

For an example of dynamic elements in action, consider an application whose first page contains a list of user choices of actions to perform. This page is shown in Figure 9.

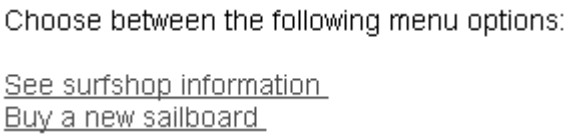

Figure 9. Main Page

This list could be hard-coded into an HTML page, but it is more extensible if it the list is produced using dynamic elements in a component. (Because this is the first component in the application, it is called Main.) Figure 10 shows how this same part of the page looks in WebObjects Builder.

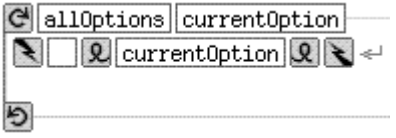

Figure 10. Main Page in WebObjects Builder

The elements shown in Figure 10 are a WORepetition, a WOHyperlink, and a WOString. The WORepetition element corresponds to a **for** loop in C code. That is, it iterates through a list of items and, for each item in that list, prints its contents. In this example, the contents are a WOHyperlink and a WOString. The WOHyperlink is a hyperlink whose destination is determined at run-time, and the WOString is a string whose contents are determined at run-time.

When you run the application, the WORepetition walks through an array of strings that the component's code supplies. For each item in the array, it displays a hyperlink whose text is the text of the string item in the array. In this array, there are two strings—"See surfshop information" and "Buy a new sailboard"—so the WORepetition creates two hyperlinks, each containing the appropriate text.

As the name implies, server-side dynamic elements operate entirely on the server (see Figure 11). That is, when a server-side dynamic element is asked to draw itself, it produces HTML code that should form part of a page, the page is constructed, and then the entire page is sent from the server to the client. Later in this chapter, you'll learn about client-side

components, which transport values and state from the server to the client and then draw themselves on the client machine.

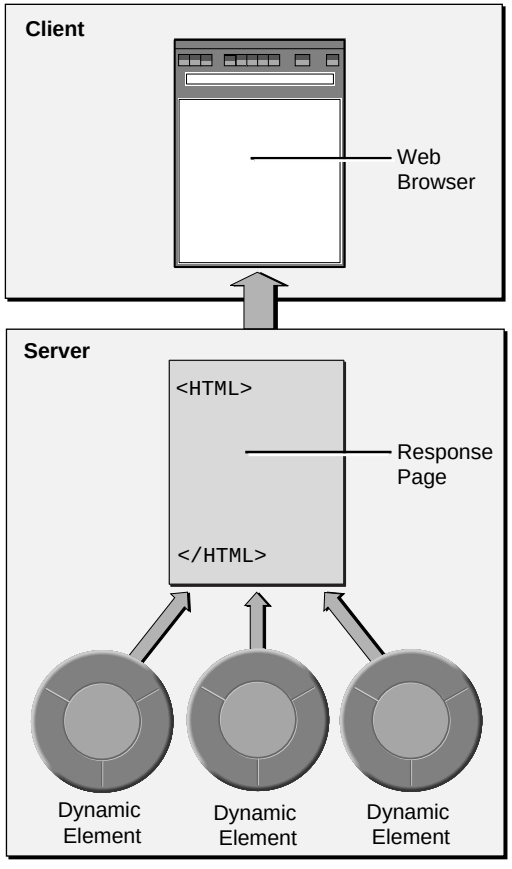

Figure 11. Server-Side Dynamic Elements

How Server-Side Dynamic Elements Work

To learn how server-side dynamic elements work, consider how the example page shown in Figure 10 would look in WebObjects Builder's raw mode:

```
Choose between the following menu options:<BR><BR>
<WEBOBJECT NAME="OPTION_REPETITION">
  <WEBOBJECT NAME="OPTION_LINK">
      <WEBOBJECT NAME="OPTION_NAME"></WEBOBJECT>
  </WEBOBJECT>
</WEBOBJECT>
```

Each WEBOBJECT tag denotes the position of a dynamic element. Notice that the tag specifies only where the dynamic element should go; it does not specify the dynamic element's type. The type is specified in the **.wod** file:

```
OPTION_REPETITION:WORepetition {
      list = allOptions;
      item = currentOption
};
OPTION_LINK:WOHyperlink {
      action = pickOption
};
OPTION_NAME:WOString {
      value = currentOption
};
```

In the **.wod** file, each element is identified by name and then its type is specified. The outermost dynamic element in the HTML file (OPTION_REPETITION) defines a WORepetition, the next element is a WOHyperlink, and the innermost element is a WOString.

Each type specification is followed by a list of *attributes*. Dynamic elements define several attributes that you bind to different values to configure the element for your application. Usually, you bind attributes to variables or methods from your component's code (see Figure 12).

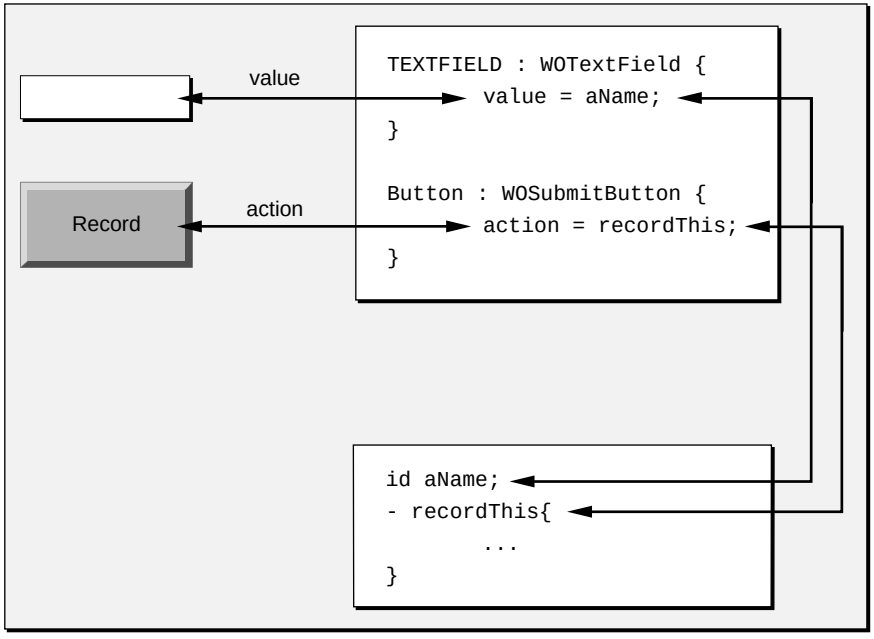

Figure 12. Dynamic Element Bindings

In our example, the Main component binds to two attributes of WORepetition: **list** and **item**. The **list** attribute specifies the list that the WORepetition should iterate over. The **item** attribute specifies a variable whose value will be updated in each iteration of the list (like an index variable in a **for** loop). The Main component binds the **list** attribute to an array named **allOptions** and the **item** attribute to a variable named **currentOption**.

The WOHyperlink has an **action** attribute, which is bound to a method called **pickOptions**. The **action** attribute specifies a method that should be invoked when the user clicks the link. In this case, the **pickOptions** method determines which link the user clicked and then returns the appropriate page.

Finally, the WOString element defines a **value** attribute, which specifies the string you want displayed. This **value** attribute is bound to the **currentOption** variable, which is also bound to the **item** attribute of the WORepetition. As you'll recall, **currentOption** is updated with each iteration that the WORepetition makes. So, for each item in the

allOptions array (assigned to the WORepetition's **list** attribute), the WORepetition updates the **currentOption** variable to point to that item and then the WOString prints it on the page.

Binding Values to Dynamic Elements

In the previous example, all of the dynamic elements are bound to variables and methods from the component that contains them (the Main component). It's common to bind to variables and methods declared directly in the current component; however, you can bind to any value that the component can access.

This means, for instance, that you can bind to variables from the application or session object because the WOComponent class declares two instance variables, **application** and **session**, which point to the current application and the current session. For example, you might define a component that displays relevant information about the application, including the date and time the application was started. It makes sense for the application object to store this date and time. Your component's **.wod** file would access it through this declaration:

```
UP_SINCE:WOString {value = application.upSince.description};
```

To retrieve a value from this binding, WebObjects uses *key-value coding*, a standard interface for accessing an object's properties either through methods designed for that purpose or directly through its instance variables. With key-value coding, WebObjects sends the same message (**takeValue:forKey:**, or **takeValueForKey** in Java) to any object it is trying to access. Key-value coding first attempts to access properties through accessor methods based on the key's name.

For example, to resolve the binding for the WOString element in the above component using key-value coding, WebObjects performs the following steps:

- It resolves the value for the **application** key by looking for a method named **application** in the component object.

  In this case, WOComponent defines the **application** method, which returns the WOApplication object.

- It resolves the value for the **upSince** key by looking for a method named **upSince** in the application object.

  If the method is not found, it looks for an **upSince** instance variable. In this case, the **upSince** instance variable is defined in the application's code file.

- It resolves the value for the **description** key by looking for a method named **description** in the **upSince** object.

  Because **upSince** is a date object, it defines a **description** method, which prints the object's value as a string.

  Note: The Java equivalent of the **description** method is **toString**, but you must use the WebScript name for methods and literals in the **.wod** file even though the application is written in Java.

Here are the general rules for binding dynamic element attributes:

- You must bind to a variable or method accessible by the current component. (You can also bind to constant values.)

- If you bind to a method, the method must take no arguments. (If you need to bind to a method that takes arguments, you can wrap it inside of a method that doesn't take arguments.)

- You can bind to any key for objects that define keys.

  For example, dictionary objects store key-value pairs. Suppose you declare a **person** dictionary that has the keys **name**, **address**, and **phone**. These keys aren't really instance variables in the dictionary, but because WebObjects accesses values using key-value coding, the following binding works:

  ```
  myString : WOString { value = person.name };
  ```

  Enterprise objects also define keys, so the same binding would work if **person** was an enterprise object.

Note: Be aware that **value = person.count** will not work, as **count** is assumed to be a dictionary key.

- You must use the Objective-C names for methods and literals.

  Even if your entire application is written in Java, you must use the Objective-C names for methods and for literals. For example, you must use **YES** instead of **true**, **NO** instead of **false**, and **description** instead of **toString**.

Declarations File Syntax

As you've seen, the **.wod** file specifies nearly all of the information necessary to create a dynamic element. Because you usually create dynamic elements and their bindings using WebObjects Builder, you normally don't have to worry about the syntax of the **.wod** file. However, here it is for the curious:

```
elementName : elementType{attribute =value; attribute =value; ...};
```

As described in the previous section, *value* can be a constant, variable, or method. It can also be a string of keys joined by a dot, similar to the Java syntax for sending messages but without the parentheses. For example:

```
application.upSince.description
```

Client-Side Java Components

Instead of using server-side dynamic elements, you could create Java applets that run on the client. Typically, applets are downloaded to the client once and then have virtually no communication with the server. Client-side Java components, however, run on the client and continuously synchronize their states with objects on the server. Client-side components can also trigger action methods on the server. For this reason, they may be said to work in virtually the same way as server-side dynamic elements.

Prior to WebObjects 4.0, client-side components were the recommended way to enhance web pages with complex custom controls—controls that provided greater functionality and interactivity than those supported directly in HTML. Now, however, WebObjects allows you to create true three-tier applications (Web client, application server, and database

server), with the interface written entirely in Java and running completely on the client. This allows your WebObjects applications to have a much richer, more responsive, and more active interface than traditional HTML-based web applications, while running on all Java-enabled platforms and browsers that support the "Swing" user interface toolkit. Although client-side components are still supported, the new JavaClient feature is now the preferred way to build client-side Java applications in WebObjects

Dynamic Elements vs. Client-Side Components

Although an application constructed using client-side components aren't nearly as powerful or flexible as one constructed with a pure Java Client interface, client-side components do give you greater control over the appearance of your application when compared to an application that uses only dynamic elements and static HTML elements in its interface. When compared only against server-side dynamic elements, client-side components have these advantages:

- Client-side components allow you to update UI elements without reloading the page.

  WebObjects applications are event driven. The events that trigger actions are HTTP requests. A WebObjects application receives an HTTP request from the client, processes it, and returns a response page. That is, the only communication that takes place between the client and the WebObjects application on the server results in a page being redrawn (or a new page being generated).

  When client-side components are used, an HTTP request can result in either the re-synchronization of state or the return of a new page. Thus, state can be synchronized without the page having to be redrawn (see Figure 13).

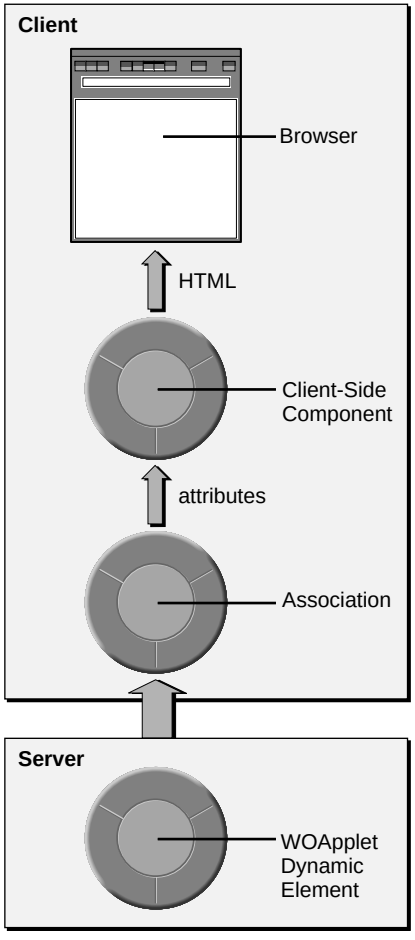

Figure 13. Client-Side Java Components

- Client-side components are more flexible than server-side dynamic elements.

  Server-side dynamic elements always generate HTML, which means that they are limited to what HTML looks like and what HTML can do. You can create client-side components that look like just about any imaginable control: a dynamic calendar, a spreadsheet, or a graphing tool.

One disadvantage to using client-side components is that they require a Java-enabled browser. Thus, you can use client-side components only when you can be certain all of your users will have Java-enabled browsers. If you can't guarantee this, you should limit your application to server-side dynamic elements.

How Client-Side Components Work

A client-side component is really just a special case of a particular server-side dynamic element named WOApplet. You use WOApplet when you want to include any Java applet in a WebObjects application. The difference between a client-side component and other Java applets is that client-side components can communicate with the server.

When you look at client-side component's bindings in the **.wod** file, it looks like this example:

```
INPUTFIELD : WOApplet {
     code = "next.wo.client.controls.TextFieldApplet.class";
     codebase = "/WebObjects/Java";
     archive = "woextensions.jar";
     agcArchive = "woextensions.jar";
     width = "200";
     height = "20";
     associationClass =
"com.apple.client.webobjects.SimpleAssociation";
     stringValue = inputString
};
```

Like any other server-side dynamic element, the WOApplet's definition contains a list of attributes bound to constants or variables in the component's code.

The **code** attribute specifies which client-side component this WOApplet should download. The **codebase** attribute specifies the path of the component relative to your web server's document root. (For the provided client-side components, this path is always **/WebObjects/Java**.)

The **archive** attribute specifies **.jar** files that should be pre-loaded onto the client machine. If you don't use this attribute, the applet downloads Java **.class** files from the server one by one as it needs them. With the **archive** attribute, you can package all necessary Java classes into archive files, and they are downloaded once. However, only web browsers that have Java 1.1 support can use **.jar** files. Because Java 1.1 is fairly new, there's a good chance your users use browsers that don't support **.jar** files. All of the

provided client-side components are packaged in a single archive file named **woextensions.jar**.

The **agcArchive** and **associationClass** attributes differentiate the client-side components from any other applet you might include in your application. The **agcArchive** attribute specifies the **.jar** file that contains the AppletGroupController object. AppletGroupController is a hidden Java applet (on the client) that controls the visible applets and handles communication back to the server. AppletGroupController works in conjunction with the object specified by the **association** attribute. The **association** attribute specifies a subclass of **com.apple.client.webobjects.Association** that the component uses to communicate with the application on the server. The Association object can get and set component state and cause methods to be invoked in the server when actions are triggered in the client. The most common association is **com.apple.client.webobjects.SimpleAssociation**. When creating your own client-side components, you can either use it or define your own association. Creating your own association is useful when you don't have the source code for your client-side component.

The final attribute, **stringValue**, is an attribute specific to the TextFieldApplet component. The Association object assigns the value of the **inputString** variable to be the value of the text field on the client and keeps the two objects in sync so that they always have the same value.

Note: Netscape Navigator 4 supports only a single **.jar** file per applet, and if there is an interface class to support the applets (that is, if you are using AppletGroupController), that object must be found in the same archive file. Thus if you must support Netscape Navigator 4, you should create one **.jar** file that contains all of your Java client-side classes, all third-party classes that you are using, and all classes in **com.apple.client.webobjects**. Bind that **.jar** file to the **archive** attribute of all of the WOApplets on your page. Bind the same **.jar** file to the **agcArchive** attribute of the first WOApplet on the page.

Creating Client-Side Components

You can create your own client-side component if the components supplied with WebObjects don't suit your needs or if you already have a Java applet that you would like to use as a client-side component. You can create a client-side component out of any Java applet, provided you know some details about it. You must know the applet's accessor methods for setting and getting state, and you must know how to detect when the applet has triggered an action (for applets that trigger actions). How you create a client-side component depends on whether you have source code for the applet.

If you don't have the applet's source code, you must create your own subclass of **com.apple.client.webobjects.Association**. As explained in "How Client-Side Components Work" (page 53), client-side components use an Association to communicate with the component on the server. Associations can extract values from the client-side component, set values in the client-side component, and trigger action methods on the server.

If you do have the source code, you may still want to provide your own subclass of Association. However, you're more likely to want to use the provided subclass SimpleAssociation (which is what all of the client-side components packaged with WebObjects use). All associations are linked on one side to one of your applets, and on the other to an AppletGroupController. AppletGroupController is a hidden Java applet (on the client) that controls the visible applets and handles communication back to the server (see Figure 14). The AppletGroupController accesses each of the applets on the page using an Association. It is through these Associations that the data or state each applet manages is passed to the AppletGroupController and, through it, to the server. When an applet fires an action on its association, the AppletGroupController does what is necessary to ensure that the bound method in the server is invoked. An AppletGroupController, once downloaded, knows what class of Association to use and what the destination applets are. To determine these, it inspects the visible applets on the page and looks for some special parameters.

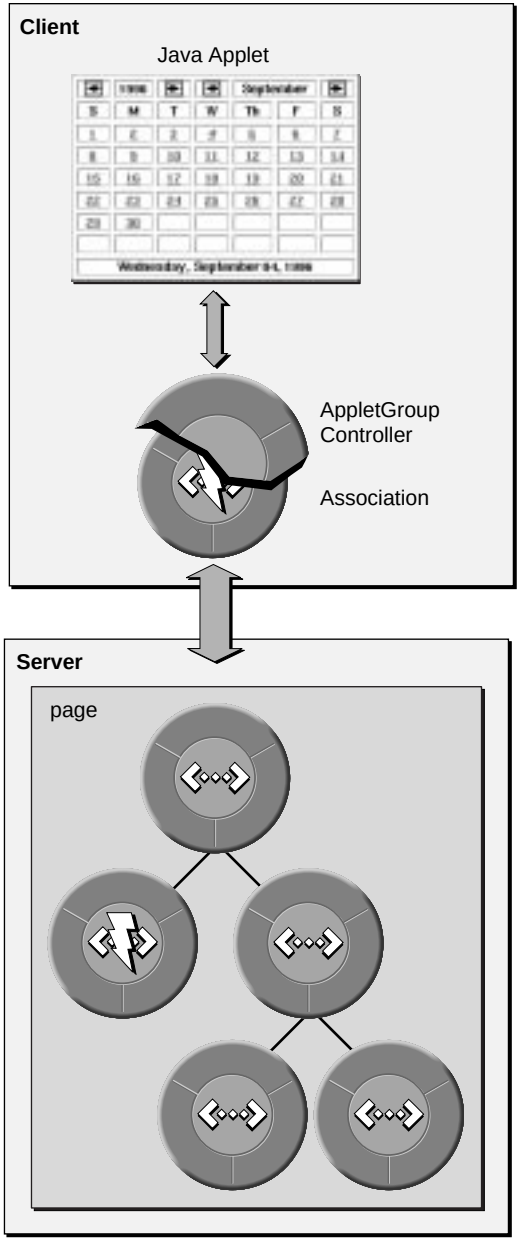

Figure 14. The Principal Objects Involved in Client-Side Components

SimpleAssociation objects don't get or set values themselves; instead, they rely on the actual client-side components and on the AppletGroupController to do so. This means that if you want to use the SimpleAssociation class, you must modify the applet so that it implements the SimpleAssociationDestination interface. This interface defines the methods that are used to get and set values.

When You Have an Applet's Source Code

If you write an applet, or acquire the source code for an applet, you can follow these steps to give the applet the associative behavior it needs to be a client-side component:

1. In Project Builder, add the ClientSideJava subproject to your project. To do so, double-click the word "Subprojects" in the browser and then choose **ClientSideJava.subproj** in the Open panel.

2. Add your class to the ClientSideJava subproject. Double-click Classes in the subproject and then choose your **.java** file in the Open panel.

3. In the class declaration, insert the "implements SimpleAssociationDestination" clause.

```
public class MyApplet extends Applet implements
SimpleAssociationDestination {
     ....
}
```

4. Implement the **keys** method to return a list (Vector) of state keys managed by the applet.

```
public Vector keys() {
    Vector keys = new Vector(1);
    keys.addElement("title");
    return keys;
}
```

5. Implement the **takeValueForKey** and **valueForKey** methods to set and get the values of keys.

```
  synchronized public Object valueForKey(String key) {
      if (key.equals("title")) {
          return this.getLabel();
      }
  }

  synchronized public void takeValueForKey(Object value,
String key) {
      if (key.equals("title")) {
          if ((value != null) && !(value instanceof String)
{
              System.out.println("Object value of wrong
type set for key
              'title'.  Value must be a String.");
          } else {
              this.setLabel(((value == null) ? "" :
(String)value));
      }
  }
```

You should be able to access the keys directly or, ideally, through accessor methods (in this example, **getLabel** and **setLabel**). It is a good idea to use the **synchronized** modifier with **takeValueForKey** and **valueForKey** because these methods can be invoked from other threads to read or set data.

The value for a key must be a property-list type of object (either singly or in combination, such as an array of string objects). The corresponding property-list type of objects for Objective-C and Java are:

Objective-C	Java
NSString	String
NSArray	Vector
NSDictionary	Hashtable
NSData	byte[]

The remaining steps apply only if the applet has an action.

6. Declare an instance variable for the applet's Association object and then, in **setAssociation**, assign the passed-in object to that variable.

```
  protected Association _assoc;
  ...
  synchronized public void setAssociation(Association
assoc) {
      _assoc = assoc;
  }
```

   The Association object must be stored so that it can be used later as the receiver of the **invokeAction** message. The Association forwards the action to the AppletGroupController, which handles the invocation of the server-side action method.

7. When an action is invoked in the applet, send **invokeAction** to the applet's Association.

```
  synchronized public boolean action(Event evt, Object
what) {
      if (_assoc != null) {
          _assoc.invokeAction("action");
      }
      return true;
}
```

When You Don't Have an Applet's Source Code

If you have an applet but do not have the source code for it, you must follow these steps to create an Association class for it:

1. Declare a subclass of the Association class.

```
  class MyAssociation extends Association {
      ...
  }
```

2. Implement the **keys** method to return a list (Vector) of keys managed by the applet. See "When You Have an Applet's Source Code" (page 57) for an example.

3. Implement the **takeValueForKey** and **valueForKey** methods to set and get the values of keys. Use Association's **destination** method to obtain the destination object (that is, the applet).

```
  synchronized public Object valueForKey(String key) {
      Object dest = this.destination();
      if (key.equals("title")) {
          return ((MyApplet)dest).getLabel();
      }
  }

  synchronized public void takeValueForKey(Object value,
String key) {
      Object dest = this.destination();
      if (key.equals("title")) {
          if ((value != null) && !(value instanceof
String)) {
              System.out.println("Object value of wrong
type set for key
              'title'.  Value must be a String.");
      } else {
          ((MyApplet)dest).setLabel(((value == null)
              ? ""
              : (String)value));
      }
  }
```

   Note that the class of the destination applet (in this example, MyApplet) must be cast.

   If the applet triggers an action method, it must have some mechanism for communicating this event to observers (such as an **observeGadget** method).

4. The Association responds to the triggering of the applet's action by sending **invokeAction** to itself.

```
  // fictictious method
  public void observeGadget(Object sender, String action)
{
      if ((sender instanceof Gadget) &&
action.equals("vacuum")) {
          this.invokeAction(action);
      }
  }
```

Note that in this hypothetical example, the Association must first set itself up as an observer in the applet.

Chapter 3

Common Methods

The methods that you write for your WebObjects application provide the behavior that makes your application unique. Because you are writing subclasses of WOApplication, WOSession, WOComponent, and WODirectAction you inherit the methods provided by those classes. These inherited methods take care of the details of receiving HTTP requests and generating responses. However, you'll sometimes find that you need to override some of the inherited methods to perform certain tasks.

This chapter describes the types of methods that you generally write in a WebObjects application. These types are:

- Action methods
- Initialization and deallocation methods
- Request-handling methods

In cases where you override existing methods, those methods are invoked at standard, predictable times during the application's request-response loop (the main loop for a WebObjects application). For background on the request-response loop, see the chapter "WebObjects Viewed Through Its Classes" (page 95).

As you're writing methods, refer to the class specifications for WOApplication, WOSession, WOComponent, and WODirectAction to learn which messages you can send to these objects. The class specifications are in the online book *WebObjects Class Reference*.

Action Methods

An *action method* is a method you associate with a user action—for instance, clicking a submit button, an active image, or a hyperlink. There are two types of action methods that you can implement: *component actions* and *direct actions*. Component actions are the default. See "What Is a WebObjects Application?" (page 17) for a general description of the differences between component actions and direct actions and when you might want to use each one.

Component Actions

In component actions, the component currently being displayed defines the action method to be performed. For example, the first page of a WebObjects application is the Main component, so Main defines the first action method to be performed.

To associate the component's action method to a user action, you map it to a dynamic element that has an attribute named **action** (such as a WOSubmitButton, WOActiveImage, or WOHyperlink). When the user performs the associated action, your method is invoked. For example, in the HelloWorld example application, the submit button is mapped to a method named **sayHello** in the Main component. When a user types in a name and clicks the button on the main page, it initiates the application's request-response loop, and **sayHello** is invoked.

Action methods take no arguments and return a page (component) that will be packaged with an HTTP response. For example, the **sayHello** method creates a new page named Hello and sends that page the name the user has typed into the text field.

```
//WebScript HelloWorld Main.wos
- sayHello
{
      id nextPage;
      nextPage = [self pageWithName:@"Hello"];
      [nextPage setVisitorName:visitorName];
      return nextPage;
}
```

If you're programming in Java, the **sayHello** method looks like this:

```
//Java HelloWorld Main.java
public WOComponent sayHello() {
      Hello nextPage = this.pageWithName("Hello");
      nextPage.setVisitorName(visitorName);
      return nextPage;
}
```

In this example, the component Main is used to generate the page that handles the user request, and the component Hello generates the page that represents the response. Main is the *request component* or the *request page*, and Hello is the *response component* or the *response page*.

It's common for action methods to determine the response page based on user input. For example, the following action method returns an error page if the user has entered an invalid part number (stored in the variable **partnumber**); otherwise, it returns an inventory summary:

```
// WebScript example
- showPart {
      id errorPage;
      id inventoryPage;

      if ([self isValidPartNumber:partnumber]) {
          errorPage = [self pageWithName:@"Error"];
          [errorPage setErrorMessage:@"Invalid part number %@.",
              partnumber];
          return errorPage;
      }
      inventoryPage = [self pageWithName:@"Inventory"];
      [inventoryPage setPartNumber:partnumber];
      return inventoryPage;
}

// Java example
public WOComponent showPart() {
      Error errorPage;
      Inventory inventoryPage;

      if (isValidPartNumber(partNumber)) {
          errorPage = (Error)this.pageWithName("Error");
          errorPage.setErrorMessage("Invalid part number " +
              partnumber);
          return errorPage;
      }
      inventoryPage = (Inventory)this.pageWithName("Inventory");
      inventoryPage.setPartNumber(partnumber);
      return inventoryPage;
}
```

Component action methods don't have to return a new page. They can instead direct the application to use the request page as the response page by returning **nil** (**null** in Java). For example, the following action method records the name of the last visitor, clears the text field, and redraws the Main component, which contains it:

```
// WebScript Main.wos
- recordMe
{
      if ([aName length]) {
          [[self application] setLastVisitor:aName];
          [self setAName:@""]; // clear the text field
      }
}
```

```
// Java Main.java
public WOComponent recordMe
{
      if (aName.length != 0) {
          ((Application)application()).setLastVisitor(aName);
          aName = ""; // clear the text field
      }
      return null;
}
```

Note: Always return **nil** (**null**) in an action method instead of returning **self** (**this**). Returning **nil** uses the request component as the response component. Returning **self** uses the *current* component as the response component. At first glance, these two return values seem to do the same thing. However, if the action method is in a component that's nested inside of the request component, a return value of **self** will make the application try to use the nested component, which represents only a portion of the page, as the response component. This, most likely, is not what you want. Therefore, it is safer to always return **nil**.

Direct Actions

If you want your application to perform direct action request handling instead of component action request handling (as described in "The Request-Response Loop" on page 30), you place your action methods in a different class and you write them slightly differently.

With direct actions, there is no concept of a request page that can perform the requested action. (Recall from the previous section that the Main component is the request page for the **sayHello** action, meaning that Main implements the **sayHello** method.) Instead, the direct action method is defined in a subclass of WODirectAction.

To associate a direct action method to a user action, bind the dynamic element's **directActionName** attribute to the action method's name (enclosing the **directActionName** attribute value in double-quotes), and optionally bind the element's **actionClass** attribute to the name of the WODirectAction subclass that implements the action method. If you don't specify the element's **actionClass**, a default class name of "DirectAction" is assumed. As well, if you've specified an **actionClass** or WODirectActionRequestHandler is the default request handler, you can omit the **directActionName** and WebObjects will invoke the "defaultAction" method within the specified (or default) **actionClass**.

For an example of how to make WODirectActionRequestHandler the default request handler, see "Setting the Default Request Handler" on page 69.

When working with elements inside a WOForm, note that the WOForm's request handler determines how the form's elements are processed. If the WOForm uses component action request handling, setting the **actionClass** or **directActionName** attributes on action elements inside the form—such as the WOSubmitButton—will have no effect. Instead, you must either assign the **actionClass** and **directActionName** attributes to the WOForm element itself, or you must set the WOForm's **actionClass** as follows, which then causes the WOForm's direct action element attributes to be used:

```
myForm : WOForm {
  actionClass = "";
}
button : WOSubmitButton {
  directActionName = "sayHello";
  actionClass = "MyAction";
}
```

Direct action method names are derived from action names by appending "Action" to the action name. Thus, setting your element's **directActionName** attribute to "sayHello" as in the above code causes MyAction's **sayHelloAction** method to be invoked when the WOSubmitButton is clicked.

The WODirectAction class is simply a container for action methods. When you create a WebObjects application project in Project Builder, you automatically get a subclass of WODirectAction named DirectAction (DirectAction is the default name for the WODirectAction subclass; you can rename it, if you prefer). Each action method in your WODirectAction class must end with the string "Action" and should return either a WOComponent or a WOResponse object. For example:

```
- (WOComponent *)sayHelloAction
```

or:

```
public WOComponent sayHelloAction()
```

Note that your application can use as many WODirectAction subclasses as necessary.

Your direct action method should return an object that conforms to the WOActionResults protocol (Java interface); typically, your method will return a WOComponent or a WOResponse (both implement the methods in WOActionResults). Unlike component actions, direct actions may not return **nil** or **null**.

To use a more complete illustration, in an implementation of the HelloWorld application using direct actions the Main component's declarations file might look like this:

```
myForm : WOForm {
  directActionName = "";
}

textField : WOTextField {
  value = visitorNameIVar;
  name = "visitorName";
}

button : WOSubmitButton {
  directActionName = "sayHello";
  actionClass = "MyAction";
}
```

The **actionClass** and **directActionName** attributes specify that clicking this hyperlink should trigger the MyAction object's **sayHelloAction** method. Note that WOForms cannot have submit buttons to both component and direct actions. Because of this, when using a submit button bound to a direct action the WOForm must be triggered to cooperate. You do this by having an empty binding for direct action, as shown in the previous code excerpt, or by just putting the action directly on the WOForm.

The implementation of **sayHelloAction** is similar to the implementation of **sayHello** shown in the previous section. It creates a new page named Hello and sends that page the name the user has typed into the text field.

```
//WebScript DirectAction.wos
- (WOComponent *)sayHelloAction
{
      id nextPage;
      id aName = [[self request] formValueForKey:@"visitorName"];

      nextPage = [self pageWithName:@"Hello"];
      [nextPage setVisitorName:aName];
      return nextPage;
}
```

```
//Java DirectAction.java
public WOComponent sayHelloAction() {
      Hello nextPage;
      String aName =
(String)request().formValueForKey("visitorName");
      nextPage = (Hello)this.pageWithName("Hello");
      nextPage.setVisitorName(aName);
      return (WOComponent)nextPage;
}
```

Where **sayHelloAction** differs from **sayHello** is that it must perform an additional step: it must extract the form values out of the request itself. The visitor's name is recorded in an instance variable in the Main component (**visitorNameIVar** in the example above). With component actions, any action that is performed by a dynamic element in Main is implemented in Main's code. Because the action is implemented in Main, you can reference the instance variable directly. Direct actions are performed in a separate class and do not have access to Main's instance variables. Instead, direct action methods must extract the information they need from the HTTP request. In this example, the visitor's name is recorded in a text field named "visitorName." Thus, **sayHelloAction** asks for the value for "visitorName" from the request and passes that value to the Hello component.

Suppressing Session IDs in a Direct Action URL

When you construct an HTML template that has some of its components bound to direct actions and some bound to component actions, depending on the placement of your direct action components their URLs may include session IDs. You can prevent the inclusion of a session ID in a direct action URL as shown in the following example:

```
MyLink:WOHyperlink {
  directActionName = "something";
  ?wosid = NO;
}
```

Setting the Default Request Handler

If a request URL doesn't have a request handler key (as is the case with the initial URL used to begin a session with a WebObjects application), WOApplication uses whatever its default request handler is set to be. By default, the default request handler is WOComponentRequestHandler.

If you want to write an application entirely using direct actions, set the default request handler in your WOApplication's **init** method or constructor in this way:

```
// Java implementation
public WOApplication() {
  super();
  ...
  setDefaultRequestHandler(requestHandlerForKey(
      WOApplication.directActionRequestHandlerKey()));
  ...
}
//WebScript implementation
- init {
  self = [super init];
  ...
  [self setDefaultRequestHandler:[self requestHandlerForKey:
      [WOApplication directActionRequestHandlerKey]]];
  ...
  return self;
}
```

Initialization and Deallocation Methods

Like all objects, WOApplication, WOSession, and WOComponent implement initialization methods (or constructors in Java). Because most subclasses require some unique initialization code, these are the methods that you override most frequently. In WebScript, the initialization method is named **init**. In Java, the initialization method is the constructor for the class.

Complementing **init** is the **dealloc** method. This method lets objects deallocate their instance variables and perform other clean-up tasks. The **dealloc** method is used primarily for Objective-C objects. Standard **dealloc** methods in Objective-C send each instance variable a **release** message to make sure that the instance variables are freed. WebScript and Java, because they have automatic garbage collection, usually make a deallocation method unnecessary. If you find it necessary, you can implement **dealloc** in WebScript and **finalize** in Java.

The Structure of init

The **init** method must begin with an invocation of super's **init** method and must end by returning **self**.

```
- init {
      self = [super init];
      /* initializations go here */
      return self;
}
```

Likewise, in Java, the constructor must begin with an invocation of the superclass's constructor (as with all Java classes):

```
public Application() {
      super();
      /* initializations go here */
}
```

Application Initialization

The application **init** method is invoked only once, when the application is launched. You perform two main tasks in the application's **init** method:

- Initialize application variables
- Configure applicationwide settings

For example:

```
// WebScript Application.wos
- init {
      self = [super init];
      lastVisitor = @"";
      [self setDefaultRequestHandler:
          [self requestHandlerForKey:
          [WOApplication directActionRequestHandlerKey]]];
      return self;
}
```

```
// Java Application.java
public Application () {
  super();
  ...
  lastVisitor = "";
  setDefaultRequestHandler(requestHandlerForKey
      (WOApplication.directActionRequestHandlerKey()));
  ...
}
```

This method begins by calling the superclass's **init** method. Then, it initializes the application variable **lastVisitor** to be the empty string. (The application has just started, so there has been no last visitor.) Finally, it sets the default request handler to be the WODirectActionRequestHandler. WODirectActionRequestHandler handles requests for direct actions like the one shown in "Direct Actions" (page 66). You set it to be the default request handler if you want to have initial requests go through the "defaultAction" of DirectAction.

You might want to do other configurations in the application object's **init** method as well. For example, you can control how pages and components are cached and how state is stored. For more information, read the chapter "Managing State" (page 157).

Session Initialization

A session object is created each time the application receives a component action request from a new user. An application may have multiple sessions running concurrently. The session ends when a session time-out value is reached.

Note: By default, applications create session objects during the first cycle of the request-response loop. This is because by default, the first request is handled by the component action request handler, which creates a session. If you change the default request handler to the direct action request handler, you do not have a session object. See "WebObjects Viewed Through Its Classes" (page 95) for clarification.

In the session object's **init** method, you set the session's time-out value and initialize variables that should have unique values for each session. For example, if you wanted to have each session keep track of what number it is, you could do so in the session object's **init** method:

```
//WebScript Session.wos
- init {
      id woApp = [WOApplication application];
      self = [super init];
      [self setTimeOut:120]; // session idle time is 2 minutes.
      [woApp setSessionCount:[woApp sessionCount + 1];
      sessionNumber = [woApp sessionCount];
      return self;
}
```

```
//Java Session.java
public Session() {
      super();
      Application woApp =
(Application)WOApplication.application();
      this.setTimeOut(120);
      woApp.setSessionCount(woApp.sessionCount() + 1);
      sessionNumber = woApp.sessionCount();
}
```

Component Initialization

A component object's **init** method is invoked when the component object is created. Just how often a particular component object is created depends on whether the application object is caching pages. For more information, see "WebObjects Viewed Through Its Classes" (page 95). If page caching is turned on (as it is by default), the application object generally creates the component object once and then restores that object from the cache every time it is involved in a user request. If page caching is turned off, the component object is freed at the end of the request-response loop.

Note: The **pageWithName:** methods shown in the section "Action Methods" (page 63) always create a new component object, even if page caching is turned on.

A component object's **init** method usually initializes component variables. For example, the following method initializes a component variable named **departments**:

```
// WebScript Department.wos
id departments;
- init {
      id departmentsPath;

      [super init];
      departmentsPath = [[[self application] resourceManager]
          pathForResourceNamed:@"Departments.array"
          inFramework:nil
          languages:[[self session] languages]];
      departments = [NSArray
arrayWithContentsOfFile:departmentsPath];
      return self;
}
```

WODirectAction Initialization

Unlike applications, components, and sessions, WODirectAction objects do not persist between cycles of the request-response loop. A WODirectAction object is initialized at the beginning of a direct action request-response loop cycle and is released or marked for garbage collection at the end of the cycle. The designated initializer for WODirectAction is **initWithRequest:**. In Java, the constructor must take a WORequest argument, for example:

```
public DirectAction(WORequest) { ... }
```

In the **initWithRequest:** method (or constructor), you perform anything that should happen before the WODirectAction performs any of the actions that it declares.

Component Action Request-Handling Methods

By default, your application uses the component action request handling loop. (“Setting the Default Request Handler” on page 69 shows you how to make the direct action request handling loop the default for your application.) Component action request handling is performed in three phases, which correspond to three methods you can override:

- Taking input values from the request (**takeValuesFromRequest:inContext:** or **takeValuesFromRequest**)
- Invoking the action (**invokeActionForRequest:inContext:** or **invokeAction**)
- Generating a response (**appendToResponse:inContext:** or **appendToResponse**)

Each of the methods is implemented by WOApplication, WOSession, and WOComponent. In each phase, WOApplication receives the message first, then sends it to the WOSession, which sends it to the WOComponent, which sends it to all of the dynamic element and component objects on the page.

The request-handling methods handle three types of objects:

- A request object (WORequest) is passed as an argument in the first two phases. This object represents a user request. You can use it to retrieve information about the request, such as the method line, request headers, URL, and form values.
- A context object (WOContext) is passed as an argument in all three phases. This object represents the current context of the request. It contains references to information specific to the request, such as the current component, current session, and current request.
- A response object (WOResponse) is passed in the final phase. This object encapsulates information contained in the generated HTTP response, such as the status, response headers, and response content.

You should override these methods if you need to perform a task that requires this type of information or you need access to objects before or after the action method is invoked. For example, if you need to modify the header lines of an HTTP response or substitute a page for the requested page, you would override **appendToResponse:inContext:**.

As you implement request-handling methods, you must invoke the superclass's implementation of the same methods. But consider *where* you invoke it because it can affect the request, response, and context information available at any given point. In short, you want to perform certain tasks before **super** is invoked and other tasks after **super** is invoked.

Request Handling Initialization and Post-Processing

At the beginning of each cycle of the component action request-response loop, a method named **awake** is sent to the WOApplication, WOSession, and WOComponent objects. Like the **init** method or constructor, the **awake** method performs initialization tasks, but **awake** is invoked at a different time during an object's life than the **init** method or constructor is. The **init** message or constructor message is sent once, when the object is first created. In contrast, **awake** is sent at the beginning of each cycle of the request-response loop that the object is involved in. Thus, it may be sent several times during an object's life.

Complementing **awake** is the **sleep** method. The **sleep** method is invoked at the end of each cycle of the component action request-response loop (in contrast to the **dealloc** or **finalize** methods, which are invoked at the end of the object's life). The sleep method is rarely used, but you could use it to perform any clean-up task necessary before the next request begins.

Application Awake

The application's **awake** method is invoked at the start of every cycle of the request-response loop. Therefore, in the **awake** method, you perform anything that should happen before each and every user request is processed. For example, if you wanted to keep track of the number of requests that an application receives, you could do that in the **awake** method:

```
// WebScript Application.wos
- awake {
      ++requestCount;
      [self logWithFormat:@"Now serving request %@",
requestCount];
}

// Java Application.java
public void awake() {
      ++requestCount;
      this.logString("Now serving request " + requestCount);
}
```

Session Awake

After the application object has performed its own **awake** method, it restores the appropriate session object and sends it the **awake** message too.

If you wanted to keep track of the number of requests per session instead of per application, you could do so in the session's **awake** method:

```
- awake {
  requestCount++;
}
```

Component Awake

The component **awake** method is invoked immediately after the **init** method and each time the component object is restored from the page cache. Just as in **init**, you can implement an **awake** method that initializes component variables. For example, a component might have a **shoppingCart** variable that is a snapshot of the session's **shoppingCart** variable. Each time the component is restored from the cache, its **shoppingCart** variable should be updated with the session's **shoppingCart**:

```
// WebScript Car.wos
- awake {
  shoppingCart = [[self session] shoppingCart];
}
```

In general, you use **init** or the component's constructor to initialize component instance variables instead of **awake** because **init** is invoked only at component initialization time, whereas **awake** is potentially invoked much more than that. If, however, you want to minimize the amount of state stored between cycles of the component action request-response loop, you might choose to initialize component instance variables in **awake** and then deallocate them in **sleep** (by setting them to **nil** in WebScript or **null** in Java). For more information, see the chapter "Managing State" (page 157).

Taking Input Values From a Request

The **takeValuesFromRequest:inContext:** method is invoked during the first phase of the component action request-response loop, immediately after all of the objects involved in the request have performed their **awake** methods. When this phase concludes, the request component has been initialized with the bindings made in WebObjects Builder.

You might override **takeValuesFromRequest:inContext:** if you want to perform post-processing on user input. The following example takes the values for the component's **street**, **city**, **state**, and **zipCode** instance variables and stores them in **address** variable formatted as a standard mailing address.

```
// WebScript example
- takeValuesFromRequest:request inContext:context {
  [super takeValuesFromRequest:request inContext:context];
  address = [NSString stringWithFormat:@"%@\n%@, %@  %@",
      street, city, state, zipCode];
}
```

```
// Java example
public void takeValuesFromRequest(WORequest request,
WOContext context) {
  super.takeValuesFromRequest(request, context);
  address = street + city + state + zipCode;
}
```

This first phase of the component action request-response loop is only performed if the request has form values to use as input. The first request that an application receives, for example, would not have input values, and thus **takeValuesFromRequest:inContext:** is not performed for that first request. If you override **takeValuesFromRequest:inContext:**, make sure that your method is only necessary when there are form values; do not override this method to perform something that should occur for every request.

For example, it may be tempting to override **takeValuesFromRequest:inContext:** when you want access to information contained in the request or context objects. Because you can not guarantee that **takeValuesFromRequest:inContext:** will be invoked, you should not do this. Instead, you can access the request and context in the **awake** method this way:

```
- awake {
  id userAgent = [[[self context] request]
      headerForKey:@"user-agent"];
  [self recordUserAgent:userAgent];
}
```

Invoking an Action

The second phase of the component action request-response loop involves **invokeActionForRequest:inContext:**. WebObjects forwards this method from object to object until it is handled by the dynamic element associated with the user action (typically, a submit button, a hyperlink, and active image, or a form).

You rarely need to override **invokeActionForRequest:inContext:**. One reason you might do so is if you want to return a page other than the one requested. This scenario might occur if the user requests a page that has a dependency on another page that the user must fill out first. The user might, for example, finish ordering items from a catalog application and want to go to a fulfillment page but first have to supply credit card information.

Limitations on Direct Requests

Users can access any page in an application without invoking an action. All they need to do is type in the appropriate URL. For example, you can access the second page of HelloWorld without invoking the **sayHello** action by opening this URL:

```
http://serverhost/cgi-bin/WebObjects/Examples/HelloWorld.woa/wo/Hello
```

When a WebObjects application receives such a request, it bypasses the user-input (**takeValuesFromRequest:inContext:**) and action-invocation (**invokeActionForRequest:inContext:**) phases because there is no user input to store and no action to invoke. As a result, the object representing the requested page—Hello in this case—generates the response.

By implementing security mechanisms in **invokeActionForRequest:inContext:**, you can prevent users from accessing pages without authorization, but only if those pages are not directly requested in URLs. To prevent users from directly accessing pages in URLs, override **appendToResponse:inContext:** instead.

Generating a Response

The **appendToResponse:inContext:** method is invoked in the final phase of the request-response loop, during which the application generates HTML for the response page.

Note: Unlike, **takeValuesFromRequest:inContext:** and **invokeActionForRequest:inContext:**, the **appendToResponse:inContext:** method may be invoked by the direct action request-response loop. If the direct action method returns a WOComponent object, that component's **appendtoResponse:inContext:** method is invoked to generate the response.

You can override this method to add to the response content or otherwise manipulate the HTTP response. For example, you can add a cookie to the response as in the following example:

```
- appendToResponse:aResponse inContext:aContext
{
     id aCookie = [WOCookie cookieWithName:@"myCookie"
         value:@"important information goes here"];

     [super appendToResponse:aResponse inContext:aContext];
     [aResponse addCookie:aCookie];
}
```

In a similar manner, you can use **appendToResponse:inContext:** to add text to the response content. In the following example, a component's **appendToResponse:inContext:** method adds bold and italic markup elements around a string's value as follows:

```
id value;
id escapeHTML;
id isBold;
id isItalic;

- appendToResponse:aResponse inContext:aContext
{
     id aString = [value description];

     [super appendToResponse:aResponse inContext:aContext];
     [aResponse appendContentHTMLString:@"<p>"];
     if (isBold) {
         [aResponse appendContentHTMLString:@"<b>"];
     }
     if (isItalic) {
         [aResponse appendContentHTMLString:@"<i>"];
     }

     if (escapeHTML) {
         [aResponse appendContentString:aString];
     } else {
         [aResponse appendContentHTMLString:aString];
     }

     if (isItalic) {
         [aResponse appendContentHTMLString:@"</i>"];
     }
     if (isBold) {
         [aResponse appendContentHTMLString:@"</b>"];
     }
}
```

After you invoke **super**'s **appendToResponse:inContext:**, the application generates the response page. At this point you could do something appropriate for the end of the request. For example, the following implementation terminates the current session:

```
public void appendToResponse(WOResponse response,
WOContext context) {
  super.appendToResponse(response, context);
  session().terminate();
}
```

For more details on each phase of the request-response loop, read the chapter "WebObjects Viewed Through Its Classes" (page 95).

Chapter 4

Debugging a WebObjects Application

In the previous chapters, you learned the pieces of a WebObjects application and the kinds of methods you need to write. Once you've put together an application, you should debug it to make sure it runs properly. The techniques you use to debug vary according to the languages you've used to write the application.

This chapter describes how to debug WebScript code, Java code, and Objective-C code in a WebObjects application.

Before you debug, it's a good idea to test your installation and verify that it works properly. If you haven't already done so, follow the instructions in the online document *Post-Installation Information*.

Launching an Application for Debugging

You debug WebObjects applications using Project Builder, as described in the online book *WebObjects Tools and Techniques*. The executable you launch differs based on which language you used to write the application. This section tells you how to begin a debugging session for WebObjects applications written in each of the three available languages: WebScript, Java, and Objective-C.

Debugging WebScript

To debug WebScript code, you rely on log messages and trace statements described in the section, "Debugging Techniques" (page 86).

If you've written an application entirely in WebScript, you typically debug it by running ***NEXT_ROOT*/Library/WebObjects/Executables/WODefaultApp** from the Project Builder launch panel, as described in *WebObjects Tools and Techniques*. When you do, the output from the debugging and trace statements is displayed in the launch panel.

Debugging Java

If your application is written in Java, you can use Project Builder's Java debugger. For more information on debugging a Java application with Project Builder, see "Debugging Java Applications" in the Java Tutorial (which can be accessed from Project Builder's home page).

In addition to using the Java debugger, you can use the methods described in "Debugging Techniques," below, as well as **System.out.println** statements. Build the executable for your project using Project Builder, then launch that executable in the launch panel. Output from the debugging methods appears in the launch panel.

Debugging Objective-C

If all or part of your application is written in Objective-C, you can use the **gdb** debugger in Project Builder. For more information on debugging an Objective-C application with Project Builder, see Project Builder's online help.

If your application contains WebScript code as well as Objective-C code, you debug the WebScript portion using **debugWithFormat:** and WOApplication trace statements as described in "Debugging Techniques," below.

Debugging Techniques

To debug WebScript, you rely primarily on log messages and trace statements that write to standard output. This section describes the statements you can include in your code to help you debug.

Specifying the Project Search Path

When you use Project Builder to build an application, it places the built application inside of the project directory. For WebObjects applications after you build the project, you'll have a **.woa** directory inside of the project directory. This is typically the copy of the application you want to debug. If you use this default setup, WebObjects uses the components, script files, images, and other resources from the project directory instead

of the copies inside of the **.woa** directory while you debug the project. This way, you can edit scripts (**.html**, **.wod**, **.wos** files) or change image files in your project without having to rebuild or even restart the application.

Sometimes, you want to debug framework code as well as application code. Other times, you might have moved the **.woa** directory outside of the project directory but you still want to use the project's copies of resource files instead of those in the **.woa** directory's. In these cases, you should set the NSProjectSearchPath user default. NSProjectSearchPath should point to the directory that contains all of your projects. WebObjects looks in the directories specified by NSProjectSearchPath for a project that has the same name as the application or framework being loaded (note that the project name is defined inside **PB.project** and is *not* the project's directory name). If it finds a project, it uses the resources from the project directory instead of the resources inside the **.woa** directory.

You can change NSProjectSearchPath on a command line as follows:

```
% defaults write NSGlobalDomain NSProjectSearchPath
@("someDirectory", "someOtherDirectory", ...)
```

Debugging Without a Web Server

WebObjects applications are meant to be used in conjunction with a web server. When you are in development mode, however, you do not have to use a web server to interact with the application. Instead, you simply specify the number of the port where the application should receive requests using the WOPort user default. By default, WOPort is -1, which assigns an arbitrary high port number to the application. Thus, if you specify no port number at all, you can still run your application without a web server. However, it is probably a good idea to assign a specific port number to your application. To do so, use the following command in a command shell window:

```
% defaults write MyWebApp WOPort portNumber
```

When a port number is assigned to an application, you can access it by typing the following URL in the browser (provided the application is already running):

```
http://localhost:portNumber
```

Writing Debug Messages

The method **debugWithFormat:** (in Java, **debugString**) writes a formatted string to standard error (**stderr**).

In WebScript and Objective-C, **debugWithFormat:** works like the **printf()** function in C. This method takes a format string and a variable number of additional arguments. For example, the following code excerpt prints the string "The value of myString is Elvis":

```
myString = @"Elvis";
[self debugWithFormat:@"The value of myString is %@", myString];
```

When this code is parsed, the value of **myString** is substituted for the conversion specification **%@**. The conversion character @ indicates that the data type of the variable being substituted is an object (that is, of the **id** data type).

Because in WebScript all variables are objects, the conversion specification you use must always be **%@**. Unlike **printf()**, you can't supply conversion specifications for primitive C data types such as **%d**, **%s**, **%f**, and so on. (If you do, you might see the address of the variable rather than its value.)

In Java, the equivalent of **debugWithFormat:** is **debugString**. You can send it to WOApplication, WODirectAction, and WOComponent objects. Instead of using **printf**-style specifications, **debugWithFormat:** uses concatenation to construct the string. For example, here's how you'd write the same lines of code in Java:

```
myString = "Elvis";
application().debugString("The value of myString is " + myString);
```

Perhaps the most effective debugging technique is to use **debugWithFormat:** to print the contents of **self**. This prints the values of all of your component variables. For example, this statement at the end of the **sayHello** method in HelloWorld's **Main.wos**:

```
[self debugWithFormat:@"The contents of self in sayHello are %@",
self];
```

produces output that resembles the following:

```
The contents of self in sayHello are <<Main: 0x8cb08 name=Main
subcomponents=0x0> visitorName=frank>
```

Here's how you'd write the same line of code in Java:

```
application().debugString("The contents of this in sayHello are "
  + this.toString());
```

All objects that respond to the **debugWithFormat:** message also respond to the **logWithFormat:** (or **logString**) message. It's recommended to use **debugWithFormat:** for debugging messages because you can turn off the output of these messages with the WODebuggingEnabled user default. By default, WODebuggingEnabled is set to YES so all **debugWithFormat:** messages print to **stderr** while you are in development mode. When you are ready to deploy the application, you can disable the debugging messages with this command:

```
% defaults write MyApplication WODebuggingEnabled NO
```

When WODebuggingEnabled is NO, debugging statements are not printed. Thus, you should use **logWithFormat:** messages for information that you want to print to **stderr** whether or not you are in development mode.

Note: Although setting WODebuggingEnabled to NO prevents debugging statements from being printed, **debugWithFormat:** imposes a small performance penalty each time it is encountered. Once a component has stabilized, you'll want to strip out debugging statements.

Using Trace Methods

WOApplication provides trace methods that log different kinds of information about your running application. These methods are useful if you want to see all or part of the call stack. The following table describes the trace methods:

Method	Description
– trace:	Enables all tracing.
– traceAssignments:	Logs information about all assignment statements.
– traceStatements:	Logs information about all statements.

Method	Description
– traceScriptedMessages:	Logs information when an application enters and exits a scripted method.
– traceObjectiveCMessages:	Logs information about all Objective-C methods invocations.

The output from the trace methods appears in Project Builder's launch panel.

You use the trace methods wherever you want to turn on tracing. Usually, you do this in the **init** method (or constructor) of a component or the application:

```
// Objective-C
- init {
  [super init];
  [[self application] traceAssignments:YES];
  [[self application] traceScriptedMessages:YES];
  return self;
}

// Java
public Main() {
  super();
  this.application.traceAssignments(true);
  this.application.traceScriptedMessages(true);
  .
  .
  .
}
```

Debugging Dynamic Elements and Reusable Components

If a dynamic element's or resuable component's bindings are not working like you expect them to, or you just want more insight into how non-synchronized components work, you can enable debugging for that element. All elements support a **WODebug** attribute. If you bind the **WODebug** attribute to YES, the element prints messages when it resolves its bindings with its parent component. This results in logs like the following being generated:

```
[NestedList:WXNestedList] (item: {label = Alpha.2.1; value =
A.2.1; }) ==> currentItem
[NestedList:WXNestedList] (index: 0) ==> currentIndex
[NestedList:WXNestedList] sublist <== (currentItem.sublist: *nil*)
[NestedList:WXNestedList] (item: {isNew = 1; label = Alpha.2.2;
value = A.2.2; }) ==> currentItem
[NestedList:WXNestedList] (index: 1) ==> currentIndex
[NestedList:WXNestedList] sublist <== (currentItem.sublist: *nil*)
```

If you want, you can customize the messages that the elements print by overriding two methods in WOApplication. These methods are:

- **logTakeValueForDeclarationNamed:type:bindingNamed: associationDescription:value:** (**logTakeValueForDeclarationNamed** in Java)
- **logSetValueForDeclarationNamed:type:bindingNamed: associationDescription:value:** (**logSetValueForDeclarationNamed** in Java)

Isolating Portions of a Page

If a component is producing unexpected HTML output, you can try to isolate the problem by displaying small portions of the page at a time. Use HTML comments (**<!--**) to comment out all but the suspect portion of the page and reload the component. Verify that this portion works as you intend it to. Reduce the size of the commented out portion of the page until more and more of the page is visible in the browser. Continue until you have found the offending area.

Programming Pitfalls to Avoid

This section describes some things to look out for as you debug your application.

WebScript Programming Pitfalls

Because WebScript looks so much like Objective-C and C, it's easy to forget that WebScript isn't either of these languages. When you're debugging WebScript code, watch out for the following tricky spots:

- WebScript supports only objects that inherit from NSObject. As most objects inherit from NSObject, this limitation is easy to overlook. Notably, EOFault does not inherit from NSObject, so you cannot use it in WebScript code.

- The == operator is supported only for NSNumber objects. If you use == to compare two objects of any other class, the operator compares the addresses of the two objects, not the values of the two objects. To test the equality of two objects, use the **isEqual:** or **isEqualToString:** methods.

```
NSString *string1, *string2;

// WRONG!
if (aString1 == aString2) ...

// Right
if ([aString1 isEqualToString:string2]) ...
```

- The postincrement and postdecrement operators are not supported. If you use them, you won't receive an error message. Instead, they behave like preincrement and predecrement operators.

```
i = 0;
if (i++ < 1 )
  // This code never gets executed.
```

For more information, see the chapter "The WebScript Language" (page 201).

Java Programming Pitfalls

When debugging Java code, watch out for the following tricky spots:

- You can't define multiple constructors or overloaded methods for the classes WOApplication, WOSession, WOComponent, or any other class that originates as an Objective-C class. For example, the following code causes your application to crash:

```
public class MyComponent extends WOComponent {
  public void myMethod() { .... }

  //WRONG! Overloaded method causes runtime error.
  public void myMethod(int anInt) { ... }
}
```

- The **pageWithName** method creates the page by looking up and instantiating the component class that has the same name as the argument you provide to **pageWithName**. For this reason, your subclass of WOComponent shouldn't be given a package name. For example, if you create a component named **MyPage.wo** and place its Java file in the package **myClasses.web**, **pageWithName** won't find the **MyPage.class** file.

- Java is a more strictly typed language than is Objective-C or WebScript. If you're more familiar with Objective-C, you'll find that you need to cast the return types frequently. For example, suppose you define a method named **verify** in the file **Session.java** and you want to invoke that method from a component's Java file. To do so, you must cast the return type of the component's **session** method as in the following:

  ```
  // From a component's Java file.
  ((Session)session()).verify();
  ```

  By definition, **session** returns a WOSession object. Because WOSession does not define a method named **verify**, your code won't compile unless you cast the return value of **session** to your WOSession subclass.

Chapter 5

WebObjects Viewed Through Its Classes

As you learned at the end of the first chapter, WebObjects applications respond to HTTP requests and return responses in the form of dynamically generated HTML pages. The main loop of a WebObjects application, in which the application performs this work, is called the request-response loop. You have a very broad understanding of how this works: the web browser sends a request to the HTTP server, which forwards it to the WebObjects adaptor, which translates it into a form that a WebObjects application can understand. For the response, the process is reversed.

This chapter describes in much greater detail what happens during the request-response loop. It does so by describing the request-response loop as WebObjects views it: as a communication between objects. In this chapter, you learn about the objects that are involved at each level of the loop, each object's duty during each part of the request-response loop, and the way these objects generate an appropriate HTML page in response to the user request. You also learn about the two varieties of request handling (component action and direct action) and exactly how these two differ.

In the chapter "Common Methods" (page 61), you learned some of the methods that are invoked during the request-response loop, and you learned about cases where you might want to override these methods. As you write more complex WebObjects applications, it becomes necessary to know exactly what happens at each point in the processing of an HTTP request and the generation of an HTTP response. You should read this chapter to learn that level of detail. You can also refer to the class specifications in the online book *WebObjects Class Reference*.

The Classes in the Request-Response Loop

The request-response loop begins when an HTTP request from a client web browser is handled by the HTTP server. This section starts at that point and then dives into the request-response loop layer by layer, telling you which classes get involved, and at which point. Later sections walk you through the sequence of events that happen during one cycle of the request-response loop and the sequence of events for generating an HTML page.

Server and Application Level

At the server and application level, the request-response loop looks like that shown in Figure 15.

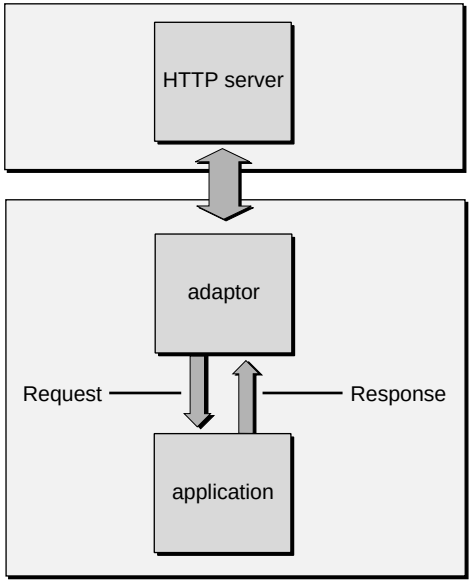

Figure 15. Request-Response Loop: Application and Server Level

The HTTP server forwards a request to the application's adaptor. The adaptor packages the incoming HTTP request in a form the WebObjects application can understand and forwards it to the application. The application determines the type of the request (component action request or direct action request) and then forwards it to the appropriate request handler. The request handler manages the process of request handling and returns the completed response to the application, which passes it on to the adaptor, which gives it to the HTTP server in a form the server can understand.

Two classes are involved at this level:

- WOAdaptor

  Defines the interface for objects mediating the exchange of data between an HTTP server and a WebObjects application. This is an abstract class.

- WOApplication

  Receives requests from the adaptor, determines which request handler should handle the request, and forwards the request to that handler. After the request handler completes its processing, the application returns a response to the adaptor. WOApplication also creates dynamic elements “on the fly” and manages adaptors, sessions, application resources, and components.

- WORequestHandler

  Manages the process of request handling and returns the completed response to the application. This is an abstract class.

Session Level

At the session level, the request-response loop looks like that shown in Figure 16.

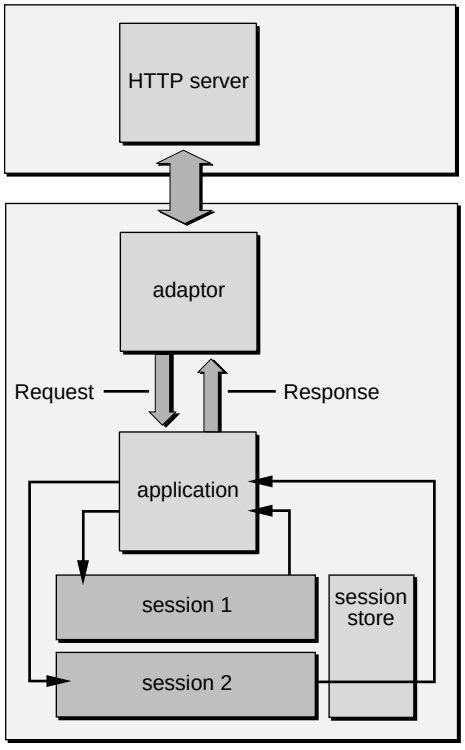

Figure 16. Request-Response Loop: Session Level

The objects dedicated to session management ensure that state with session-wide scope persists between cycles of the request-response loop. These objects always exist if your application uses component actions. They do not exist if you write your application entirely using direct actions, unless your code explicitly creates them.

Two classes are involved at this level:

- WOSession

  Encapsulates the state of a session. WOSession objects persist between the cycles of the request-response loop. WOSession objects store (and restore) the pages of a session, the values of session variables, and any other state that components want to persist throughout a session. The number of pages stored by the session object is dependent on the page-cache size set in WOApplication. Setting the page-cache size is described in the chapter “Managing State” (page 157). Each session object is identified by a unique session ID, which is either reflected in the URL or stored in a cookie.

- WOSessionStore

  Provides the strategy or mechanism through which WOSession objects are made persistent. A WOSessionStore object stores session objects in the server or in the page, and restores them upon request by the application.

When a user makes an initial component action request to a WebObjects application, or explicitly creates a session, the application creates a session object (WOSession). At the end of the request-response cycle, the application stores the state-bearing session object using the facilities of WOSessionStore. With each subsequent cycle of the component action request-response loop for that user, the application restores the state of the session at the beginning of the cycle and stores it again at the end of the cycle. To learn more about how to use WOSessionStore, see the chapter “Managing State” (page 157).

Request Level

The request-response cycle has three phases, the first for transferring user-entered data to the objects associated with the request page, the second for invoking an action method, and the third for generating and returning the response. Figure 17 shows how WebObjects requests are handled at the transaction level.

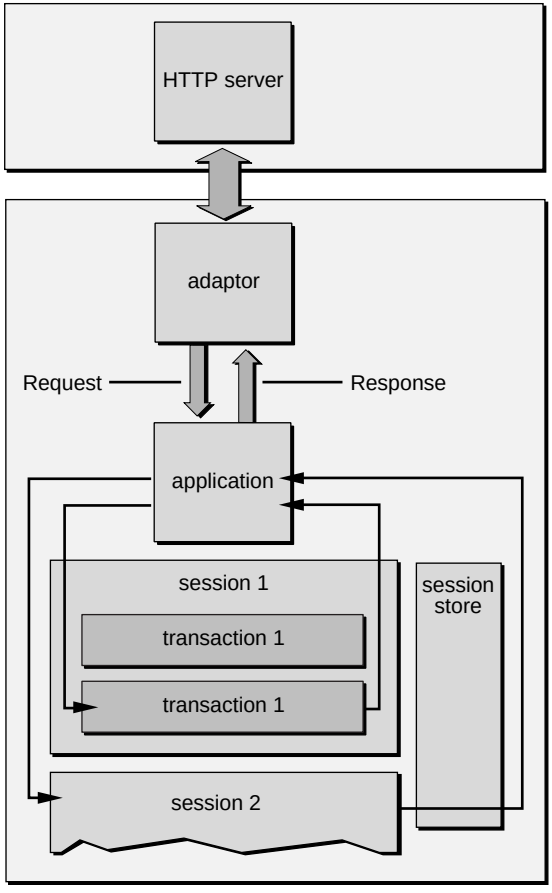

Figure 17. Request-Response Loop: Transaction Level

Three classes are involved at this level:

- WORequest

  Stores essential data about an HTTP request, such as header information, form values, HTTP version, host and page name, and session, context, and sender IDs.

- WOResponse

  Stores and allows the modification of HTTP response data, such as header information, status, and HTTP version. It also provides convenience methods for appending HTML and simple textual data to the content of the response (that is, the response page).

- WOContext

  Provides access to the objects involved in the current cycle, such as the current request, response, and session objects. It also stores the current component (either the current page or one of its subcomponents) to which the elements of the page make reference when they "push and pull" values through associations. See "How HTML Pages Are Generated" (page 125) for an explanation. The WOContext object acts as a "cursor," traversing the object graph during each phase of the component action request-response loop. The WOContext for a cycle is identified by a unique context ID.

You rarely need to work directly with WOContext, and only occasionally with WORequest and WOResponse. At the beginning of the request-response loop, the WOAdaptor and WORequestHandler objects create instances of these three classes and they are passed from object to object as needed. From these objects, the components, dynamic elements, and other objects involved in the cycle get essential information. See "How WebObjects Works—A Class Perspective" (page 107) for more on the mechanics of request handling.

Page Level

At the page level, objects of many classes (most of them private) work together to compose the HTML content of response pages (see Figure 18). Many of the same objects also set their variable values from data entered into request pages and respond to user actions.

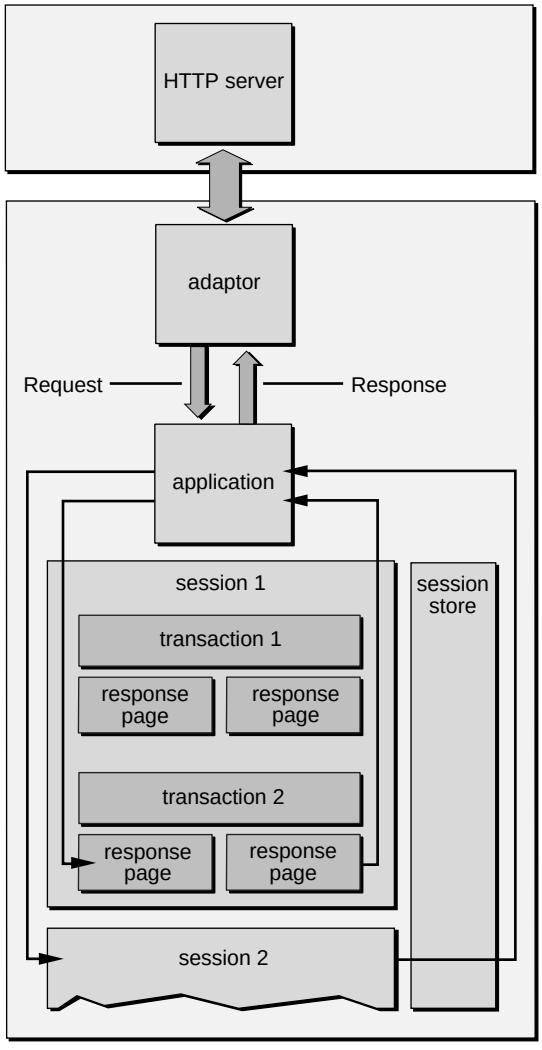

Figure 18. Request-Response Loop: Page Level

Two major branches of these objects descend from WOElement: WOComponent objects, which represent components, and WODynamicElement objects, which represent dynamic HTML elements on the page. For details on how this happens and for more on these classes, see “How HTML Pages Are Generated” (page 125).

Four classes are involved at this level. Of these four, most developers only interact with WOComponent.

- WOComponent

  Represents an integral, reusable page (or portion of a page) for display in a web browser.

- WOElement

  Declares the three component action request-handling methods: **takeValuesFromRequest:inContext:**, **invokeActionForRequest:inContext:**, and **appendToResponse:inContext:**. WOElement is an abstract class. Each node in an object graph, which represents the HTML elements of a component and their relationships, is an object that inherits from WOElement.

- WODynamicElement

  An abstract class for subclasses that generate particular dynamic elements.

- WOAssociation

  Knows how to find and set a value by reference to a key. WODynamicElement objects generally have WOAssociation instance variables.

Database Integration Level

Database integration is handled mainly by classes in the Enterprise Objects Framework (see Figure 19). The Enterprise Objects Framework converts operations on objects to database operations on records, thereby allowing your WebObjects application to interact with a database in an object-oriented manner.

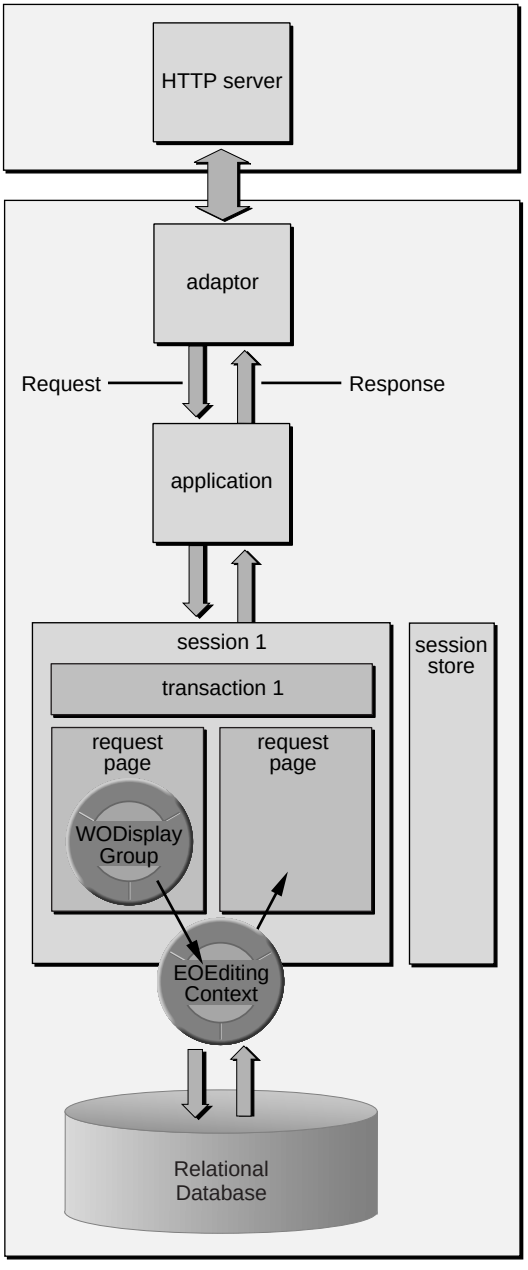

Figure 19. Request-Response Loop: Database Access

Two classes are involved at this level:

- WODisplayGroup

  Performs fetches, queries, creations, and deletions of records from one table in the database. WODisplayGroup is a sort of bridge between the dynamic elements on your page and the objects in the Enterprise Objects Framework.

- EOEditingContext (in Java, this is in the package **com.apple.yellow.eocontrol**)

  Manages a graph of objects fetched from a database. The objects represent tables, rows, and columns in the database.

When a WebObjects application accesses a database, one or more of the components in the application contain one or more WODisplayGroup objects. The session object provides access to an EOEditingContext object that is used, for example, when changed data is saved to the database. Each session uses an EOEditingContext to manage graphs of objects fetched from a database and to ensure that all parts of an application remain synchronized. For read-only applications, you can customize WOSession to return a per-application EOEditingContext.

For more information on how the WebObjects and Enterprise Objects classes interact, see the *Enterprise Objects Framework Developer's Guide*.

How WebObjects Works—A Class Perspective

You've now had a brief introduction to the classes used in WebObjects. This section describes the sequence of events that happen during a cycle of the request-response loop—how the application starts up, what happens when it receives an HTTP request, and how it processes that request.

Starting the Request-Response Loop

A WebObjects application's entry point is the same as that of any C program: the **main** function. In a WebObjects application, **main** is usually very short. Its job is to create and run the application.

The **main** function typically consists of a single line of code that calls the **WOApplicationMain** function, passing the name of your application's principal class and any arguments you may have passed to your application. **WOApplicationMain** processes any arguments, then creates an instance of your principal class (which should be a subclass of WOApplication, and is named **Application** by default). In your application class' **init** method (or constructor) the application creates and stores one or more adaptors in an instance variable. These adaptors—all instances of a WOAdaptor subclass—handle communication between an HTTP server and the WOApplication object. The application first parses user defaults dictionary for the specified adaptors (with necessary arguments); if none are specified, it creates a suitable default adaptor.

Also during application initialization, the application object creates its pool of WORequestHandlers. By default, applications have three request handlers: one for component actions (WOComponentRequestHandler), one for direct actions (WODirectActionRequestHandler), and one for resource requests (WOResourceRequestHandler). All three of these are private subclasses of WORequestHandler. If your application has additional request handlers, they should be registered with the application at application initialization time.

After your application class has been initialized, **WOApplicationMain** sends it a **run** message, initiating the request-response loop. When **run** is invoked, the application sends **registerForEvents** to each of its adaptors to tell them to begin receiving events. Then the application begins running in its run loop.

The autorelease pool is released and recreated immediately before the **run** message is sent. Releasing the autorelease pool at this point releases any temporary variables created during initialization of the application class. Creating a new autorelease pool before sending **run** ensures that all variables created while running the application will be released eventually. The last message releases the autorelease pool, which in turn releases any object that has been added to the pool since the application started running.

In the rest of this section, we look at what happens during one complete cycle of the request-response loop.

Determining the Request Type

The first step in the request-response cycle is to determine which request handler should handle the request. A cycle of the request-response loop begins when the WOAdaptor receives an incoming HTTP request. The adaptor object packages this request in a WORequest object and forwards this object to the application object in a **dispatchRequest:** message.
The application object determines which WORequestHandler should handle the request with **handlerForRequest:**. (The request handler key in the request URL specifies which WORequestHandler should be used.)

If the request is the first one for a given user session, the request URL looks like the URL shown in Figure 20.

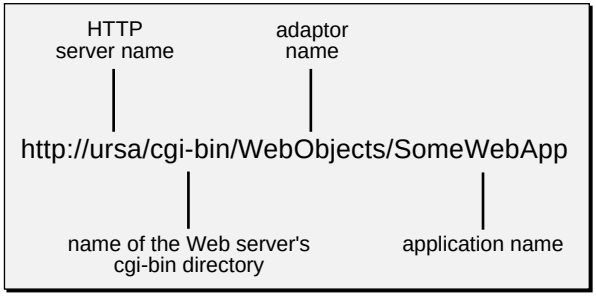

Figure 20. URL to Access a WebObjects Application

This URL does not have a request handler key. In this case, WOApplication uses the default request handler (returned by its **defaultRequestHandler** method). Unless you override the default using **setDefaultRequestHandler:**, the default request handler is WOComponentRequestHandler.

After the initial request, subsequent URLs look like the one shown in Figure 21.

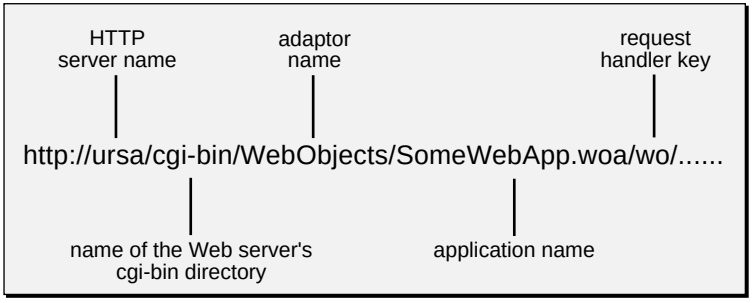

Figure 21. WebObjects URL in Existing Session

This URL contains a request handler key immediately following the application name. WOApplication maintains a dictionary of WORequestHandlers. It uses the request handler key to look up which request handler to use. By default, "wo" is the request handler key specifying component actions, and "wa" specifies direct actions (which are handled by the WODirectActionRequestHandler object).

Once the application has determined which request handler to use, it sends that handler a **handleRequest:** message. From this point forward, what happens during the request-response loop is highly dependent on which type of request is being processed: a component action request, or a direct action request. The rest of this section looks at each type of request handler in detail.

Handling Component Action Requests

The first phase of the component action request-response loop (see Figure 22) synchronizes the state of the request component with the HTML page as submitted by the user. In this phase, the appropriate dynamic elements extract the values that users enter and the choices they make in the request page and assign them to declared variables.

For example, if the user clicked a checkbox, the dynamic element that represents that checkbox must be set to the "checked" state. In other words, the **checked** attribute of the appropriate WOCheckbox dynamic element must be set to YES.

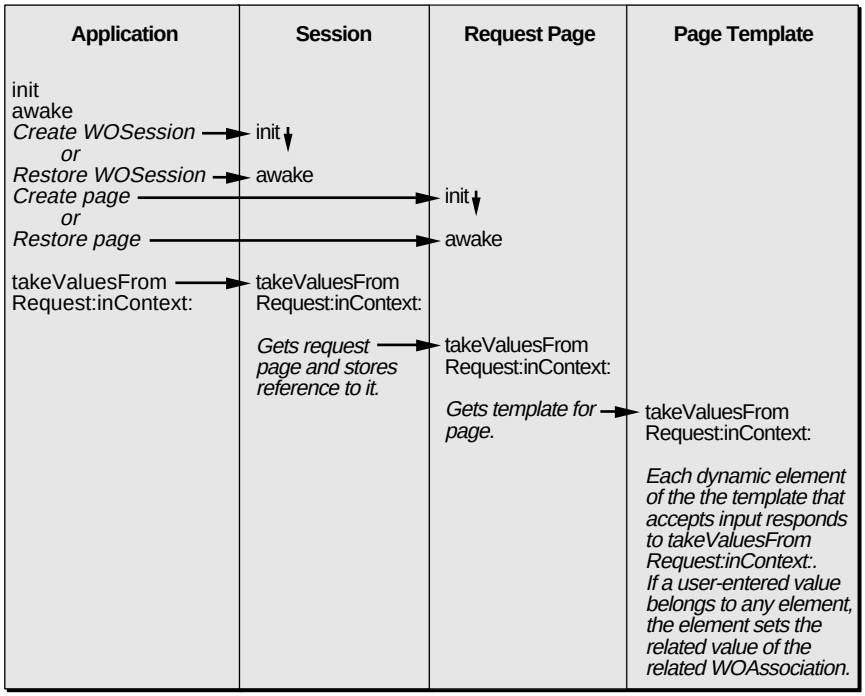

Figure 22. Taking Values from the Request

When the component action request handler receives the **handleRequest:** message from the application, it does the following:

1. It creates the WOResponse and WOContext objects that will be needed.
2. It invokes the application's **awake** method.
3. It determines which session and which request page are associated with the request, as described next.

Accessing the Session

The component action request handler determines whether to create a new session or access an existing session by searching the request URL for a session ID. If the request is the first one for the session, the request URL looks like the URL shown in Figure 20. This URL does not contain a session ID, so the request handler creates a new session by performing the following steps:

1. It sends the application a **createSessionForRequest:** message.
2. As part of the **createSessionForRequest:** method, the application sends the **init** message or the constructor message to the WOSession class to create a new session object.
3. The application sends the **awake** message to the session object.

If the request is part of an existing session, the request URL looks like the one shown in Figure 23.

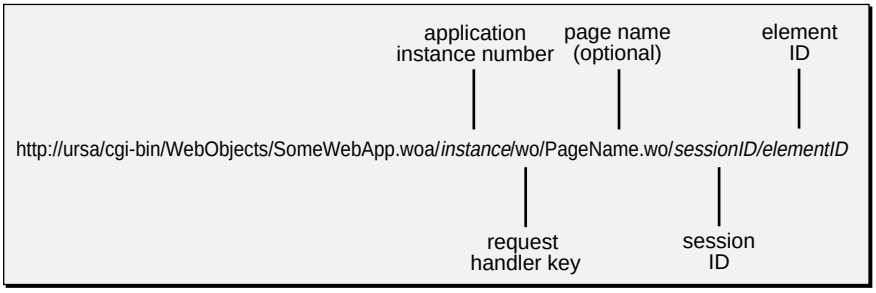

Figure 23. Component Action Request URL

This URL contains all of the information necessary to restore the state of the existing session. The session ID comes right after the page name in the URL (the page name is optional; if it isn't present in the URL, the session ID comes right after the request handler key). Because sessions are designed to protect the data of one user's transactions from that of another, session IDs must not be easily predicted or faked. To this end, WebObjects uses randomly generated 17-character sequence of letters and numbers. (You can also override WOSession's **sessionID** method and implement another security scheme if you'd like.)

The application keeps active sessions in the WOSessionStore object. The application object uses the session ID to retrieve the appropriate session from the session store (see Figure 24). The appropriate session object is then sent the **awake** message to prepare it for the request.

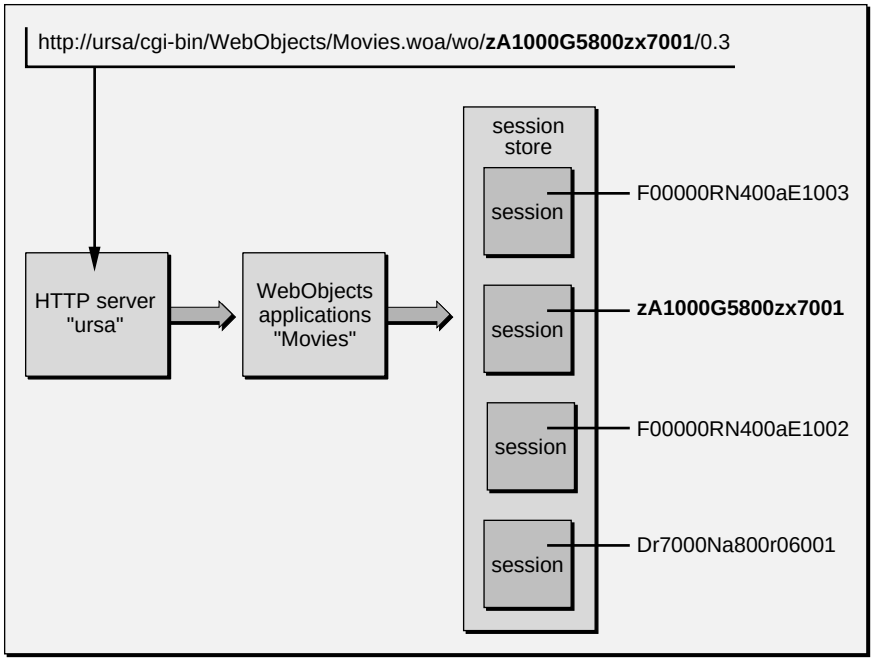

Figure 24. Associating a Request with a Session Object

Creating or Restoring the Request Page

After the session receives the **awake** message, the next step is to find the *request page*. Each request received by a WebObjects application is associated with one of the application's pages—the request page. The request page is usually the response page from the last request. (The response page shows the result, or output, of the request; note, however, that backtracking may alter this.)

If the user has just begun a new session (that is, if the request URL looks like the one shown in Figure 20), the user has not requested a specific page. Therefore, the application object creates a new instance of the WOComponent class for the page named "Main." The application object performs the following steps to create a component:

1. It looks in the runtime system for a WOComponent subclass that has the same name as the request page (in this case, "Main"). If it finds such a class with the same name, it creates an instance of that class.

2. If the application object fails to find a class in the run-time system, it looks for a scripted component with the name of the request page—first looking in the application, and then searching any linked frameworks. When it finds the **.wo** directory, it creates a component object using a unique WOComponent subclass for the scripted component and makes the scripted code the class implementation.

3. It invokes the WOComponent subclass's **init** method or constructor.

4. It invokes the WOComponent subclass's **awake** method to prepare it for the request.

If the request is made from an established session, the application object attempts to retrieve the request page from its cache. By default, an application caches component (or page) instances once they're created, primarily to facilitate backtracking: when users backtrack, they're revisiting pages restored by the application. The request URL contains the information needed to retrieve the page from the cache (see Figure 23). This information includes the page name and a context ID.

The component may not be in the cache for one of three reasons:

- The page-caching feature is turned off.
- The request is the first for that page during the session.
- The user has backtracked beyond the page cache limit.

If the component is not in the cache, the application object creates the component using the procedure described above. If the component is in the cache, it sends the component the **awake** message.

Note that to retrieve the page from the cache, a context ID is required. The context ID identifies a page *as it existed at the end of a particular request-response loop*. Why is the context ID necessary? Imagine you're accessing a WebObjects application that lets you subscribe to various publications.

You navigate from the site's home page to the order page, where you select a publication, and then you go to the customer information page and fill in your address. After submitting this information, you navigate back to the home page. Next, you decide to enter a subscription for a friend. You follow the process a second time, selecting a different publication and entering your friend's address.

At this point, within a single session with the subscriptions application, you've accessed the same pages twice, entering different information each time. Let's say that you now realize that you made a mistake in your own address, so you backtrack to that page, change the address, and resubmit the information. It's important that the new address information is submitted to the customer information page as it existed during the first order so that the revised information can be associated with the right publication order.

WebObjects associates a different context ID (again, a randomly generated integer—to maintain security) with each request-response loop cycle. A request to a session includes both the name of request page and a context ID so the session object can locate, from its cache of page instances, the appropriate one to handle the request.

Taking Input Values From a Request

At this point, the application, session, and component objects have been created (if necessary) and awakened so that they are ready for the request. The next step is to extract user-entered values and assign them to variables. The application checks the WORequest object to see if it contains any user-entered values. If so, the following basic sequence of events takes place:

1. The application object sends **takeValuesFromRequest:inContext:** (in Java, **takeValuesFromRequest**) to itself; its implementation simply invokes the session object's **takeValuesFromRequest:inContext:** method.

2. The session sends the **takeValuesFromRequest:inContext:** message to the request component.

3. The component, in its implementation of **takeValuesFromRequest:inContext:**, gets its template and forwards the message to the template's root object. A *template* is an object graph that represents the static HTML elements, dynamic HTML elements, and subcomponents that together compose the page associated with a component instance.

4. All dynamic elements in the page template and in the templates of subcomponents receive the **takeValuesFromRequest:inContext:** message. If one of these elements "owns" a user-entered value,

it responds to the message by storing the value in the appropriate variable defined in the request component's declarations file.

For more on how components are associated with templates, and on how HTML elements participate in request-handling, see "How HTML Pages Are Generated" (page 125).

This step takes place *only if the request has input values*. If the request does not have input values, **takeValuesFromRequest:inContext:** is not performed.

Invoking an Action

In the second phase of the request-response loop (see Figure 25), the application first determines which dynamic element the user has clicked (or otherwise activated) and then has that element trigger the appropriate action method in the request component. This method returns the *response page*—the component responsible for generating an HTTP response. If the user has not triggered an action, the request component is used as the response component.

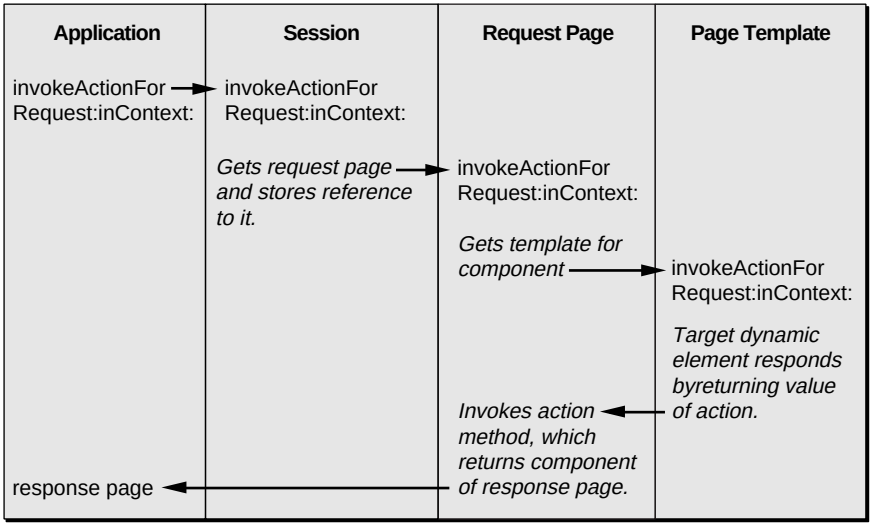

Figure 25. Invoking an Action

Here is the basic sequence of events for invoking an action:

1. The application object sends **invokeActionForRequest:inContext:** (in Java, **invokeAction**) to itself; its implementation simply invokes the session object's **invokeActionForRequest:inContext:** method.

2. The session sends **invokeActionForRequest:inContext:** to the request component.

3. The component, in its implementation of **invokeActionForRequest:inContext:**, gets the template of the component and forwards the message to the template's root object.

4. Suitable dynamic elements in the request-page template and in subcomponent templates handle the **invokeActionForRequest:inContext:** message and invoke the appropriate action method in the request component. This action method returns the response page.

   To be suitable, an element must be able to respond to user actions (a WOSubmitButton or a WOActiveImage, for example). Each of these elements evaluates the invoked action to determine if it "owns" it (that is, its **elementID** matches the request's **senderID**).

For more on how components are associated with templates and on how HTML elements participate in request-handling, see "How HTML Pages Are Generated" (page 125).

Generating the Response

In the final phase of request-response loop (see Figure 26), the response page generates an HTTP response. Generally, the response contains a dynamically generated HTML page. Each element (static and dynamic) that makes up the response page appends its HTML code to the total stream of HTML code that will be interpreted by the client browser.

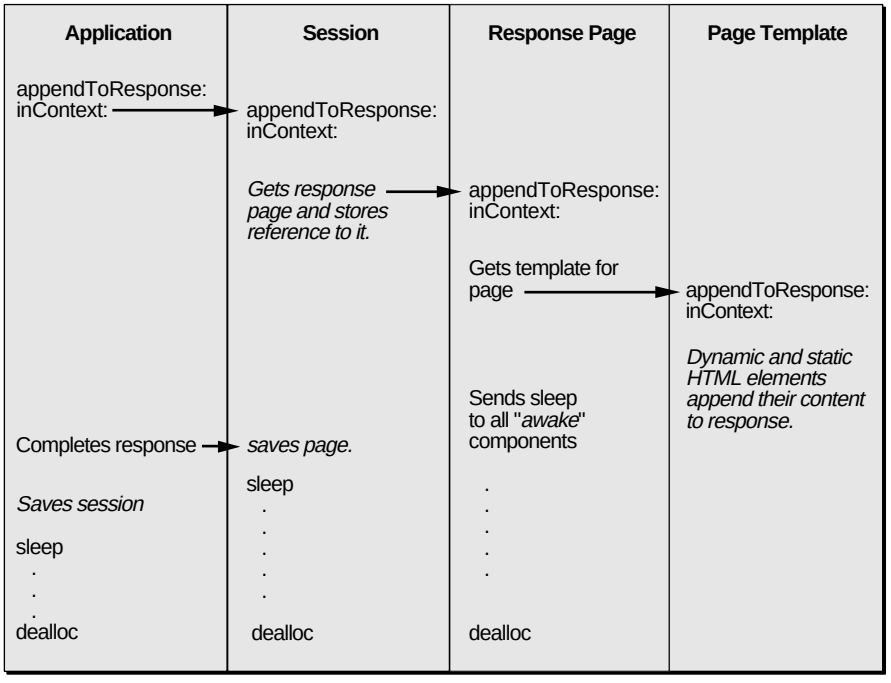

Figure 26. Generating the Response

Here is the basic sequence of events for generating a response:

1. The application object stores the response component indicated by the action method's return value. (This action method was invoked during phase 2.)

2. If the response component is different from the request component, application sends the **awake** message to the response component.

3. The application object sends **appendToResponse:inContext:** to itself; its implementation simply invokes the session object's **appendToResponse:inContext:** method.

4. The session pushes the response component onto the WOContext stack and sends the response component the **appendToResponse:inContext:** message.

5. The response component, in its implementation of **appendToResponse:inContext:**, gets the template for the component and sends **appendToResponse:inContext:** to the template's root object.

6. All static and dynamic HTML elements in the response-page template, and in subcomponent templates, receive the **appendToResponse:inContext:** message. In it, they append to the content of the response the HTML code that represents them. For dynamic elements, this code includes the values assigned to variables.

7. When control returns to the session object, the session object asks the WOStatisticsStore to record statistics about the response. WOStatisticsStore sends the session a **descriptionForResponse:inContext:** message. The session, in turn, sends the response component **descriptionForResponse:inContext:** message. By default, this method returns the response component's name.

Request Post-Processing

After the response has been generated, but before returning the response to the adaptor, the component action request handler concludes request handling by doing the following:

1. It causes the **sleep** method—the counterpart of **awake**—to be invoked in all components involved the cycle (request, response, and subcomponents). As described in the chapter "Managing State" (page 157), in the **sleep** method objects can release resources that don't have to be saved between cycles.

2. It requests the session object to save the response page in the page cache.

3. It invokes the session object's **sleep** method.

4. It saves the session object in the session store.

5. It invokes its own **sleep** method.

When an Objective-C object is about to be destroyed, its **dealloc** method is invoked at an undefined point in time after a cycle (indicated by the vertical

ellipses in Figure 26). In the **dealloc** method, the object releases any retained instance variables. In WebScript, this usually happens implicitly; you therefore usually don't need to implement the **dealloc** method in any objects you write. In Java, objects have automatic garbage collection, so this deallocation step is unnecessary.

Note: WOApplication provides two Java methods—**garbageCollectionPeriod()** and **setGarbageCollectionPeriod()**—that allow you to get and set the amount of time between garbage collections.

Handling Direct Action Requests

As you saw in the previous section, component action requests are handled in three basic phases: taking input values from the request, invoking the action, and generating the response. The WOApplication, WOSession, and WOComponent objects are involved in each phase. Direct actions have a simpler request-response loop cycle.

When the direct action request handler receives the **handleRequest:** message from the application, its first step is to determine what action should be performed and by what object. It does so by looking at the request URL. The URL for a direct action request looks like the one shown in Figure 27.

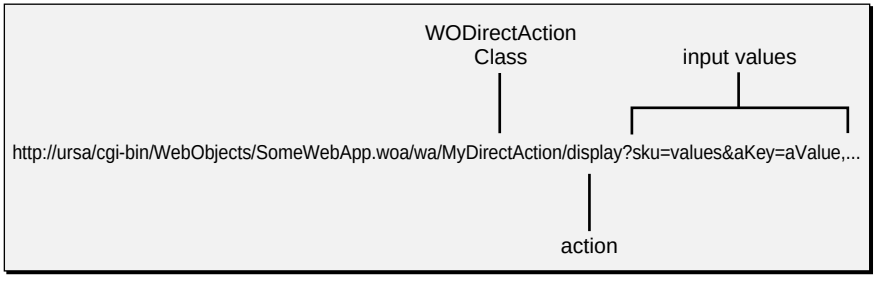

Figure 27. Direct Action Request URL

The WODirectAction class field is optional. If the WODirectAction class isn't specified in the URL, a subclass named DirectAction is assumed.

If the request is the first one for a given user session, the request URL looks like the URL shown in Figure 20 (on page 109). This URL contains neither a WODirectAction class name nor an action name. In this case, the direct action request handler assumes that the action to be performed is the **defaultAction**

method in the class DirectAction. Remember that the direct action request handler does not process the first request in a session unless you send the **setDefaultRequestHandler:** to the WOApplication and specify the WODirectActionRequestHandler.

Since URLs can contain class names, the WODirectActionRequestHandler always checks to make sure that the class specified is a subclass of WODirectAction.

Invoking the Action

Once the direct action request handler has determined the action to be performed and which object should perform the action, the handler does the following:

1. It creates an instance of the appropriate WODirectAction subclass, which in turn creates a WOContext object.

2. It sends that instance a **performActionNamed:** message.

3. In its implementation of **performActionNamed:**, the WODirectAction object looks for a method named *action***Action**. For example, if the action specified in the URL is “display”, WODirectAction looks for a method named **displayAction**. It then invokes that method.

4. If necessary, the action method takes input values from the request and assigns them either to its own local variables or to the object’s instance variables.

   With direct actions, all input values are in the WORequest object as form values. If the form method was a “GET” then the URL contains the form values.

5. When it is finished processing, the action method invoked in the previous step returns either a WOResponse or WOComponent object. The WODirectAction object returns that object to the direct action request handler.

Generating the Response

Upon receiving a return value from **performActionNamed:**, the direct action request handler sends the returned object a **generateResponse** message.

Typically, the returned object is a WOComponent. When a WOComponent receives **generateResponse** message, the following sequence of events take place:

1. The component gets the template for itself and sends **appendToResponse:inContext:** to the template's root object.

2. All static and dynamic HTML elements in the component's template, and in subcomponent templates, receive the **appendToResponse:inContext:** message. In it, they append to the content of the response the HTML code that represents them. For dynamic elements, this code includes the values assigned to variables.

The object returned by **performActionNamed:** can actually be any object that conforms to the WOActionResults procotol (or Java interface). WOActionResults defines the **generateResponse** method. In the WebObjects framework, two classes conform to WOActionResults: WOResponse and WOComponent. (When a WOResponse object receives **generateResponse**, it simply returns itself.)

Request Post-Processing

After the response has been generated, but before returning the response to the application, the direct action request handler concludes request handling by releasing the WODirectAction instance it created, or in Java, marking it for garbage collection. If the WODirectAction created a component and returned that as the response, that component is released along with the WODirectAction. Thus, in the default implementation, WODirectActions do not maintain any session state between cycles of the request-response loop.

Component Actions vs. Direct Actions

Figure 28 shows the sequence of events in processing a component action request and compares it to the sequence of events for processing the direct action. Note that in both component actions and direct actions, the bulk of the time is spent in the generate response phase, in which the component performs **appendToResponse:inContext:** and sends each of its dynamic elements **appendToResponse:inContext:**. This step is the same in

component actions and direct actions.

Component Action	Direct Action
The adaptor creates a WORequest object and passes it to the application.	The adaptor creates a WORequest object and passes it to the application.
The application determines that the WOComponentRequestHandler should handle the request.	The application determines that the WODirectActionRequestHandler should handle the request.
The application, session, and the request component are created, if necessary, and sent the **awake** message.	Application **awake** is called.
The **takeValuesFromRequest:inContext:** message is propagated from the application to the session to the request component to each dynamic element in the request component (if the request has input values).	WODirectActionRequestHandler parses the URL and instantiates the WODirectAction class.
The **invokeActionForRequest:inContext:** message is propagated from the application to the session to the request component to each dynamic element in the request component, resulting in the appropriate action method in the component being invoked.	WODirectActionRequestHandler sends the message **performActionNamed:** to the WODirectAction, resulting in the appropriate action being invoked. If there are any input values, WODirectAction uses **takeFormValues...** methods to extract them from the WORequest.
The **action** method creates and returns a response component or response.	The **action** method creates and returns a response component or response.
The application awakens the response component. The **appendToResponse:inContext:** message is propagated from the application to the session to the response component to each dynamic element in the response component.	The object returned by the action method is sent a **generateResponse** method to guarantee that the object returned is a WOResponse. If the action returns a WOComponent, WOComponent's **generateResponse** invokes **appendToResponse:inContext:**, which sends each dynamic element in the component an **appendToResponse:inContext:** message as well.
The application forwards the WOResponse to the adaptor.	The application forwards the WOResponse to the adaptor.
The application, session, and all of the components are sent the **sleep** message.	The WODirectAction is release or marked for garbage collection. Application **sleep** is called.
The component is saved in the session so it can handle any subsequent requests.	If the returned component contained any component actions, the component is saved in the session so it can handle any subsequent requests.

Figure 28. Comparison of Component Action and Direct Action request processing.

The major differences between component actions and direct actions are:

- Component actions require state, primarily so that they can determine which component should perform the action.

  Direct actions are stateless actions. They do not require any state to be preserved between requests. For this reason, direct actions do not create session objects by default. WODirectAction defines a method that does create a session, so direct actions can create a session and store state if necessary.

- Component actions extract input values from the request without requiring you to write any code to do so.

  Direct actions do not automatically extract input values from the request. If there are input values, the HTML element places them in the HTTP request, and the action method must explicitly request them from the WORequest object.

- Component actions have dynamic, unpredictable URLs because the URL contains the session ID and context ID, which are unique to each session.

  Direct actions have static, predictable URLs. Regardless of which session is performing the action, the URL for the action always refers to a specific action—one that was determined when the current page was generated (not when the request is handled).

  Because their URLs are static and because they do not require state, direct action requests can be bookmarked by your application's users and can be revisited at any time.

How HTML Pages Are Generated

So how exactly are request-handling messages propagated from a component to its HTML elements? To answer this, we must understand the relationship between a component and an HTML element.

Both components and HTML elements (static and dynamic) share a common ancestor, WOElement. WOElement declares, but does not implement, the three component action request-handling messages: **takeValuesFromRequest:inContext:**, **invokeActionForRequest:inContext:**, and **appendToResponse:inContext:**. This common inheritance, of course, makes it possible for both components and HTML elements to participate in request handling. But there the inherited similarities end. Although components can generate HTML content, this capability is not an essential characteristic, as it is with objects on the other branch of the inheritance tree.

Component Templates

The first step to generating a component's HTML page is to create a *template* for the component. This template is not the same as the HTML template discussed in the chapter "What Is a WebObjects Application?" (page 17). In this context, a template is a graph of WOElement and WOComponent objects created by parsing and integrating the component's **.html** and **.wod** files (see Figure 29). The network of references corresponds to locations on the page and to parent-child relationships; for instance, a WOForm element would probably have WOTextField and WOSubmitButton children.

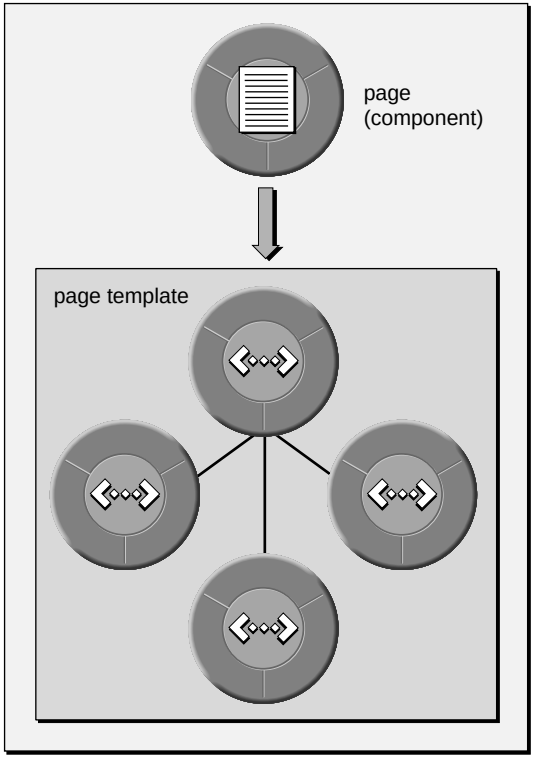

Figure 29. An Object Graph for a Page's Template

The template is created at run-time when the component is first requested. The template is part of a larger *component definition*, which also includes information that allows instances of this component to share resources. Instances carry only the instance-variable values that are distinctive to them; the rest is stored in the component definition. You can, if you wish, enable caching of component definitions so that the component is parsed only once during an application's lifetime. To do so, send the application object a **setCachingEnabled:** message in its initialization method. Otherwise, the component is parsed each time it changes.

For each request-handling message, WOComponent's default behavior is to forward the message to the objects in its template. To do so, it first retrieves the template from the component definition. The component definition returns the WOElement object at the root of the object graph. This root object, in turn, forwards the message to each of its child elements; if they have any

children, these elements send the message to them. Thus, each element has, if appropriate, an opportunity to extract user data from the request, to invoke an action in the component, and to append its HTML representation to the response.

Each HTML element on a template has an element ID to identify it within the object graph. This element ID is dynamically determined as each of the three phases (takeValuesFromRequest, invokeActionForRequest, and appendToResponse) is processed. You can request the current element ID from the WOContext object.

Associations and the Current Component

A dynamic HTML element, such as a text field or a pop-up button, differs from a static HTML element, such as a heading, in that its attributes can change over a cycle of the request-response loop. These attributes can include values that determine behavior or appearance (a "disabled" attribute, for instance), values that users enter into a field, values that are returned from a method, and actions to invoke when users click or otherwise activate the element. Each dynamic element stores its attributes as instance variables of type WOAssociation. WOAssociation objects know how to obtain and set the value they represent. They generally do this using key-value coding.

Keys (including actions) are WOAssociations defined for each dynamic element. The values for these keys are constants assigned in the **.wod** file, or they derive from bindings to variables, to methods, or to entities retrieved through a WODisplayGroup (for applications that access a database).

WOAssociation objects refer to the *current component* for the initial value of this sequence. They get this object from the cycle's WOContext object. Often the current component is the request or response page of the cycle, but it can be a reusable component embedded in a page, or even a component incorporated by one of those subcomponents. See "Subcomponents and Component References" (page 129) for more on this.

A dynamic element uses its WOAssociations to "pull" values from the request (that is, set its values to what the user specifies) or to "push" its values onto the response page. When a dynamic element that can respond to user actions (such as WOSubmitButton) requests the value of its "action" association, the appropriate action method is invoked and the response page is returned.

The exchange of data through an association that binds an attribute of a parent component to an attribute of a child component can be two-way. This two-way binding allows the synchronization of state between the two components. Consider this declaration in **Main.wod**:

```
START:Calendar {
  selectedDate = startDate;
  callBack = "mainPage";
};
```

In this example, Main is the parent component and Calendar is the child component. The **startDate** variable (or method) belongs to the parent component while **selectedDate** is a variable of the child component. A change in the parent component instance variable is automatically communicated through the association to the child variable. Conversely, a change in value in the child component variable is communicated to the parent variable. Component synchronization occurs at the beginning and end of each of the three request-handling phases of a component action request response loop (**takeValuesFromRequest:inContext:**, **invokeActionForRequest:inContext:**, and **appendToResponse:inContext:**). Synchronization is performed through the accessor methods of both components.

Note: The only request-response loop phase that component actions and direct actions have in common is the **appendToResponse:inContext:** phase. When you are using direct actions, component synchronization occurs at the beginning and end of this phase, which means that the direct action method is performed before the components are synchronized.

This aspect of synchronization has implications for developers. Because WebObjects invokes explicitly implemented accessor methods many times during the same component action request-response loop, your accessor methods must have no side effects. Instead, they should simply set a variable's value or return a value. And if they return a value, there should be some way for WebObjects to set the value.

This rule applies also when the binding involves a parent or a child component's method instead of an instance variable. To illustrate this, assume that **startDate** is a method of the Main component instead of an instance variable. Even in this case, WebObjects attempts to synchronize **startDate** with the **selectedDate** value. In other words, WebObjects attempts to invoke

a **setStartDate:** method and raises an exception if such a method does not exist.

See the chapter “Creating Reusable Components” (page 133) for more on state synchronization between child and parent component.

Subcomponents and Component References

A “node” in a template’s object graph can represent a *subcomponent* (also called a reusable component) as well as a dynamic or static HTML element. A dynamic element called a *component reference* represents all occurrences of the subcomponent in the parent component. At run-time, the component reference binds itself to the separate instances. Figure 30 is an example of an object graph for a page with a subcomponent.

A subcomponent can fire actions against its parent component (using **performParentAction:**), and if the parent’s state changes, its state is synchronized accordingly. In other words, its state is updated to reflect changes according to its bindings with the parent.

An element ID is assigned to each instance of a subcomponent. When a request-handling message traverses an object graph and reaches the component reference, it resolves references to its instances according to the element ID of each instance.

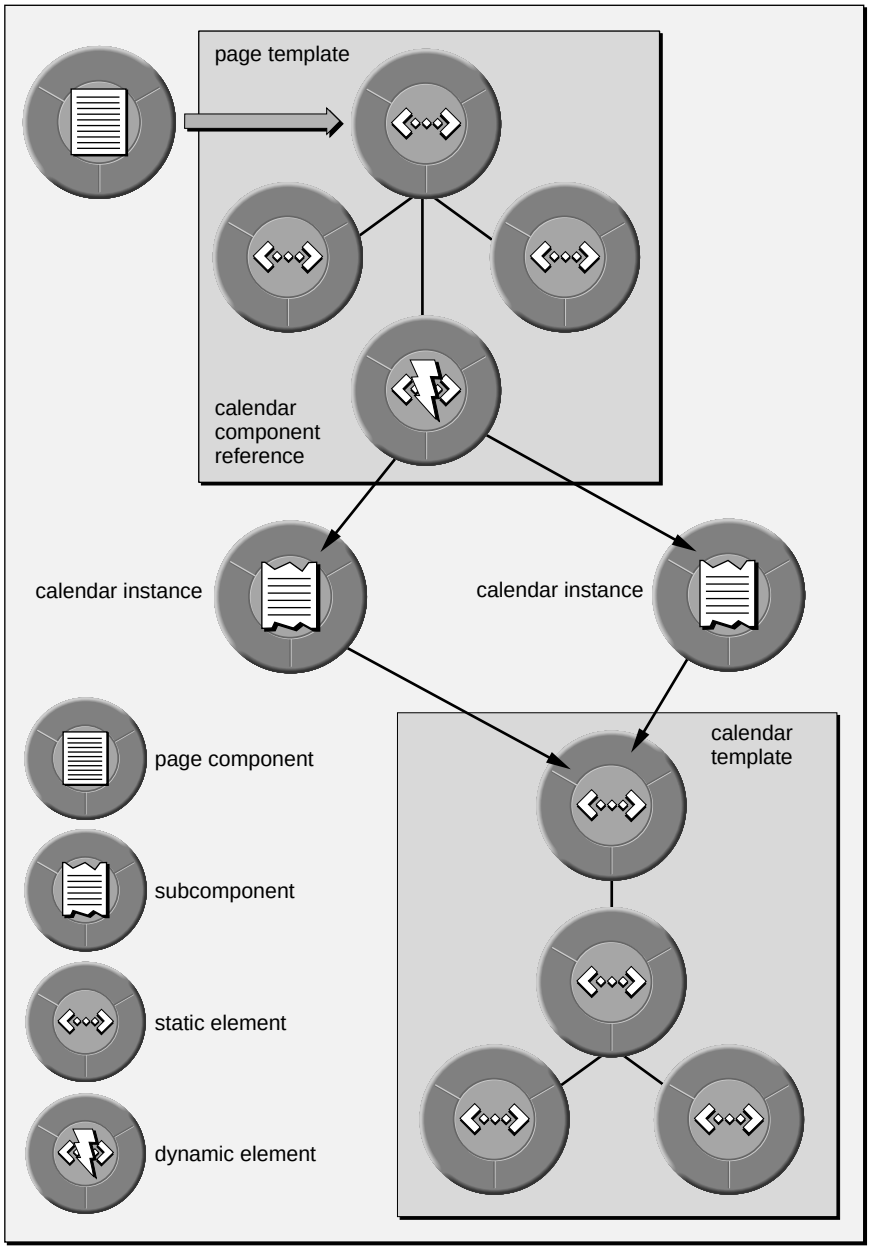

Figure 30. An Object Graph for a Page With a Subcomponent

Part II

Special Tasks

Chapter 6

Creating Reusable Components

In the simplest applications, each component corresponds to an HTML page, and no two applications share components. However, one of the strengths of the WebObjects architecture is its support of reusable components: components that, once defined, can be used within multiple applications, multiple pages of the same application, or even multiple sections of the same page.

This chapter describes reusable components and shows you how to take advantage of them in your applications. It begins by illustrating the benefits of reusable components. It then describes how to design components for reuse, how reusable components can communicate with the parent component, and how state is synchronized between parent and child components. Finally, it provides some design tips for you to consider when designing your own reusable components.

Benefits of Reusable Components

Reusable components benefit you in two fundamental ways:

- They help you centralize application resources.
- They simplify interfaces to packages of complex, possibly parameterized, logic and display.

The following sections explain these concepts in detail.

Centralizing Application Resources

One of the challenges of maintaining a web-based application is the sheer number of pages that must be created and maintained. Even a modest application can contain scores of HTML pages. Although some pages must be crafted individually for each application, many (for example, a page that gathers customer information) could be identical across applications. Even pages that aren't identical across applications can share at least some portions (header, footer, navigation

bars, and so on) with pages in other applications. With reusable components, you can factor out a portion of a page (or a complete page) that's used throughout one or more applications, define it once, and then use it wherever you want, simply by referring to it by name. This is a simple but powerful concept, as the following example illustrates.

Suppose you want to display a navigational control like the one shown in Figure 31 at the bottom of each page of your application.

Figure 31. A Navigational Control

The HTML code for this control is:

```
<HTML>
<HEAD>
  <TITLE>World Wide Web Wisdom, Inc.</TITLE>
</HEAD>

<BODY>
Please come visit us again!

<!-- start of navigation control -->
<CENTER>
<TABLE BORDER = 7 CELLPADDING = 0 CELLSPACING = 5>
  <TR ALIGN = center>
      <TH COLSPAN = 4> World Wide Web Wisdom, Inc.</TH>
  </TR>
  <TR ALIGN = center>
      <TD><A HREF = "http://www.wwww.com/home.html"> Home
<a></TD>
      <TD><A HREF = "http://www.wwww.com/sales.html"> Sales
<a></TD>
      <TD><A HREF = "http://www.wwww.com/service.html"> Service
<a></TD>
      <TD><A HREF = "http://www.wwww.com/search.html"> Search
<a></TD>
  </TR>
</TABLE>
</CENTER>
<!-- end of navigation control -->

</BODY>
</HTML>
```

Thirteen lines of HTML code define the HTML table that constitutes the navigational control. You could copy these lines into each of the application's pages or use a graphical HTML editor to assemble the table wherever you need one. But as application size increases, these approaches becomes less practical. And obviously, when a decision is made to replace the navigational table with an active image, you must update this code in each page. Duplicating HTML code across pages is a recipe for irritation and long hours of tedium.

With a reusable component, you could define the same page like this:

```
<HTML>
<HEAD>
  <TITLE>World Wide Web Wisdom, Inc.</TITLE>
</HEAD>

<BODY>
Please come visit us again!

<!-- start of navigation control -->
<WEBOBJECT NAME="NAVCONTROL"></WEBOBJECT>
<!-- end of navigation control -->

</BODY>
</HTML>
```

The thirteen lines are reduced to one, which positions the WebObject named NAVCONTROL. The declarations file for this page binds the WebObject named NAVCONTROL to the component named NavigationControl:

```
NAVCONTROL: NavigationControl {};
```

All of the application's pages would have entries identical to these in their template and declarations files.

NavigationControl is a component that's defined once, for the use of all of the application's pages. Its definition is found in the directory **NavigationControl.wo** in the file **NavigationControl.html** and contains the HTML for the table:

```
<CENTER>
<TABLE BORDER = 7 CELLPADDING = 0 CELLSPACING = 5>
<TR ALIGN = center>
  <TH COLSPAN = 4> World Wide Web Wisdom, Inc.</TH>
</TR>
<TR ALIGN = center>
  <TD><A HREF = "http://www.wwww.com/home.html"> Home <a></TD>
  <TD><A HREF = "http://www.wwww.com/sales.html"> Sales <a></TD>
  <TD><A HREF = "http://www.wwww.com/service.html"> Service
<a></TD>
  <TD><A HREF = "http://www.wwww.com/search.html"> Search
<a></TD>
</TR>
</TABLE>
</CENTER>
```

Since NavigationControl defines a group of static elements, no declaration or code file is needed. However, a reusable component could just as well be associated with complex, dynamically determined behavior, as defined in an associated code file.

Now, to change the navigational control on all of the pages in this application, you simply change the NavigationControl component. What's more, since reusable components can be shared by multiple applications, the World Wide Web Wisdom company could change the look of the navigational controls in all of its applications by changing this one component.

If your application's pages are highly structured, reusable components could be the prevailing feature of each page:

```
<HTML>
<HEAD>
  <TITLE>World Wide Web Wisdom, Inc.</TITLE>
</HEAD>

<BODY>

<WEBOBJECT NAME="HEADER"></WEBOBJECT>
<WEBOBJECT NAME="PRODUCTDESCRIPTION"></WEBOBJECT>
<WEBOBJECT NAME="NAVCONTROL"></WEBOBJECT>
<WEBOBJECT NAME="FOOTER"></WEBOBJECT>

</BODY>
</HTML>
```

The corresponding declarations file might look like this:

```
HEADER: CorporateHeader {};
PRODUCTDESCRIPTION: ProductTable {productCode = "WWWW0314"};
NAVCONTROL: NavigationControl {};
FOOTER: Footer {type = "catalogFooter"};
```

Notice that some of these components above take arguments—that is, they are parameterized. For example, the ProductTable component's **productCode** attribute is set to a particular product identifier, presumably to display a description of that particular product. The combination of reusability and customizability is particularly powerful, as you'll see in the next section.

Simplifying Interfaces

Another benefit of reusable components is that they let you work at a higher level of abstraction than would be possible by working directly with HTML code or with WebObjects' dynamic elements. You (or someone else) can create a component that encapsulates a solution to a possibly complicated programming problem, and then reuse that solution again and again without having to be concerned with the details of its implementation. Examples of this kind of component include:

- A menu that posts different actions depending on the user's choice
- A calendar that lets a user specify start and end dates
- A table view that displays records returned by a database query

To illustrate this feature, consider a simple reusable component, an alert panel like the one shown in Figure 32.

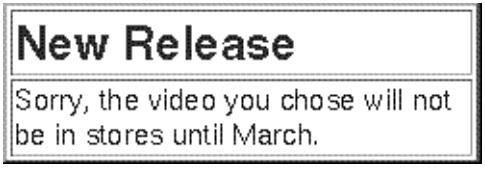

Figure 32. An Alert Panel

This panel is similar to the navigation table shown in Figure 31, but as you'll see, most of the component's attributes are customizable.

To use this component, you simply declare its position within the HTML page and give it a name:

```
<HTML>
<HEAD>
  <TITLE>Alert</TITLE>
</HEAD>
<BODY>

  <WEBOBJECT NAME = "ALERT"></WEBOBJECT>

</BODY>
</HTML>
```

The declarations file specifies the value for each of the panel's attributes, either by assigning a constant value or by binding the attributes value to a variable in the component's code (as with the **alertString** and **infoString** attributes here):

```
ALERT: AlertPanel {
      alertString = alertTitle;
      alertFontColor = "#A00000";
      alertFontSize = 6;
      infoString = alertDescription;
      infoFontSize = 4;
      infoFontColor = "#500000";
      tableWidth = "50%";
};
```

The component's code defines the **alertTitle** and **alertDescription** instance variables or methods, which set the text that's displayed in the upper and lower panes of the alert panel. The **alertDescription** method could, for example, consult a database to determine the release date of the video.

WebObjects Builder makes working with reusable components such as AlertPanel even easier. Component clients can simply drag the alert panel into their components and use the Inspector to set the bindings. They don't need to manually edit the declarations file to set these bindings. To set this up, you, as the component creator, edit a file named **AlertPanel.api** specifying both required and optional attributes. You could, for example, export only the **alertTitle** and **infoString** attributes (leaving the other attributes private) using this **AlertPanel.api** file:

```
Required = (alertTitle, infoString);
Optional = ();
```

See the *WebObjects Tools and Techniques* online book for more information.

Intercomponent Communication

Reusable components can vary widely in scope, from as extensive as an entire HTML page to as limited as a single character or graphic in a page. They can even serve as building blocks for other reusable components. When a reusable component is nested within another component, be it a page or something smaller, the containing component is known as the *parent component*, and the contained component is known as the *child component*. This section examines the interaction between parent and child components.

In the AlertPanel example shown in Figure 32, you saw how the parent component, in its declarations file, sets the attributes of the child component:

```
ALERT: AlertPanel {
      alertString = alertTitle;
      alertFontColor = "#A00000";
      alertFontSize = 6;
      infoString = alertDescription;
      infoFontSize = 4;
      infoFontColor = "#500000";
      tableWidth = "50%";
};
```

Each of the AlertPanel component's attributes is set either statically (to a constant value) or dynamically (by binding the attribute's value to a variable or method invocation in the parent's code). Communication from the parent to the child is quite straightforward.

For reusable components to be truly versatile, there must also be a mechanism for the child component to interact with the parent, either by setting the parent's variables or invoking its methods, or both. This mechanism must be flexible enough that a given child component can be reused by various parent components without having to be modified in any way. WebObjects provides just such a mechanism, as illustrated by the following example.

Consider an AlertPanel component like the one described above, but with the added ability to accept user input and relay that input to a parent component. The panel might look like the one in Figure 33.

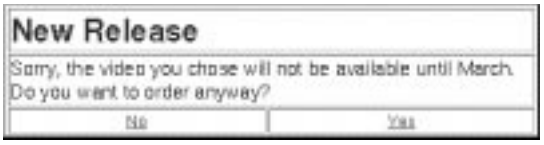

Figure 33. An Alert Panel That Allows User Input

As in the earlier example, you use this component by simply declaring its position within the HTML page:

Parent's Template File

```
<HTML>
<HEAD>
  <TITLE>Alert</TITLE>
</HEAD>
<BODY>

<WEBOBJECT NAME = "ALERT"></WEBOBJECT>

</BODY>
</HTML>
```

The corresponding declarations file reveals two new attributes (indicated in bold):

Parent's Declarations File (excerpt)

```
ALERT: AlertPanel {
      infoString = message;
      infoFontSize = 4;
      infoFontColor = "#500000";
      alertString = "New Release";
      alertFontColor = "#A00000";
      alertFontSize = 6;
      tableWidth = "50%";
      parentAction = "respondToAlert";
      exitStatus = usersChoice;
};
```

The **parentAction** attribute identifies a *callback method*, one that the child component invokes in the parent when the user clicks the Yes or No link. The **exitStatus** attribute identifies a variable that the parent can check to discover which of the two links was clicked. This attribute passes state information from the child to the parent. A reusable component can have any number of callback and state attributes, and they can have any name you choose.

Now let's look at the revised child component. The template file for the AlertPanel component has to declare the positions of the added Yes and No hyperlinks. (Only excerpts of the implementation files are shown here.)

Child Component's Template File (excerpt)

```
<TD>
  <WEBOBJECT name=NOCHOICE></WEBOBJECT>
</TD>
<TD>
  <WEBOBJECT name=YESCHOICE></WEBOBJECT>
</TD>
```

The corresponding declarations file binds these declarations to scripted methods:

Child Component's Declarations File (excerpt)

```
NOCHOICE: WOHyperlink {
      action = rejectChoice;
      string = "No";
};

YESCHOICE: WOHyperlink {
      action = acceptChoice;
      string = "Yes";
};
```

And the script file contains the implementations of the **rejectChoice** and **acceptChoice** methods:

Child Component's Script File (excerpt)

```
id exitStatus;
id parentAction;

- rejectChoice {
  exitStatus = NO;
  return [self performParentAction:parentAction];
}

- acceptChoice {
  exitStatus = YES;
  return [self performParentAction:parentAction];
}
```

Note that **exitStatus** and **parentAction** are simply component variables. Depending on the method invoked, **exitStatus** can have the values YES or NO. The **parentAction** variable stores the name of the method in the parent component that will be invoked by the child. In this example **parentAction** identifies the parent method named **"respondToAlert"**, as specified in the parent's declarations file.

Note: You must enclose the name of the parent's action method in quotes.

Now, looking at the **rejectChoice** and **acceptChoice** method implementations, you can see that they are identical except for the assignment to **exitStatus**. Note that after a value is assigned to **exitStatus**, the child component sends a message to itself to invoke the parent's action method, causing the parent's **respondToAlert** method to be invoked. Since the parent's **usersChoice** variable is bound to the value of the child's **exitStatus** variable, the parent code can determine which of the two links was clicked and respond accordingly. Figure 34 illustrates the connections between the child and parent components.

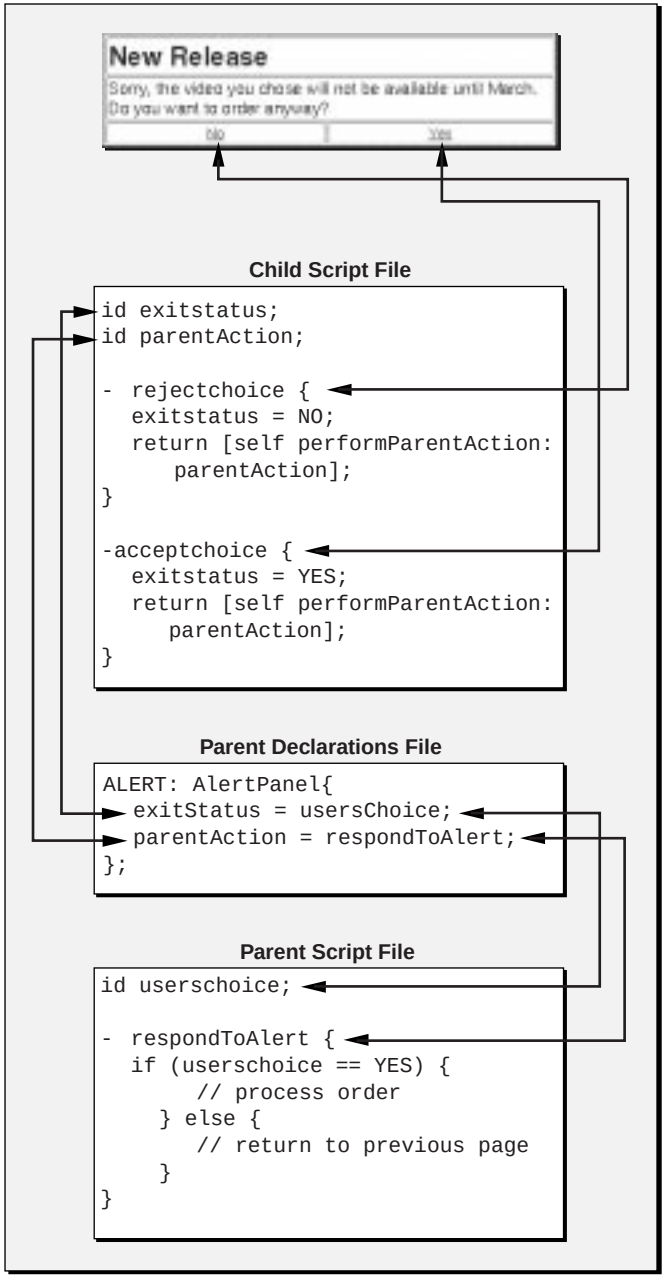

Figure 34. Parent and Child Component Interconnections

The child component's **parentAction** attribute provides a separation between a user action (such as clicking a hyperlink) within a reusable component and the method it ultimately invokes in the parent. Because of this separation, the same child component can be used by multiple parents, invoking different methods in each of them:

Parent1's Declarations File (excerpt)

```
ALERT: AlertPanel {
      ...
      parentAction = "respondToAlert";
      exitStatus = usersChoice;
};
```

Parent2's Declarations File (excerpt)

```
ALERT: AlertPanel {
      ...
      parentAction = "okCancel";
      exitStatus = result;
};
```

Parent3's Declarations File (excerpt)

```
ALERT: AlertPanel {
      ...
      parentAction = "alertAction";
      exitStatus = choice;
};
```

In summary, parent and child components communicate in these ways:

A parent component can, in its declarations file, set child component attributes by:

- Assigning constant values
- Binding an attribute to the value of a variable declared in the parent's code
- Binding an attribute to the return value of a method defined in the parent's code

A child component can communicate actions and values to a parent component by:

- Invoking the parent's callback method
- Setting variables that are bound to variables in the parent, as specified in the parent's declarations file

Synchronizing Attributes in Parent and Child Components

Because WebObjects treats attribute bindings between parent and child components as potentially two-way communication paths, it synchronizes the values of the bound variables at strategic times during the component action request-response loop. This synchronization mechanism has some implications for how you design components. Also, you can disable synchronization for the component action request-response loop as described in "Disabling Component Synchronization" (page 149).

For the sake of illustration, consider a page that displays a value in two different text fields—one provided by the parent component and one by the child (see Figure 35).

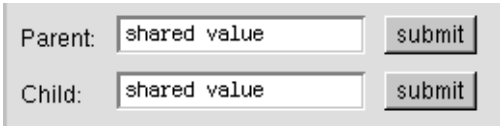

Figure 35. Synchronized Components

Setting the value of either text field and submitting the change causes the new value to appear in both text fields.

The parent's declarations file reveals the binding between the two components:

```
CHILDCOMPONENT: ChildComponent {
     childValue=parentValue;
};
```

When a value is entered in a field and the change submitted, WebObjects will, if needed, synchronize the value in the parent (**parentValue**) and child (**childValue**) at each of the three stages of the component action request-response loop:

- Before and after the components receive the **takeValuesFromRequest:inContext:** message.
- Before and after the components receive the **invokeActionForRequest:inContext:** message.
- Before and after the components receive the **appendToResponse:inContext:** message.

To synchronize values, WebObjects uses key-value coding, a standard interface for accessing an object's properties either through methods designed for that purpose or directly through its instance variables. Key-value coding always first attempts to set properties through accessor methods, reverting to accessing the instance variables directly only if the required accessor method is missing.

Given that synchronization occurs several times during each cycle of the request-response loop and that key-value coding is used to accomplish this synchronization, how does this affect the design of reusable component? It has these implications:

- You rarely need to implement accessor methods for your component's instance variables.

  For instance, it's sufficient in the example shown in Figure 35 to simply declare a **childValue** instance variable in the child component and a **parentValue** instance variable in the parent. You need to implement accessor methods (such as **setChildValue:** and **childValue**) only if the component must do some calculation (say, determine how long the application has been running) before returning the value.

- If you do provide accessor methods, they should have no unwanted side effects and should be implemented as efficiently as possible since they will be invoked several times in a request-response loop cycle.

- If you bind a component's attribute to a method rather than to an instance variable, you must provide both accessor methods: one to set the value and one to return it. This is only the case if no instance variable exists, and does not apply to actions.

  Let's say the parent component in the example shown in Figure 35 doesn't have a discrete **parentValue** instance variable but instead stores the value in some other way (for example, as an entry in a dictionary object). In that case, the parent component must provide both a **parentValue** method (to retrieve the value) and a **setParentValue:** method (to set it). During synchronization, WebObjects expects both methods to be present and will raise an exception if one is missing.

Disabling Component Synchronization

Component synchronization can sometimes lead to values being set when you don't want them to be set. You have no control over when, or how often a value is passed to and from the parent.

For these reasons, component synchronization can be disabled. When components are not synchronized, they behave more like dynamic elements in that a value is not resolved with the parent component's settings until that value is needed.

To disable component synchronization, override the method **synchronizesVariablesWithBindings** in the reusable component's code file to return NO or **false**.

With component synchronization disabled, you must get values from the parent and set values in the parent yourself. The method **valueForBinding:** gets a value from the parent, and the method **setValue:forBinding:** (**setValueForBinding** in Java) set a value in the parent. The argument that you pass to these methods is the name of one of the reusable component's attributes.

For example, consider a reusable component named NonSyncComponent that you declare in a parent component in this way:

```
//parent component's .wod file
Childcomponent : NonSyncComponent {
  stringValue = @"I'm a string!";
}
```

Suppose NonSyncComponent contains a WOTextField element that it declares in this way:

```
// NonSyncComponent.wod
MyString : WOTextField {
  value = someStringValue;
}
```

If NonSyncComponent's script file looks like the following, the value that the parent bound to the **stringValue** attribute is resolved with WOString's **value** attribute whenever WOString requests its **value** attribute. Thus, the WOString in this NonSyncComponent displays "I'm a string!"

```
// NonSyncComponent.wos
- synchronizesVariablesWithBindings {
  return NO;
}

- someStringValue {
  return [self valueForBinding:@"stringValue"];
}

- setSomeStringValue:aValue {
  [self setValue:aValue forBinding:@"stringValue"];
}
```

If NonSyncComponent has no other need for **someStringValue** than to resolve the **value** attribute for its WOString, then NonSyncComponent can instead use this shorthand notation in its declarations file:

```
// Alternate NonSyncComponent.wod
MyString : WOString {
  value = ^stringValue;
}
```

The carat (^) syntax means "use the value that my parent bound to my **stringValue** attribute." This syntax is a convenience that saves you from having to always write cover methods for **valueForBinding:** and **setValue:forBinding:**. In addition to being more convenient, this syntax is often more efficient because none of your code is invoked to do either the pushing or the pulling.

Creating a "Container" Reusable Component

WOComponentContent is a dynamic element that allows you to write nested components as HTML container elements. A container element is an element that can include text and other elements between its opening and closing tags. For example, the HTML FORM element is a container element. As well, WORepetition is a container element.

Using WOComponentContent you can, for example, write a component that defines the header and footer for all of your application's pages. To do so, you'd define a component with HTML similar to the following:

```
<HTML>
  <HEAD>
      <TITLE>Cool WebObjects App</TITLE>
  </HEAD>
  <BODY>
      <!-- A banner common to all pages here -->
      <!-- Start of content defined by the parent element -->
      <WEBOBJECT name=ParentContent></WEBOBJECT>
      <!-- End of content defined by the parent element -->
      <!-- Put a footer common to all pages here. -->
  </BODY>
</HTML>
```

The **<WEBOBJECT>** element on this page is a WOComponentContent element declared like this:

```
ParentContent : WOComponentContent {};
```

WOComponentContent is simply a marker that specifies where the contents wrapped by this component’s **WEBOBJECT** tag should go. You can have only one WOComponentContent element in a given component.

To use the component shown above, you’d wrap the contents of all of the other components in the application with a **<WEBOBJECT>** tag that specifies the component defined above. For example, suppose you named the above component **HeaderFooterPage.wo**. You could use it in another component like this:

```
<!-- HTML for a simple component wrapped with HeaderFooterPage -->
<WEBOBJECT name = templateWrapperElement>
  <P>Hello, world!</P>
</WEBOBJECT>
```

Where **templateWrapperElement** is declared in the **.wod** file like this:

```
templateWrapperElement : HeaderFooterPage {};
```

At run-time, the contents wrapped by **templateWrapperElement** are substituted for the WOComponentContent definition. As a result, the HTML generated for this component would be:

```
<HTML>
  <HEAD>
      <TITLE>Cool WebObjects App</TITLE>
  </HEAD>
  <BODY>
      <!-- A banner common to all pages here -->
      <!-- Start of content defined by the parent element -->
      <P>Hello, world!</P>
      <!-- End of content defined by the parent element -->
      <!-- Put a footer common to all pages here. -->
  </BODY>
</HTML>
```

Sharing Reusable Components Across Applications

If a component is in the WebObjects application's directory, it can be used in many places, but only within that application. This may be fine for some components, but others, you'll want the option to reuse across many applications. If a component is packaged in a framework, it can be used in any WebObjects application.

To package components in a framework for reuse by several applications, do the following:

1. Create a project in Project Builder of type WebObjectsFramework.
2. Add to this project (under Web Components) all of the components that you want to share across applications.
3. Add any resources needed by the components. If the HTTP server needs access to the resource (which is the case for image files and any file that will ultimately be referenced in the HTML file), add it under WebServerResources. Otherwise, add it under Resources.
4. Build the framework. If you perform a make install, it installs the framework in ***NEXT_ROOT*/Library/Frameworks** and the web server resources in **<*DocRoot*>/WebObjects/Frameworks**.

After the framework is installed, you need to set up the applications so that they can use components in that framework. Do the following:

5. In Project Builder, add the framework to the application's project under Frameworks.

6. Build the application.

Search Path for Reusable Components

When WebObjects encounters the name of a reusable component at runtime:

```
NAVCONTROL: NavigationControl {};
```

it must find a WOComponent object to represent the component and then find the component's resources (the HTML template file, image files, and so on).

To find an object to represent the component, WebObjects looks in the run-time system for a subclass of WOComponent with the same name as the component ("NavigationControl" in the example above). For compiled reusable components this search should succeed, but for scripted ones it should fail.

Next, WebObjects looks within the application directory for the reusable component's resources. For example, if you manually start an application that resides in **<DocRoot>/WebObjects/MyWOApps/Fortune.woa**, the **Fortune.woa** directory will be searched.

If WebObjects doesn't find the component there, it assumes that the reusable component is included in a framework. It searches all frameworks that were linked in to the application executable for a component with that name. For example, applications written entirely in WebScript use the default application executable, **WODefaultApp**. This executable is linked to the frameworks **WebObjects.framework** and **WOExtensions.framework**; any components defined in WOExtensions can be used in a scripted application.

Designing for Reusability

Here are some points to consider when creating reusable components:

- Make sure that your reusable component generates HTML that can be embedded in the HTML of its parent component.

  A reusable component should be designed to be a "good citizen" within the context in which it will be used. Thus, for example, the template file for a reusable component should not start and end with the <HTML> and </HTML> tags (since these tags will be supplied by the parent component). Similarly, it is unlikely that a reusable component's template would contain <BODY>, <HEAD>, or <TITLE> tags.

  Further, if you intend your component to be used within a form along with other components, don't declare the form (<FORM...> ... </FORM>) within the reusable component's template file. Instead, let the parent component declare the form. Similar considerations pertain to submit buttons. Since most browsers allow only one submit button within a form, putting a submit button in a reusable component severely limits where it can be used.

- Guard against name conflicts.

  Reusable components are identified by name only. See "Search Path for Reusable Components" (page 153). Those that reside within a particular application's application directory are available only to that application. Those that reside in a framework (for example, **WOExtensions.framework**) are available to all applications that link to it. Suppose you have a component named NavigationControl in your application and one of the frameworks that your application links to also has a NavigationControl component. Which one will be used in your application? The result is indeterminate.

Reusable component names need to be system-wide unique. Consider adding a prefix to component names to increase the likelihood that they will be unique.

- Provide attributes for all significant features.

  The more customizable a component is, the more likely it is that people will be able to reuse it. For example, if the AlertPanel component discussed in “Centralizing Application Resources” (page 135) let you set the titles of the hyperlinks (say, to OK and Cancel, or Send Now and Send Later), the panel could be adapted for use in many more applications.

- Provide default values for attributes wherever possible.

  Don’t require people to set more attributes than are strictly required by the design of your reusable component. In your component’s initialization method, you can provide default values for optional attributes. When the component is created, the attribute values specified in the initialization method are used unless others are specified in the parent’s declarations file.

  For example, the AlertPanel component’s **init** method could set these default values:

```
- init {
        [super init];
        alertString = @"Alert!";
        alertFontColor = @"#ff0000";
        alertFontSize = 6;

        infoString = @"User should provide an infoString";
        infoFontColor = @"#ff0000";
        infoFontSize = 4;

        borderSize = 2;
        tableWidth = @"50%";
        return self;
}
```

  Then, in a declarations file, you are free to specify all or just a few attributes. This declaration specifies values for all attributes:

Complete Declaration

```
ALERT: AlertPanel {
       infoString = message;
       infoFontSize = 4;
       infoFontColor = "#500000";
       alertString = "New Release";
       alertFontColor = "#A00000";
       alertFontSize = 6;
       tableWidth = "50%";
};
```

This declaration specifies a value for just one attribute; all others will use the default values provided by the component's **init** method:

Partial Declaration

```
ALERT: AlertPanel {
       alertString = "Choice not available.";
};
```

- Consider building reusable components from reusable components.

  Rather than building a monolithic component, consider how the finished component can be built from several, smaller components. You may be able to employ these smaller components in more than one reusable component.

- Document the reusable component's interface and requirements.

  If you plan to make your components available to other programmers, you should provide simple documentation that includes information on:

  - What attributes are available and which are required
  - What the default values are for optional attributes
  - What context needs to be provided for the component. For example, does it need to be embedded in a form?
  - Any restrictions that affect its use. For example, is it possible to have a submit button in the same form as the one that contains this component?

  In addition, it's helpful if you provide an example showing how to use your component.

Chapter 7

Managing State

Most applications must be able to preserve some application state between a user's requests. For example, if you're writing a catalog application, you must keep track of the items that the user has selected before the user actually fills out the purchasing information. By default, WebObjects stores application state on the server. If this doesn't meet your needs, WebObjects provides several alternatives strategies for storing state.

This chapter describes why, when, and how to store state in a WebObjects application. It compares all of the available state-storage strategies, shows you how to implement your own state-storage strategy, plus it describes how to control the amount of application state stored.

If you're fairly new to WebObjects programming, you'll probably just want to read the first three sections of this chapter and skip the rest. As you begin to write larger, more complex applications, memory demands and performance become an issue. At that point, you should read the rest of this chapter to learn about alternative state-storage strategies and how you can control the amount of state stored.

Before reading this chapter, you should be familiar with concepts presented in "WebObjects Viewed Through Its Classes."

Why Do You Need to Store State?

Originally, the World Wide Web was designed solely for "stateless" applications. An application could display pages and even request information from the user, but it couldn't keep track of a particular user from one transaction to the next. Such an application is like a person with no long-term memory. Each interaction begins with not so much as a "Haven't we met somewhere before?" and ends with an implied "Farewell forever!" Stateless applications aren't well-suited for on-line commerce since it wouldn't do to lose a customer's order between the catalog and billing pages. A remedy had to be found.

Given the ingenuity of software developers, not one but several solutions have been advanced. They fall into two basic categories:

- Storing state information on the client's machine. With each transaction the client passes the state information back to the server, in effect "reminding" the server of the client's identity and the state information associated with that client.

- Storing state information on the server. With each transaction, the web application locates the state information associated with a request from a particular client. The state information might be stored in memory, in a file on disk, or in a standard database, depending on the application.

Passing state back to the client with every transaction simplifies the accounting associated with state management but is inefficient and can constrain the design of your site. Storing state on the server, on the other hand, requires sophisticated applications that can keep track of per-session information no matter how many users are accessing the application simultaneously. However, without support from your programming environment, storing state on the server is not an attractive option.

As you'll see in this chapter, WebObjects lets you easily make use of any of these state-storage solutions. For a given application, state management can be as simple as selecting the management strategy you want to use and identifying the information that you want stored on a per-session basis. The WebObjects framework does the rest no matter how many users will be accessing the application simultaneously.

When Do You Need to Store State?

Web applications that store state information are somewhat more complex than those that don't. State storage can also raise performance and scalability issues (such as how much physical storage an application server should have for a given number of simultaneous users). Given these considerations, it's clearly best to avoid storing state.

Note: If your application does not need to store state, you might consider implementing it entirely using direct actions. See "WebObjects Viewed Through Its Classes" (page 95).

Applications differ widely in their state storage requirements. At one extreme are simple applications that vend read-only pages (company information, specifications for hardware devices, and so on). These traditional World Wide Web applications don't need to store state information. At the other extreme are commercial applications that let users wheel virtual shopping carts from page to page, selecting items for purchase. These applications must keep track of order information on a per-user basis. Considering that a popular site could have scores of simultaneous sessions, these commercial applications must employ a sophisticated means of handling state for each session. Somewhere between these extremes are applications with simple state storage requirements, such as keeping track of the total number of votes on an issue, the number of visitors to the web site, and so on.

Characteristically, WebObjects takes an object-oriented approach to fulfilling any of these state-storage requirements.

Objects and State

Three classes manage state in an application—WOApplication, WOSession, and WOComponent. An application object handles state associated with the application as a whole, session objects handle state associated with a particular user session within the application, and component objects handle state associated with a particular page or component within a session.

The Application Object and Application State

The application object is the logical place to store data that needs to be shared by all components in all sessions of an application. Application state is typically stored in the application object's instance variables.

For example, the following application object keeps information about the cars available and the possible prices:

```
// Java Application.java
public class Application extends WOApplication {
  public NSDictionary carData;
  public NSArray prices;
  public NSArray sortBys;

  public Application() {
      super();
      String filePath = resourceManager()
          .pathForResourceNamed("CarData.dict", null, null);
      if (null != filePath) {
          try {
              carData = new NSDictionary(new
              java.io.FilePath(filePath));
          }
          catch (Exception e) {
              //...
          }
      } else {
          // ...
      }
      int priceValues[] = { 8000, 10000, 12000, 14000, 16000, 18000,
          20000, 25000, 30000, 50000, 90000};
      NSMutableArray a = new NSMutableArray();
      for (int i=0; i<priceValues.length; i++) {
          Number num = new Integer(priceValues[i]);
          a.addObject(num);
      }
      prices = (NSArray) a;

      String sortByStrings[] = { "Price", "Type", "Model" };

      for (int i=0; i<sortByStrings.length; i++) {
          a.addObject(sortByStrings[i]);
      }
      sortBys = (NSArray) a;

}
```

```
// WebScript Application.wos
id carData;
id prices;
id sortBys;

- init {
  id filePath;

  [super init];
  filePath = [[self resourceManager]
pathForResourceNamed:@"CarData.dict" inFramework:nil
  languages:nil];
  if (filePath)
      carData =
          [NSDictionary dictionaryWithContentsOfFile:filePath];
  //...
  prices = @(8000, 10000, 12000, 14000, 16000, 18000, 20000, 25000,
30000, 50000, 90000);
  sortBys = @("Price", "Type", "Model");
  return self;
}
```

The WOComponent class defines a method **application**, which provides access to the component's application object. So any component can access application state this way:

```
//Java
public boolean isLuckyWinner() {
  Number sessionCount =
application().statisticsStore().objectForKey(
      "Total Sessions Created");
  if (sessionCount == 1000) {
      return true;
  return false;
}
```

```
// WebScript
- isLuckyWinner {
  id sessionCount = [[[self application] statisticsStore]
      objectForKey:@"Total Sessions Created"];
  if (sessionCount == 100])
      return YES;
  return NO;
}
```

WODirectActions can access application state using a similar method defined in WODirectAction. Sessions can use the WOApplication **application** class or static method.

If you're implementing direct action request handling, the only type of state you store by default is application state. Session objects are not created unless specifically requested, and components and WODirectActions do not persist between cycles of the request-response loop.

Application state persists for as long as the application is running. If your site runs multiple instances of the same application, application state must be accessible to all instances. In this case, application state might be best stored in a file or database, where application instances could easily access it. This approach is also useful as a safeguard against losing application state (such as the number of visitors to the site) if an application instance crashes.

The Session Object and Session State

A more interesting type of state that web applications can store is the state associated with a user's session. This state might include the selections a user makes from a catalog, the total cost of the selections so far, or the user's billing information.

You typically store session state as instance variables in your application's session object. It's also possible to store session state within a special dictionary provided by the session object, as we'll see shortly.

Session state is directly accessible to any component within the application (although those components can access only the state stored for their current session). The WOComponent class defines a **session** method that provides this access. For example, the component can access a session's instance variable in this way:

```
// WebScript
elapsedTime = [[self session] timeSinceSessionBegan];

//Java
elapsedTime = this.session().timeSinceSessionBegan();
```

With direct action request handling, sessions are not restored by default. If your direct action needs to interact with session data, however, WODirectAction provides a **session** method. Unlike WOComponent's **session** method, WODirectAction's **session** method checks for the existence of a session object and creates one if one does not exist.

The WOSession class provides a dictionary where state can be stored by associating it with a key. WOSession's **setObject:forKey:** and **objectForKey** methods (in Java, **setObject** and **objectForKey**) give access to this dictionary. For an example of when this session dictionary might be useful, consider a web site that collects users' preferences about movies. At this web site, users work their way through page after page of movie listings, selecting their favorite movie on each page. At the bottom of each page, a "Choices" component displays the favorites that have been picked so far in the user's session. The Choices component is a general-purpose reusable component that might be found in various applications.

The designer of the Choices component decided to store the sessionwide list in the session dictionary:

```
[[self session] setObject:usersChoiceArray forKey:@"Choices"];
```

By storing the information in the session dictionary rather than in a discrete session instance variable, this component can be added to any application without requiring code changes such as adding variables to the session object.

This approach works well until you have multiple instances of a reusable component in the same page. For example, what if users were asked to pick their most *and least* favorite movies from each list, with the results being displayed in two different Choices components in each page. In this case, each component would have to store its data under a separate key, such as "BestChoices" and "WorstChoices".

A more general solution to the problem of storing state when there are multiple instances of a reusable component is to store the state under unique keys in the session dictionary. One way to create such keys is to concatenate the component's name, context ID, and element IDs:

```
//Java example
String componentName;
Context context;
String contextID;
String elementID;
String uniqueKey;

context = this.context();
componentName = context.component().name();
contextID = context.contextID();
elementID = context.elementID();
uniqueKey = componentName + "-" + contextID + "-" + elementID;
this.session().setObject(someState, uniqueKey);

// WebScript example
id componentName;
id context;
id contextID;
id elementID;
id uniqueKey;

context = [self context];
componentName = [[context component] name];
contextID = [context contextID];
elementID = [context elementID];
uniqueKey = [NSString stringWithFormat:@"%@-%@-%@", componentName,
  contextID, elementID];
[[self session] setObject:someState forKey:uniqueKey];
```

Since, for a given context, each element in a page has its own element ID, combining the context and element IDs yields a unique key. The component name is added to the key for readability during debugging.

As described in the chapter "WebObjects Viewed Through Its Classes" (page 95), the URLs that make up the requests to a WebObjects application contain an identifier for a particular session within the application. Using this identifier, the application can restore the state corresponding to that session before the request is processed. If the request is that of a user contacting the application for the first time, a new session object is created for that user.

As you can imagine, storing data for each session has the potential of consuming excessive amounts of resources, so WebObjects lets you set a time-out for the session objects and lets you terminate them directly.

In summary, session state is accessible only to objects within the same session and persists only as long as the session object persists.

Component Objects and Component State

In WebObjects, state can also be scoped to a component, such as a dynamically generated page or a reusable component within a page. Common uses for component state include storing:

- A list of items that a user can choose from within a particular page
- The user's selection from that list
- Information that the users enters in a form
- Default values for a component's attributes

Component state typically includes the data that a page displays, such as a list of choices to present to the user. Suppose a user requests the page that lists these choices. The component that represents the page must initialize itself with the choice data and then return the response page. This completes one cycle of the request-response loop. Now, suppose the user looks at the list of choices, selects the third item, and submits a new request. The same component must be present in this second cycle to identify the choice and take the appropriate action. In short, component state often needs to persist from one cycle of the request-response loop to the next.

A simple example of component state can be seen in the first page of this sample application, which lists models, prices, and types of vehicles for the user to choose from (see Figure 36).

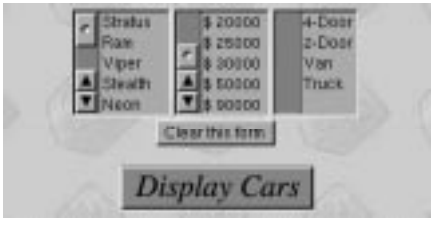

Figure 36. Sample Application First Page

This component declares instance variables for the values displayed in the browser and for the user's selection from the browsers. Before the page can be sent to the user, the instance variables that hold the values to be displayed (**model, price, type**) are initialized:

```
// Java Main.java
NSArray models, prices, types;
NSMutableArray selectedModels, selectedPrices, selectedTypes;
String model, price, type;

public Main() {
  super();
  NSEnumerator en;

  Application woApp = (Application)application();
  en = woApp.modelsDict().objectEnumerator();
  models = new NSMutableArray();
  while (en.hasMoreElements())
      ((NSMutableArray)models).addElement(en.nextElement());
  en = woApp.typesDict().elements();
  while (en.hasMoreElements())
      ((NSMutableArray)types.addElement(en.nextElement());
  prices = woApp.prices();
}

// WebScript Cars Main.wos
id models, model, selectedModels;
id prices, price, selectedPrices;
id types, type, selectedTypes;

- init {
  id anApplication = [WOApplication application];
  [super init];
  models = [[anApplication modelsDict] allValues];
  types = [[anApplication typesDict] allValues];
  prices = [anApplication prices];
  return self;
}
```

(The **selectedModels**, **selectedPrices**, and **selectedTypes** instance variables are bound to the **selections** attributes of the three WOBrowsers and so will contain the user's selections when the Display Cars button is clicked.)

When a user starts a session of the above application, the Main component's initialization method is invoked, initializing the component's instance variables from data accessed through the application object. From this point on, the Main component and its instance variables become part of the state stored for that user's session of the application. When the session is released, the component is also released. However, there are other techniques that allow you to control resource allocation on a component basis, as you'll see later in this chapter.

As with the session state, a component's state is accessible to other objects within the same session. As the result of a user's action, for example, it's quite common for one component to create the component for the next page and set its state. Consider what happens in the above sample application when the user makes a selection in the first page and clicks Display Cars. The **displayCars** method in the Main component is invoked:

```
// Java Main.java
public WOComponent displayCars()
{
  SelectedCars selectedCarsPage =
      (SelectedCars)this.pageWithName("SelectedCars");

  ...

  selectedCarsPage.setModels(selectedModels);
  selectedCarsPage.setTypes(selectedTypes);
  selectedCarsPage.setPrices(selectedPrices);
  ...
  selectedCarsPage.fetchSelectedCars();

  return (WOComponent)selectedCarsPage;
}

// WebScript Cars Main.wos
- displayCars
{
  id selectedCarsPage = [this pageWithName:@"SelectedCars"];

  ...

  [selectedCarsPage setModels:selectedModels];
  [selectedCarsPage setTypes:selectedTypes];
  [selectedCarsPage setPrices:selectedPrices];
  ...
  [selectedCarsPage fetchSelectedCars];

  return selectedCarsPage;
}
```

The new component is created by sending a **pageWithName:** message to a WOComponent object or a WODirectAction object. A series of messages is then sent to this new object to set its state before the object is returned as the response page.

Component state persists until the component object is deallocated, an action that can occur for various reasons, as described in the section "Controlling Component State" (page 178).

State Storage Strategies

WebObjects gives you the option of storing state in a couple of different ways:

- **In the server.** State is maintained in memory within a WebObjects application.
- **In custom stores.** State is stored using a mechanism of your own design.

By default, WebObjects uses the first approach, storing state on the server. To determine whether you should use this default approach or try one of the other state-storage solutions, read "A Closer Look at Storage Strategies." If you decide to use another state-storage solution, you may have to set up custom objects so that they can be archived. To learn more about this issue, read "Storing State for Custom Objects" (page 172).

An additional option when storing the state on the server is to utilize cookies for persistently storing the client's **sessionID** and **instanceID**. This allows the client to transparently have their session restored if they leave the application and return before their session has timed out. Methods on WOSession allow for the customization of this feature. This feature will be explained in more detail below.

You may find you need to control the amount of state that is stored. The sections "Controlling Session State" (page 176) and "Controlling Component State" (page 178) tell you how to do so.

A Closer Look at Storage Strategies

In a normal WebObjects application, you should set the session storage mechanism as early as possible, usually in the application object's initialization method or constructor. You set the mechanism by sending the application object a **setSessionStore:** message. This method takes a WOSessionStore object as an argument. WOSessionStore declares the serverSessionStore method to create server session stores.

State in the Server

Storing state in memory on the application server is the default behavior, so no setup is necessary. However, if you want access to the session store object, you can include the following code in the application object's initialization method:

```
// WebScript example
id sessionStore = [WOSessionStore serverSessionStore];
[self setSessionStore:sessionStore];

// Java example
WOSessionStore sessionStore =
WOSessionStore.serverSessionStore();
this.setSessionStore(sessionStore);
```

When state is stored in the server, the application keeps a cache of session objects in the session store, and each session keeps a cache of component objects. Because these objects can take up a lot of memory, WebObjects gives you ways to control the amount of memory this state storage mechanism consumes:

- Setting session time-outs (see "Controlling Session State")
- Setting the size of the page cache (see "Adjusting the Page Cache Size")

A significant consequence of storing state in memory is the effect on load-balanced applications. When an application is load balanced, there are several instances of that application running on different physical machines to reduce the load on a particular server. (The online book *Serving WebObjects* describes how to set this up.) WebObjects can route any request to any application instance running on any machine as long as that instance doesn't store state in memory in the server (that is, as long as the application is stateless or uses one of the other state-storage mechanisms described in this chapter). When state is stored in the server, however, it is stored in the application instance. Because state is stored in the application instance, all requests made by one session must return to that instance.

Using Cookies

Storing the sessionID in a cookie can give your application added benefits. For example, if a client is required to log in to your application then by storing the IDs in a cookie that client will not be forced to log in again if they exit your application and then come back later. This is dependent, of course, on that session being around when the client comes back to your application. You must set the session timeouts to longer periods of time in these situations.

You configure this feature using the WOSession methods:

```
- setStoresIDsInURLs:
- storesIDsInURLs
- setStoresIDsInCookies:
- storesIDsInCookies
```

Storing State for Custom Objects

When state is stored in the server, the objects that hold state are kept intact in memory between cycles of the request-response loop. In contrast, custom state storage mechanisms may ask objects to archive themselves (using classes and methods defined in the Foundation framework) before being put into storage. The objects that are part of the WebObjects and Foundation frameworks can archive themselves, so they require no effort on your part. But if your application has custom classes that need to store state, these classes must know how to archive and unarchive themselves. How you implement archiving for custom classes depends on whether your application accesses a database. If your application accesses a database, it uses the Enterprise Objects Framework and should use the EOEditingContext class to archive objects. If your application doesn't access a database, it should use the NSArchiver class to archive custom objects.

Archiving Custom Objects in a Database Application

If your application accesses a database, it uses the Enterprise Objects framework and should use the EOEditingContext class to archive objects. An editing context manages a graph of enterprise objects that represent records fetched from a database. You send messages to the editing context to fetch objects from the database, insert or delete objects, and save the data from the changed objects back to the database. (See the *Enterprise Objects Framework Developer's Guide* for more information.)

In WebObjects, applications that use the Enterprise Objects Framework must enlist the help of the EOEditingContext class to archive enterprise objects. The primary reason is so that EOEditingContext can keep track, from one database transaction to the next, of the objects it is designed to manage. But using an EOEditingContext for archiving also benefits your application in these other ways:

- During archiving, an EOEditingContext stores only as much information about its enterprise objects as is needed to reconstitute the object graph at a later time. For example, unmodified objects are stored as simple references that will allow the EOEditingContext to recreate the object from the database at a later time. Thus, your application can store state very efficiently by letting an EOEditingContext archive your enterprise objects.

- During unarchiving, an EOEditingContext can recreate individual objects in the graph only as they are needed by the application. This approach can significantly improve an application's perceived performance.

An enterprise object (like any other object that uses the Foundation archiving scheme) makes itself available for archiving by declaring that it conforms to the NSCoding protocol and by implementing the protocol's two methods, **encodeWithCoder:** and **initWithCoder:**. It implements these methods like this:

```
// WebScript example
- encodeWithCoder:(NSCoder *)aCoder {
  [EOEditingContext encodeObject:self withCoder:aCoder];
}

- initWithCoder:(NSCoder *)aDecoder {
  [EOEditingContext initObject:self withCoder:aDecoder];
  return self;
}
```

The Java packages provide a different archiving mechanism; your Java classes should implement the **java.io.Serializable** interface. This interface consists of two methods: **writeObject**, which roughly corresponds to **encodeWithCoder:**; and **readObject**, which roughly corresponds to **initWithCoder:**.

```
// Java example
private void writeObject(java.io.ObjectOutputStream out)
throws IOException {
  EOEditingContext.writeObjectToStream(this, out);
}

private void readObject(java.io.ObjectInputStream in)
throws IOException, ClassNotFoundException{
  EOEditingContext.initObjectFromStream(this, in);
}
```

The enterprise object simply passes on responsibility for archiving and unarchiving itself to the EOEditingContext class, by invoking the **encodeObject:withCoder:** and **initObject:withCoder:** methods in WebScript or Objective-C (**writeObject** and **readObject** in Java) and passing a reference to itself (**self** or **this**) as one of the arguments. The editing context takes care of the rest. (See the EOEditingContext class specification in the *Enterprise Objects Framework Reference* for more information.)

Archiving Custom Objects in Other Applications

Custom classes that can't take advantage of an EOEditingContext for archiving must take a different approach. These classes still must conform to the NSCoding protocol and implement its **encodeWithCoder:** and **initWithCoder:** methods; however, you must implement them differently. In **encodeWithCoder:**, you use the coder argument provided to encode the object's instance variables. In **initWithCoder:**, the object uses the decoder provided to initialize itself.

For example:

```
- encodeWithCoder:(NSCoder *)coder {
  [coder encodeObject:carID];
  [coder encodeObject:colorID];
  [coder encodeObject:colorPicture];
  [coder encodeObject:packagesIDs];
  [coder encodeObject:downPayment];
  [coder encodeObject:leaseTerm];
}

- initWithCoder:(NSCoder *)coder {
  self = [super init];
  carID = [[coder decodeObject] retain];
  colorID = [[coder decodeObject] retain];
  colorPicture = [[coder decodeObject] retain];
  packagesIDs = [[coder decodeObject] retain];
  downPayment = [[coder decodeObject] retain];
  leaseTerm = [[coder decodeObject] retain];
  car = nil;
  return self;
}
```

The Java version of the same example looks like this:

```
public void encodeWithCoder(NSCoder coder) {
  coder.encodeObject(leaseTerm);
  coder.encodeObject(downPayment);
// The custom Car object that contains all other info about the car.
  coder.encodeObject(car);
}

public ShoppingCart(NSCoder coder) {
  super();
  leaseTerm = coder.decodeObject();
  downPayment = coder.decodeObject();
  Car aCar = (Car)coder.decodeObject();
  setCar(aCar);
}
```

For more information on archiving, see the class specifications for NSCoding, NSCoder, NSArchiver, and NSUnarchiver in the *Foundation Framework Reference*.

Controlling Session State

Maintaining state in memory on the server can consume considerable resources, so WebObjects provides a number of mechanisms to control how much state is stored. This section takes a closer look at how you manage sessionwide state.

Take care that your application only stores state for active sessions and stores the smallest amount of state possible. WOSession lets you control these factors by providing a time-out mechanism for inactive sessions and by providing a way to specify exactly what state to store between request-response loop cycles.

Setting Session Time-Out

By assigning a time-out value to a session, you can ensure that the session will be deallocated after a specific period of inactivity. WOSession's **setTimeOut:** method lets you set this period and **timeOut** returns it.

Here's how the session time-out works: After a cycle of the request-response loop, WebObjects associates a timer with the session object that was involved in the request and then puts the session object into the session store. The timer is set to the value returned by the session object's **timeOut** method. If the timer goes off before the session is asked to handle another request, the session and its resources are deallocated. A user submitting a request to a session that has timed out receives an error message.

By default, a session object's time-out value is set to 3600 seconds. You should set the session time-out for your application to the shortest period that seems reasonable. You can specify the session timeout either with the WOSessionTimeout user default (user defaults such as this can be specified on the command-line when you start your application), or

in your session initialization code. For example, to set the time-out to ten minutes, you could send this **setTimeOut:** message in your session's initialization method:

```
// WebScript Session.wos
- init {
  [super init];
  [self setTimeOut:600];
  return self;
}

// Java Session.java
public Session() {
  super();
  this.setTimeOut(600);
}
```

The argument to **setTimeOut:** is interpreted as a number of seconds.

At times, a user's choice signals the end of a session (such as when the Yes button is clicked in response to the query, "Do you really want to leave the Intergalactic Web Mall?"). If you are sure a session has ended, you can send a **terminate** message to the session object, marking it (and the resources it holds) for release.

A session marked for release won't actually be released until the end of the current request-response loop. Other objects may need to know whether a particular request-response loop is their last, so they can close files or do other clean up. They can learn their fate by sending the session object an **isTerminating** message.

Using awake and sleep

Another strategy for managing session state is to create it at the beginning of the request-response loop and then release it at the end. The session object's **awake** and **sleep** methods provide the hooks you need to implement this strategy. A session object receives an **awake** message at the beginning of the request-response loop (where you can re-initialize the session state) and a **sleep** message at the end (where you can release it). Remember that **awake** and **sleep** are not sent during the direct action request-response loop, but the direct action request-response loop does not use session objects by default.

Controlling Component State

Component objects exist within a particular session and are stored along with the session object between each cycle of the component action request-response loop. Since a user can visit many pages during a session, managing component state can be crucial to reducing your application's storage requirements.

Managing Component Resources

Typically, page caching occurs both on the client machine and on the WebObjects application server. WOApplication provides methods to control caching on either end of a web connection. This section discusses server-side caching and the section "Client-Side Page Caching" looks at the consequences of page caching on the client.

There are two common techniques for controlling component resources:

- Adjusting the page cache size
- Using **awake** and **sleep** to initialize and release resources

Adjusting the Page Cache Size

As noted in "WebObjects Viewed Through Its Classes" (page 95), except for the first request, a component action request to a WebObjects application contains a session ID, page name, and context ID. The application uses this information to ask the appropriate session object for the page identified by the name and context ID. As long as the page is still in the cache, it can be retrieved and enlisted in handling the request.

By default, a WebObjects application caches the last 30 pages that a user has visited within a session. You can change the size of the cache using the application object's **setPageCacheSize:** method and retrieve the cache size with the **pageCacheSize** method. Within each session, new pages are added to the cache until the cache size limit is reached. Thereafter, for each new page added to the cache, the cached page object representing the least recently visited page is released.

To reduce the resource requirements for an application, you could set the page cache to a smaller number. However, doing so increases the possibility that a request could address a page that is no longer in the cache. For example, if you set the page cache size to four, a user could backtrack five pages to an order form, make some changes, and resubmit the form. The result would be an error page.

To keep users from encountering this error, your application should maintain a moderate sized cache of pages. The default cache size of 30 pages is a reasonable value that protects users from reaching the backtracking limit under normal conditions; however, you can adjust the limit to any positive value you like or even zero.

Setting the page cache size to 0 has two effects. As expected, it disables page caching. Furthermore, it signals to WebObjects that you intend to provide for component state persistence rather than rely on WebObjects' inherent support. Thus, if you set the cache size to 0, no error page is generated if a request addresses a page that can't be found in the cache. Instead, WebObjects creates a new page by sending the application object a **pageWithName:** message. Since with this model pages do not persist from one request to the next, you assume responsibility for maintaining any needed component state. For this reason, it's rarely advisable to turn off page caching.

If your application uses frames and uses the default page cache size of 30, your users are more likely to see a backtracking error message. In frames-based applications, the first few pages that the application generates (for example the navigation frame) often stay visible for long periods without being regenerated. It's possible for those pages to be bumped from the page cache.

For this reason, WebObjects actually supports two separate page caches: the page cache described above and a persistent page cache. If you want to ensure that a particular component is never bumped from the page cache, you should add it to the persistent page cache using the WOSession method **savePageInPermanentCache:**. For example, if your application always shows a navigation component in the left frame, you should add that component to the persistent page cache. This way, your users never see the backtracking error message in the application's left frame. Adjust the size of the persistent page cache with WOApplication's **setPermanentPageCacheSize:** method.

Using awake and sleep

Another way to control the amount of component state that's maintained between cycles is to make use of WOComponent's **awake** and **sleep** methods. Unlike WOComponent's **init** method that's invoked just once in the life of the component, a component's **awake** and **sleep** methods are invoked at the beginning and end of any request-response loop that involves the component.

By moving a component's variable initialization routines from its **init** method to its **awake** method and implementing a **sleep** method to release those variables, you can reduce the space requirements for storing a component. For example, the code shown in the section "Component Objects and Component State" (page 167) could be changed to:

```
// rewritten Main.wos
id models, model, selectedModels;
id prices, price, selectedPrices;
id types, type, selectedTypes;

- awake {
  anApplication = [WOApplication application];
  models = [[anApplication modelsDict] allValues];
  types = [[anApplication typesDict] allValues];
  prices = [anApplication prices];
}

- sleep {
  models = nil;
  types = nil;
  prices = nil;
}
```

Note that in WebScript you set a variable to **nil** to mark it for release. In Objective-C you send the object a **release** message:

```
- sleep {
  [models release];
  models = nil;
  [types release];
  types = nil;
  [prices release];
  prices = nil;
}
```

Of course, what you save in storage by moving variable initialization to the **awake** method is lost in performance, since these variables will be reinitialized on each cycle of the request-response loop.

Client-Side Page Caching

When accessing a web page, the user's browser associates the URL with the HTML page it downloads from the server and stores this information on the user's machine. If the browser is asked to display the URL again at a later date, it fetches the cached page rather than emitting another request. In many cases, this short-circuit is desirable because it reduces network traffic and increases a web site's perceived responsiveness.

Sometimes, however, you need to make sure the user is seeing the most up-to-date information. You must therefore disable client-side caching. WOApplication provides the **setPageRefreshOnBacktrackEnabled:** method for this purpose. In general, you send this message in the application object's initialization method:

```
// WebScript example
- init {
  [super init];
  [self setPageRefreshOnBacktrackEnabled:YES];
  return self;
}
```

The **setPageRefreshOnBacktrackEnabled:** method adds a header to the HTTP response. This header sets the expiration date for an HTML page to the date and time of the creation of the page. Later, when the browser checks its cache for this page, it finds that the page is no longer valid and so refetches it by resubmitting the request URL to the WebObjects application.

If you only need to disable client caching for certain responses within your application, you can use WOResponse's **disableClientCaching** method to disable caching on a response-by-response basis.

A WebObjects application handles a page-refresh request differently than it would a standard request. When the application determines that the request URL is identical to one it has previously received (that is, the session and context IDs in the request URL are identical to those in a request it has previously received), it simply returns the response page that was associated with this earlier request. The first two steps of a normal component action request handling loop (value extraction from the request and action invocation) don't occur.

Page Refresh and WODisplayGroup

If you're using a WODisplayGroup object in your application, you must enable page refresh so that the application and the client browser stay in agreement about which objects are being displayed.

A WODisplayGroup holds a set of objects (generally enterprise objects fetched from a database) and provides "batched" access to these objects. For example, if a user submits a query (such as, "Show me the movies released in 1996.") to a Movies application, a WODisplayGroup might return 10 records at a time to the user's browser. The application would offer controls to let the user display the next and previous batches of 10 movie titles. When the user decides to order one of the movies, the WODisplayGroup needs to know which batch the item comes from.

As the user presses the Next Ten Movies or Previous Ten Movies buttons, the WODisplayGroup updates its record of which 10 movies are being displayed. When the user decides to order the second movie in the list, the WODisplayGroup can determine the actual record since it knows which batch is being displayed and which record is number 2 in that batch. But if the user backtracks to a previous page (with page refresh disabled) and chooses the second record, the WODisplayGroup will erroneously pick the second record from its current batch. By enabling page refresh, the WODisplayGroup is alerted each time the user backtracks and can update its notion of the current batch, eliminating this problem.

Chapter 8

Deployment and Performance Issues

After you've written your application, tested it, and verified that it works properly, it's ready to be deployed for use by your customers. Before you deploy, you'll want to perform some finishing touches.

This chapter describes finishing touches that you might want to add after you're through debugging the bulk of your application's code. It covers such topics as how to record application usage statistics, how to shut down an application gracefully, how to substitute your own code when an error occurs, and how to improve the application's performance.

You'll also want to read the online document *Serving WebObjects*. This document is intended for the person who sets up the website and maintains the application after it is deployed. It covers such topics as how to perform load balancing, how to maintain a log file, and how to test and improve performance of the applications running on a site.

Recording Application Statistics

While your application runs, a WOStatisticsStore object records statistics about the application. It records such information as how many sessions are active, how many requests have been processed, and which pages have been accessed. This section describes how to maintain a log file, access those statistics, and add to them.

Maintaining a Log File

WOStatisticsStore has the ability to record session information to a log file that can be analyzed by a Common Log File Format (CLFF) standard analysis tool. WOStatisticsStore does not maintain this log file by default. To store information in a log file, you must set the path to the log file early in your application. For example:

```
// Java
public Application() {
  super();

this.statisticsStore().setLogFile("/tmp/WebObjects.log
", 1);
  ...
}
```

When a log file is set, WOStatisticsStore records all information returned by **descriptionForResponse:inContext:** to that log file at the end of each cycle of the request-response loop. Note that **descriptionForResponse:inContext:** is only invoked during component actions; thus, the log file won't contain entries for direct actions.

Note that descriptions are not saved in the log file in sequential order. A session accumulates descriptions as a user traverses from page to page, and then saves them all at once in the CLFF log file when the session times out. Thus, the WebObjects CLFF log file has a big advantage over a typical HTTP CLFF log file (which saves accesses sequentially): by inspecting it you can clearly see the paths followed by each individual user in your WOApplication. Of course, the WebObjects log file can still be parsed by any web server analysis tool.

Accessing Statistics

If your application has a WOStats page, you can look at the statistics that WOStatisticsStore gathers. WOStats is a reusable component stored in the WOExtensions framework (which WebObjects applications link to by default). While your application is running, you can access the WOStats page with a URL like the following:

```
http://localhost/cgi-bin/WebObjects/MyApp.woa/wa/WOStats
```

Note: You can access any component directly using a URL with this form.

Figure 37 shows a WOStats page.

Figure 37. WOStats Page

The source for the WOStats page is provided, along with the source for all of the components in the WOExtensions framework, in **/System/Developer/Examples/WebObjects/Source/WOExtensions** (**NeXT_ROOT\Developer\Examples\WebObjects\Source\WOExtensions** on Windows NT systems).

If you want access to statistics programmatically, send the WOStatisticsStore a **statistics** message. For example:

```
// WebScript
NSDictionary *myDict = [[[self application]
statisticsStore]
      statistics];

// Java
NSDictionary myDict =
  this.application().statisticsStore().statistics;
```

For a list of keys to this dictionary, see the WOStatisticsStore class specification in the *WebObjects Class Reference*.

Note that this dictionary is created on demand. It is costly to call this method repeatedly.

Recording Extra Information

There may be occasions when you want to have the WOStatisticsStore object record more information than it usually does. For example, it may be useful to know the value of a certain component variable each time the page is accessed. This is most easily accomplished by overriding **descriptionForResponse:inContext:** in your component and having it return the desired information.

For example, the HelloWorld example's Hello component could return the value of its **visitorName** instance variable along with the component name:

```
// WebScript Hello.m
- (NSString *)descriptionForResponse:(WOResponse
*)response inContext:(WOContext *)context {
      return [NSString stringWithFormat:@"%@/%@",
          [self name], visitorName];
}

//Java Hello.java
public String descriptionForResponse(WOResponse response,
WOContext context) {
      return new String(this.name() + visitorName);
}
```

The response component receives the **descriptionForResponse:inContext:** message after it receives the message **appendToResponse:inContext:**. The default implementation of **descriptionForResponse:inContext:** prints the page name. Unlike other methods invoked during the component action request-response loop, **descriptionForResponse:inContext:** is not sent to all components and dynamic elements on the page; it is sent only to the top-level response component.

Note that this method receives the response and context objects as arguments, just as **appendToResponse:inContext:** does. This means you can add such information as the HTTP header keys, or any other information recorded in these objects, to your description string.

This description string is then appended to the log file in the CLFF format as discussed in "Maintaining a Log File" (page 185).

When an application is deployed, a session object keeps in memory all these descriptions in the order their components were accessed by the session's user. When the session times out, three things happen:

- All its descriptions are written in the CLFF log file (if any).
- All its descriptions appear in the WOStats page under "Last User's Statistics."
- All its descriptions are added to all previous descriptions, and these totals appear in the WOStats page under "Detailed Statistics."

Error Handling

When an error occurs, WebObjects by default returns a page containing debugging information and displays that page in the Web browser. This information is useful when you're in the debugging phase, but when you're ready to deploy, you probably want to make sure that your users don't see such information.

The WOApplication class provides the following methods that you can override to show your own error page.

Method	Invoked When
handleSessionCreationErrorInContext:	The application needs to create a new session but can't.
handleSessionRestorationErrorInContext:	The application receives a request from a session that has timed out.
handlePageRestorationErrorInContext:	The application tries to access an existing page but cannot. Usually, this occurs when the user has backtracked beyond the limit set by **setPageCacheSize:** and **setPageRefreshOnBacktrackEnabled:** is NO.
handleException:inContext: (**handleException** in Java)	The application receives an exception; that is, any general type of error has occurred.

For example, the following implementation of **handleException:inContext:** returns a component named ErrorPage whenever an error occurs in the application.

```
public WOResponse handleException(java.lang.Throwable
anException, WOContext aContext) {
  WOResponse response = aContext.component().
      pageWithName("ErrorPage").generateResponse();

  return response;
}
```

Notice that this method, and all of the error-handling methods, return a WOResponse object instead of a WOComponent object.

Automatically Terminating an Application

Unless an application is very carefully constructed, the longer it runs, the more memory it consumes. As more memory is consumed, the server machine's performance begins to degrade. For this reason, you may find that performance is greatly improved if you occasionally stop an application instance and start a new one.

You can stop an application manually using the Monitor application (described in the online document *Serving WebObjects*). Or you can include code in the application to have it automatically terminate itself under certain conditions. Either way, you might want to turn on application auto-recovery in the Monitor application; that way, when the application dies, it automatically restarts.

- **Idle time.** To shut down an application after it has been idle for a given number of seconds, use WOApplication's **setTimeOut:** method, as illustrated here:

  ```
  public Application() {
    super();
    this.setTimeOut(2*60*60); //shut down if idle 2 hours.
    ...
  }
  ```

- **Running time.** You can have an application terminate itself after a specific amount of time has elapsed, regardless of whether it is idle or not using the **terminateAfterTimeInterval:** method. For example, the following application will terminate after 6 hours.

```
public Application() {
  super();
  this.terminateAfterTimeInterval(6*60*60);
  ...
}
```

  After the specified time limit has elapsed, **terminateAfterTimeInterval:** immediately stops all current processing. If any sessions are active, users may lose information.

- **Session count.** An application can also terminate if the number of active sessions falls below a certain number. Use **setMinimumActiveSessionsCount:** to set this number, and then send **refuseNewSessions:** to prevent the application from creating more sessions. For example, if you want to shut down your application after 6 hours but you want any current users to be able to end their sessions first, you might write the following code:

```
// WebScript Application.wos
id startDate;
- init {
  [super init];
  [self setMinimumActiveSessionCount:1];
  return self;
}

- sleep {
  if (!startDate) // get the start date from
statisticsStore
  {
      startDate = [[[self statisticsStore] statistics]
          objectForKey:@"StartedAt"];
  }
  // Compare start date to current date. If the difference
is
  // greater than 6 hours, refuse any new sessions.
  if ((([[NSDate date] timeIntervalSinceReferenceDate] -
      [startDate timeIntervalSinceReferenceDate]) >
21600)
  {
      [self refuseNewSessions:YES];
  }
}
```

When the application's active session count falls below the minimum of one session, it will terminate. Sending **refuseNewSessions:** guarantees that the active session count will eventually fall below the minimum.

Performance Tips

As more users access your application, you may become more concerned about its performance. Here are some suggestions about how to improve an application's performance.

Note: This section covers only programmatic ways to improve performance. Performance is affected by several factors, such as the load on your system, the amount of memory available, and whether the load is shared among multiple application instances. For information about other ways to improve performance, see the on-line document *Serving WebObjects*. In particular, you may want to check out the section "Testing Performance," which describes some tools you can use to do performance testing.

Cache Component Definitions

As described in the chapter "WebObjects Viewed Through Its Classes" (page 95), each component has a component definition consisting of the component's template (the result of parsing the **.html** and **.wod** files) and information about resources the component uses. If you cache component definitions, the **.html** and **.wod** files are parsed only once per application rather than every time they are changed. When caching is disabled, at each new instance of a component the time stamp of the **.html** and **.wod** are checked to see if the files have been modified. If they have, they're reloaded.

To cache component definitions, use WOApplication's **setCachingEnabled:** method:

```
public Application() {
  super();
  this.setCachingEnabled(true);
  ...
}
```

By default, this type of caching is disabled as a convenience for debugging. If component-definition caching is disabled and you're writing an entirely scripted application, you can change code in a scripted component and see the effects of that change without having to relaunch the application. You should always enable component-definition caching when you deploy an application, since performance improves significantly.

Instead of using **setCachingEnabled:**, you can also perform component-definition caching by setting the WOCachingEnabled user default either on the command line or using the **defaults** command.

```
HelloWorld -WOCachingEnabled YES

defaults write HelloWorld WOCachingEnabled YES
```

For more information on command-line options, see the online document *Serving WebObjects*.

Compile the Application

Applications written entirely in WebScript run more slowly than applications written in a compiled language such as Java or Objective-C. You may want to write in WebScript at first to speed the development cycle. Then, when you're ready to deploy, consider translating your WebScript code into a compiled language.

Control Memory Leaks

Make sure that all objects allocated by your application are being deallocated. The Yellow Box provides some tools that can help you check your code for memory leaks, such as the ObjectAlloc application. For more information, see ObjectAlloc's on-line help or the MallocDebug application.

Another way to control leaks is to have the application shut down and restart periodically, as described in the section "Automatically Terminating an Application" (page 190).

Limit State Storage

As the amount of memory required by an application becomes large, its performance decreases. You can solve this problem by limiting the amount of state stored in memory or by storing state using some other means, as described in the chapter "Managing State" (page 157). You can also set up the application so that it shuts down if certain conditions occur, as described in the section "Automatically Terminating an Application" (page 190).

One common mistake is neglecting to set a session time-out value. By default, sessions almost never expire, so the application may be using valuable memory to store sessions that users have long forgotten. When you set the session time-out value, if the session is idle for that amount of time, it terminates and its state is removed from memory. This is described in more detail in "Managing State."

Limit Database Fetches

Every database access that your application performs is a potential drag on performance. One easy way to limit trips to the database is to perform prefetching. For more information, see the chapter "Answers to Common Design Questions" in the *Enterprise Objects Framework Developer's Guide*.

If you have components that load images from a database, you should store the image in the WOResourceManager object's application-wide data cache if you know that the image is used more than once. To have the image stored in the cache, set the dynamic element's **key** attribute. When the key attribute is set, the image is stored in the cache under that key and WOResourceManager tries to retrieve the image from the cache before loading it from the database.

Limit Page Sizes

Be aware of the size of the HTML pages that you are downloading to the client machine. The larger the page, the more time it takes to download and draw. At first glance, your component's HTML might not seem unreasonably large; however, be sure you take into account the following:

- **HTML comments**. HTML comments consume unnecessary download time. The **WOIncludeCommentsInResponses** user default forces WebObjects to strip all HTML comments from all

generated components, reducing page sizes. Note that when using recording and playback, care must be take to both record and playback with the same value for this user default. Otherwise, differences will be found and playback will fail on correct pages.

- **Image files**. Does the page download a lot of images? If so, how large are these images? If image files are making the page too large, consider using GIF images, which are often much smaller than other formats, or consider limiting the number of images you use.

- **Reusable components**. Does the page include reusable components? If so, does the reusable component itself contain any reusable components? You must factor in the size of each component included and all of the image files that each component uses.

- **Repetitions**. If the page uses a repetition, how large it the array that the repetition iterates over? How large is the amount of HTML generated for each element in the array? In particular, if you have a repetition that generates a table row for each element in a large array, the page may take a long time to render.

  Consider implementing a batching display mechanism to display the information in the table. For example, if the array contains hundreds of entries, you might choose to only display the first 10 and provide a button that allows the user to see the next 10 entries. If the repetition is populated by a WODisplayGroup, you can use WODisplayGroup's **setNumberOfObjectsPerBatch:** method to set up this batching, and it then controls the display for you. For more information, see the WODisplayGroup class specification in the online book *WebObjects Framework Reference*.

Another reason a page might take a long time to draw is if the action that generates it takes a long time to perform. If this is the case, you can implement a WOLongResponsePage. WOLongResponsePage is an abstract subclass of WOComponent defined in the WOExtensions framework. When you use WOLongResponsePage, it displays a status page while the action is being performed and then displays your page upon completion. For an example of how to use WOLongResponsePage,

see the LongRequest example in **/System/Developer/Examples/WebObjects/ObjectiveC/LongRequest** (on Windows NT systems, **NEXT_ROOT\Developer\Examples\WebObjects\ObjectiveC\LongRequest**). Be sure to review HandlingLongRequests.rtf, which explains the workings of this example and the WOLongResponsePage component.

Installing Applications

When an application is ready to be deployed, do the following in Project Builder:

1. Click the inspector button to open the Build Attributes Inspector. In the Install in field, type **$(LOCAL_LIBRARY_DIR)/WebObjects/Applications**.

   If you're installing a framework, type **$(LOCAL_LIBRARY_DIR)/Frameworks**.

2. If your project contains web server resources, go to the **Makefile.preamble** file under Supporting Files. Uncomment the line that defines this macro:

   ```
   INSTALLDIR_WEBSERVER
   ```

3. In the Project Build panel, click the checkmark button to bring up the Build Options panel.

4. Choose "install" as the build target, and close the Build Options panel.

5. Click the Build button to start the build and installation process.

Assuming that your application is named **MyApp.woa**, this procedure installs these directories:

```
NEXT_ROOT/Local/Library/WebObjects/MyApp.woa
  MyApp[.exe]
  Resources/
  WebServerResources/
```

and (assuming that you have web server resources):

```
<DocRoot>/WebObjects/MyApp.woa
  WebServerResources/
```

While you can install the entire directory under **<*DocRoot*>/WebObjects**, doing so presents a security problem if you have scripted components. Any client can access any file under the document root; if you've installed scripted components there, you are exposing source code to outside users. Instead, you should install most of your application in ***NEXT_ROOT*/Local/Library/WebObjects/Applications** and install only the web server resources under the document root.

If you install the application in a subdirectory of **<*DocRoot*>/WebObjects**, you should set the WOApplicationBaseURL user default to point to the exact location of the application directory. (As with all user defaults, you can set WOApplicationBaseURL on the command line when launching the application or you can use the **defaults** command.) For example:

```
defaults write MyApp WOApplicationBaseURL
/WebObjects/MyWebApps
```

If you don't set WOApplicationBaseURL, your application can still run but cannot find image files and other web server resources. For more information, see *Serving WebObjects*.

Dynamically Loading Frameworks

When you run your WebObjects applications, you can specify extra frameworks to be dynamically loaded using the WOLoadFrameworks command-line option. You use this option as follows:

```
./HelloWorld -WOLoadFrameworks "(aFramework, bFramework, ...)"
```

The ".framework" extension is optional. If you specify a relative path to the framework (as in the example above), directories in your DYLD_FRAMEWORK_PATH are checked and, if the framework isn't found there, the directories specified by the NSProjectSearchPath are checked. Frameworks specified using WOLoadFrameworks are loaded just after the application class has been initialized. Each framework is loaded by invoking the framework's principal class' **init** method; you'll need to set the NSPrincipalClass in your framework's **CustomInfo.plist** in order to take advantage of this feature.

Part III

WebScript

Chapter 9

The WebScript Language

To speed the development cycle, you may want to write your application in a scripting language. The WebObjects scripting language is called WebScript. You can write all or part of your WebObjects application in WebScript.

This chapter tells you everything you need to know to use the WebScript language: its syntax and language constructs. WebScript is very similar to Objective-C. If you already know Objective-C, you may want to just scan this chapter and pay special attention to the section "WebScript for Objective-C Developers" (page 228), which describes the differences between WebScript and Objective-C.

WebScript is an object-oriented programming language. This chapter attempts to give you a brief introduction to the object-oriented concepts you'll need to know to follow the discussion, but it by no means teaches you object-oriented programming. To learn object-oriented programming, there are several books you can read, such as *Object-Oriented Programming and the Objective-C Language*.

Objects in WebScript

In WebScript, you work entirely with *objects*. An object is composed of data (called *instance variables*) and a set of actions that act upon that data (called *methods*). All variables that you declare are objects, and all values that a method returns are objects. There are no simple data types like **int** or **char** in C.

Each file that you write in WebScript defines an object. The definition of an object is called a *class*. A class specifies the instance variables that will be created for each object and the methods that the object will be able to perform. To create an object, you create an *instance* of a class (or *instantiate* a class).

For example, the following is a typical WebScript file.

```
id number, aName;

- awake {
   if (!number) {
      number = [[self application] visitorNum];
      number++;
      [[self application] setVisitorNum:number];
  }
  return self;
}

- recordMe {
  if ([aName length]) {
      [[self application] setLastVisitor:aName];
      [self setAName:@""]; // clear the text field
  }
}
```

Instance variables are declared at the top of the script file. In the example above, **number** and **aName** are instance variables of type **id**. An object's behavior is defined by its *methods*. **awake** and **recordMe** are examples of methods.

When you define a new class, you *subclass* an existing class. Subclassing gives you access not only to the variables and methods that you explicitly define but also to the variables and methods defined for the existing class (called the *superclass*). As you learned in the chapter "What Is a WebObjects Application?" (page 17), WebObjects applications can contain four kinds of script files: a component script inside a **.wo** directory, an application script, a session script, and a direct action script. These four kinds of scripts create subclasses of the WebObjects classes WOComponent, WOApplication, WOSession, and WODirectAction, respectively. As you'll learn later, you can also subclass other classes in WebScript, but doing so is rare.

WebScript Language Elements

This section describes WebScript language elements. WebScript is based on Objective-C, which in turn is based on C. If you are familiar with C, most of the statements, operators, and reserved words will be very familiar to you. Because WebScript is an object-oriented programming language and because all variables are objects, there is some difference

in the way you declare variables and there is added syntax for working with objects.

Variables

To declare a variable in WebScript, use the syntax:

```
id myVar;
id myVar1, myVar2;
```

In these declarations, **id** is a data type. The **id** type is a reference to any object—in reality, a pointer to the object's data (its instance variables). Like a C function or an array, an object is identified by its address; thus, all variables declared in WebScript are pointers to objects. In the examples above, **myVar1** and **myVar2** could be any object: a string, an array, or a custom object from your application.

Note: Unlike C, no pointer manipulation is allowed in WebScript.

Instead of using **id**, you can specifically refer to the class you want to instantiate using this syntax:

className **variableName*;

For example, you could specify that a variable is an NSString object using this syntax:

```
NSString *myString1;
NSString *myString2, *myString3;
```

For more information on specifying class names in variable declarations, see the section "Data Types" (page 214).

In WebScript, there are two basic kinds of variables: local variables and instance variables. You declare instance variables at the top of the file, and you declare local variables at the beginning of a method or at the beginning of a block construct (such as a **while** loop). The following shows where variables can be declared:

```
id instanceVariable; // An instance variable for this
class.

- aMethod {
  id localVariable1; // A local variable for this method.

  while (1) {
      NSString *localVariable2; // A local variable for
this block.
  }
}
```

Variables and Scope

Each kind of variable has a different scope and a different lifetime. Local variables are only visible inside the block of text in which they are declared. In the example above, **localVariable1** is declared at the top of a method. It is accessible within the entire body of that method, including the **while** loop. It is created upon entry into the method and released upon exit. **localVariable2**, on the other hand, is declared in the **while** loop construct. You can only access it within the curly braces for the **while** loop, not within the rest of the method.

The scope of an instance variable is object-wide. That means that any method in the object can access any instance variable. You can't directly access an instance variable outside of the object that owns it; you must use an accessor method instead. See "Accessor Methods" (page 210).

The lifetime of an instance variable is the same as the lifetime of the object. When the object is created, all of its instance variables are created as well and their values persist throughout the life of the object. Instance variables are not freed until the object is freed.

As you learned in the chapter "Common Methods" (page 61):

- A WOApplication is created when you started a WebObjects application
- A WOSession is created each time a different user accesses that application during the component action request-response loop
- A WOComponent is created the first time a user accesses that page in the application
- A WODirectAction is created at the beginning of each direct action request-response loop cycle.

Thus, the variables you declare at the top of the application script (**Application.wos**) exist as long as the application is running. The variables you declare at the top of the session script (**Session.wos**) exist for the length of one session. As new users access your application, new sessions are created, so new copies of the session's instance variables are created too. These copies of instance variables are private to each session; one session does not know about the instance variables of another session. As sessions expire, their instance variables are freed. The variables you declare at the top of a component script are created and released as that component is created and released. Finally, the variables you declare at the top of a direct action script (**DirectAction.wos**) are created at the beginning of the direct action request-response loop cycle and released at the end of the cycle.

Note: Just how often a particular component object is created depends on whether the application object is caching pages. For more information, see "WebObjects Viewed Through Its Classes" (page 95).

Assigning Values to Variables

You assign values to variables using the following syntax:

```
myVar = aValue;
```

A value can be assigned to a variable at the time it is declared or after it is declared. For example:

```
NSNumber *myVar1;
id myVar2 = 77;

myVar1 = 76;
```

The value you assign to a variable can be either a constant or another variable. For example:

```
// assign another variable to a variable
myVar = anotherVar;
// assign a string constant to a variable
myString = @"This is my string.";
```

Note: The `//` syntax denotes a comment.

You can assign constant values to objects of four of the most commonly used classes in WebScript: NSNumber, NSString, NSArray, and NSDictionary. These classes are defined in the Foundation framework.

To learn how to initialize objects of all other classes, see "Creating Instances of Classes" (page 212) in this chapter.

NSNumber is the easiest class to initialize. You just assign a number to the variable, like this:

```
NSNumber *myNumber = 77;
```

For the remaining three classes, WebScript provides a convenient syntax for initializing constant objects. In such an assignment statement, the value you're assigning to the constant object is preceded by an at sign (@). You use parentheses to enclose the elements of an NSArray and curly braces to enclose the key-value pairs of an NSDictionary. The following are examples of how you use this syntax to assign values to constant NSString, NSArray, and NSDictionary objects in WebScript:

```
myString = @"hello world";
myArray = @("hello", "goodbye");
myDictionary = @{"key" = 16};
anotherArray = @(1, 2, 3, "hello");
aDict = @{ "a" = 1; "b" = "hello world"; "c" = (1,2,3);
     "d" = { "x" = 1; "r" = 2 }};
```

The following rules apply when you use this syntax to create constant objects:

- The value you assign must be a constant (that is, it can't include variables). For example, the following is not allowed:

  ```
  // This is not allowed!!
  myArray = @("hello", aVariable);
  ```

- You shouldn't use @ to identify an NSString, NSArray, or NSDictionary inside the value being assigned. For example:

  ```
  // This is not allowed!!
  myDictionary = @(@"value" = 3);

  // Do this instead
  myDictionary = @("value" = 3);
  ```

For more information on NSNumber, NSString, NSDictionary, and NSArray, see the chapter "WebScript Programmer's Quick Reference to Foundation Classes" (page 233).

Methods

To define a new method, simply put its implementation in the script file. You don't need to declare it ahead of time. For example:

```
- recordMe {
  if ([aName length]) {
      [[self application] setLastVisitor:aName];
      [self setAName:@""]; // clear the text field
  }
}
```

Methods can take arguments. To define a method that takes arguments, you place the argument name after a colon (:). For example, the following method takes two arguments. It adds the two arguments together and returns the result:

```
- addFirstValue:firstValue toSecondValue:secondValue {
  id result;
  result = firstValue + secondValue;
  return result;
}
```

The strings that appear to the left of the colons are part of the method name. The method above is named **addFirstValue:toSecondValue:**. It takes two arguments, which it calls **firstValue** and **secondValue**.

If you want, you can add type information for the return values and parameter values. For example, the following method, **subtractFirstValue:fromSecondValue:**, subtracts one number from another and returns the result:

```
- (NSNumber *)subtractFirstValue:(NSNumber *)firstValue
fromSecondValue:(NSNumber *)secondValue {
  NSNumber *result;
  result = secondValue - firstValue;
  return result;
}
```

In these examples, note that type information is optional. When there is no type, **id** is assumed. Also note that both example methods return a value, stored in **result**. If a method doesn't return a meaningful value, you don't have to include a return statement (a return value of **nil** is returned for methods without an explicit return statement).

Invoking Methods

When you want an object to perform one of its methods, you send the object a *message*. In WebScript, message expressions are enclosed in square brackets:

[*receiver message*]

The *receiver* is an object, and the *message* tells it what to do. For example, the following statement tells the object **aString** to perform its **length** method, which returns the string's length:

```
[aString length];
```

When the method requires arguments, you pass values by placing them to the right of the colon. For example, to invoke the method **addFirstValue:toSecondValue:** shown previously, you would do this:

```
three = [aComponent addFirstValue:2 toSecondValue:1];
```

One message can also be nested inside another. Here the **description** method returns the string representation of an NSCalendarDate object **myDate**, which is then appended to **aString**. The resulting string is assigned to **newString**:

```
newString = [aString stringByAppendingString:[myDate
description]];
```

As another example, here the array **anArray** returns an object at a specified index. That object is then sent the **description** message, which tells the object to return a string representation of itself, which is assigned to **desc**:

```
id desc = [[anArray objectAtIndex:anIndex] description];
```

Accessor Methods

As stated previously, you can access any instance variable within any method declared in the same object. If you need to access a variable in a different object, you must send a message to that object.

Accessor methods are methods that other objects can use to access an object's instance variables. When you declare an instance variable, WebScript automatically defines two accessor methods: one to retrieve the instance variable's value, and one to change the value.

For example, suppose an **Application.wos** script declared this instance variable, which keeps track of the number of visitors:

```
id visitorNum;
```

When WebScript parses this file, it sees this declaration and implicitly defines two methods that work like this:

```
- visitorNum {
      return visitorNum;
}

- setVisitorNum:newValue {
      visitorNum = newValue;
}
```

(You don't see these methods in the script file.) The **Main.wos** script can access the application's **visitorNum** variable using these statements:

```
number = [[self application] visitorNum];
...
[[self application] setVisitorNum:number];
```

Note: self is a keyword that represents the current object. For more information, see "Reserved Words" (page 218).

You can also access an instance variable declared in one component script from another component script. This is something you commonly do right before you navigate to a new page, for example:

```
id anotherPage = [[self application]
pageWithName:@"Hello"];
[anotherPage setNameString:newValue];
```

The current script uses the statement `[anotherPage setNameString:newValue];` to set the value of **nameString**, which is declared in the page named Hello.

Sending a Message to a Class

Usually, the object receiving a message is an *instance* of a class. For example, in this statement the variable **aString** is an instance of the class NSString:

```
[aString length];
```

You can also send messages to a class. You send a class a message when you want to create a new instance of that class. For example this statement tells the class NSString to invoke its **stringWithString:** method, which returns an instance of NSString that contains the specified string:

```
aString = [NSString stringWithString:@"Fred"];
```

Note that a class is represented in a script by its corresponding class name—in this example, NSString.

In WebScript, the classes you use include both *class methods* and *instance methods*. Most class methods create a new instance of that class, whereas instance methods provide behavior for instances of the class. In the following example, a class method, **stringWithFormat:**, is used to create an instance of a class, NSString. Instance methods are then used to operate on the instance **myString**:

```
// Use a class method to create an instance of NSString
NSString *myString = [NSString
  stringWithFormat:@"The next word is %@", word];

// Use instance methods to operate on the instance
myString
length = [myString length];
lcString = [myString lowercaseString];
```

In an Objective-C class definition, class methods are preceded by a plus sign (+), while instance methods are preceded by a minus sign (-). You cannot declare class methods in WebScript, but you can use the Objective-C class methods defined for any class.

Creating Instances of Classes

The section "Variables" (page 205) told you how to declare variables, which represent objects. Before you use a variable, you must first create the object, or *instantiate* the class that defines the object. In WebScript, there are two different ways to create objects. The first approach, which applies only to the most commonly used classes, is to initialize with constant objects as described in the section "Assigning Values to Variables" (page 207). The second approach, which applies to all classes, is to use creation methods.

All classes provide creation methods that you can use to create an instance of that class. Depending on the class and the particular creation method, the instances of the class you create are either *mutable* (modifiable) or *immutable* (constant). Usually, the instances of a class are always mutable or always immutable. (You must read the specifications in the *Foundation Framework Reference* to find out if a particular class is immutable or mutable.) However, some classes, including NSString, NSArray, and NSDictionary provide both mutable and immutable forms, and you can choose which one to create. It's best to create immutable objects wherever possible. Use mutable objects only if you need to change its value after you initialize it.

Here are some examples of using creation methods to create mutable and immutable NSString, NSArray, and NSDictionary objects:

```
// Create a mutable string
string = [NSMutableString stringWithFormat:@"The string
is %@", aString];

// Create an immutable string
string = [NSString stringWithFormat:@"The string is %@",
aString];

// Create a mutable array
array = [NSMutableArray array];
anotherArray = [NSMutableArray
arrayWithObjects:@"Marsha", @"Greg",
  @"Cindy", nil];

// Create an immutable array
array = [NSArray arrayWithObjects:@"Bobby", @"Jan",
@"Peter", nil];

// Create a mutable dictionary
dictionary = [NSMutableDictionary dictionary];
// Create an immutable dictionary
id stooges = [NSDictionary
     dictionaryWithObjects:@("Mo", "Larry", "Curley")
     forKeys:@("Stooge1", "Stooge2", "Stooge3")];
```

The following examples show how you can create and work with NSCalendarDates, which are always immutable:

```
// Using the creation method date, create an
NSCalendarDate instance
// 'now' that contains the current date and time
now = [NSCalendarDate date];

// Return a string representation of 'now' using a format
string
dateString = [now descriptionWithCalendarFormat:@"%B %d,
%Y"];

// Using the creation method dateWithString:, create an
NSCalendarDate
// instance 'newDate' from 'dateString'
newDate = [NSCalendarDate dateWithString:dateString
  calendarFormat:@"%B %d, %Y"];

// Return a new date in which newDate's day field is
decremented
date = [newDate addYear:0 month:0 day:-1 hour:0 minute:0
second:0];
```

For a detailed discussion of these classes and a more complete listing of methods, see the chapter "WebScript Programmer's Quick Reference to Foundation Classes" (page 233).

Data Types

Several of the examples in this chapter show how you can specify a data type when you define a method or variable. For example:

```
NSString *myString = @"This is my string.";

- (NSString *)appendString:(NSString *)aString {
      NSString *returnString = [NSString
          stringWithFormat:@"%@ %@", myString, aString];
      return returnString;
}
```

Explicitly specifying a class in a variable or method declaration is called *static typing*.

Because variables and return values are always objects, the only supported data types are classes. You can specify any class that your application recognizes when you statically type a variable or a method. For example, each component in your application is a class, so you can do this:

```
CarPage *carPage = [[self application]
pageWithName:"CarPage"];
```

Also, the default application executable used to run your application contains the definitions of classes from the Foundation, WebObjects, and Enterprise Objects frameworks, so these declarations are valid:

```
NSString *myString; //Foundation classes
WOContext *theContext; //WebObjects classes
EOEditingContext *editingContext; //Enterprise Objects
classes
```

Plus, if you're writing a component that uses database access, your application has an EOModel file that translates tables in your database into objects. You can specify any entity named in that model file as a class. For example:

```
Movies *moviesEntity; //Entities from your eomodel.
```

Static typing is supported so that WebObjects Builder can correctly parse your script file and help you decide which variables you can correctly bind to certain dynamic elements. (For more information on this, see the online book *WebObjects Tools and Techniques*.) As far as WebScript is concerned, all variables are of type **id**.

Note: WebObjects performs absolutely no type checking. The following is valid WebScript:

```
NSNumber *aNumber = @"Wait! I'm a string, not a number!";
NSString *aString = 1 + 2;
```

Statements and Operators

WebScript supports most of the same statements and operators that C supports, and they work in much the same way. This section describes only the differences between statements and operators in C and statements and operators in WebScript.

Control Flow Statements

WebScript supports the following control-flow statements:

```
if
else
for
while
break
continue
return
```

Arithmetic Operators

WebScript supports the arithmetic operators +, - , /, *, and %. The rules of precedence in WebScript are the same as those for the C language. You can use these operators in compound statements such as:

```
b = (1.0 + 3.23546) + (((1.0 * 2.3445) + 0.45 + 0.65) -
3.2);
```

Logical Operators

WebScript supports the negation (!), AND (&&), and OR (||) logical operators. For the most part, you can use these operators as you would in the C language, for example:

```
if ( !(!a || a && !i) || (a && b) && (c || !a && (b+3)) )
i = 0;
```

As in C, Boolean expressions *short-circuit*. For example, in this statement:

```
(!a || (a && !i))
```

The expression is only evaluated up to the point where the outcome can be surmised. That is, if **!a** is true, the other side of the || expression is not evaluated because the || operator only requires one side of the statement to be true. Likewise, if the && statement is evaluated and **a** is false, the second half of the && expression is not evaluated because the && operator does require both sides to be true.

Relational Operators

WebScript supports the relational operators <, <=, >, >=, ==, and !=. In WebScript these operators behave as they do in C.

In WebScript, relational operators are only reliable on NSNumbers. If you try to use them on any other class of object, you may not receive the results you expect.

For example, the following statement compares the addresses stored in the two string variables. If both point to the same address, the strings are equal. If they point to different addresses, even if the strings have identical contents, the statement will be false.

```
NSString *string1, *string2;
if (string1 == string2) // compares pointer values.
```

To compare two strings, or two of any other type of object, use the **isEqual:** method.

```
NSString *string1, *string2;
if ([string1 isEqual:string2]) // compares values of
objects
```

Note: The == operator may fool you at times by appearing to test equality of objects other than NSNumbers. This is because constant values are uniqued within a script file. That is, if you do the following, WebScript stores the constant string objects in one location and has both variables point to the same string constant.

```
NSString *string1 = @"This is a string";
NSString *string2 = @"This is a string";
```

If you compare these two variables, it may appear that WebScript compares the values they point at, but in reality, it is testing the pointer addresses.

```
NSString *string1 = @"This is a string";
NSString *string2 = @"This is a string";

if (string1 == string2) {
  //This code gets executed because string1 and string2
  //point to the same memory address.

if ([string1 isEqual:string2])
  //This code gets executed because string1 and string2
  //have the same contents.
```

Increment and Decrement Operators

WebScript supports the preincrement (++) and predecrement (--) operators. These operators behave as they do in the C language, for example:

```
// Increment myVar and then use its value
// as the value of the expression
++myVar;
```

The postincrement and postdecrement operators are not supported. They behave like the preincrement and predecrement operators. For example:

```
// WATCH OUT!! Probably not what you want.
i = 0;
while (i++ < 1) {
  //this loop never gets executed because i++ is a
preincrement.
}
```

Reserved Words

WebScript includes the following reserved words:

```
if
else
for
while
id
break
continue
self
super
nil
YES
NO
```

Three reserved words are special kinds of references to objects: **self**, **super**, and **nil**. You can use these reserved words in any method.

self refers to the object (the WOApplication object, the WOSession object, the WOComponent object, or the WODirectAction object) associated with a script. When you send a message to **self**, you're telling the object associated with the script to perform a method that's implemented in the script. For example, suppose you have a script that

implements the method **giveMeARaise**. From another method in the same script, you could invoke **giveMeARaise** as follows:

```
[self giveMeARaise];
```

This tells the WOApplication, WOSession, WOComponent, or WODirectAction object associated with the script to perform its **giveMeARaise** method.

When you send a message to **self**, the method doesn't have to be physically located in the script file. Remember that part of the advantage of object-oriented programming is that a subclass automatically implements all of its superclass's methods. For example, WOComponent defines a method named **application**, which retrieves the WOApplication associated with this component. Thus, you can send this message in any of your components to retrieve the application object:

```
[self application]
```

WOComponent also defines a session method, so you can do this to retrieve the current session:

```
[self session]
```

Sometimes, you actually do want to invoke the superclass's method rather than the current object's method. For example, when you initialize an object, you should always give the superclass a chance to perform its initialization method before the current subclass. To do this, you send the **init** message to **super**, which represents the superclass of the current object.

```
- init {
      [super init];
      // my initialization.
      return self;
}
```

The **nil** keyword represents an empty object. Any object before it is initialized has the value **nil**. **nil** is similar to a null pointer in C. For example, to test whether an object has been allocated and initialized, you do this:

```
if (myArray == nil) //myArray hasn't been initialized.
```

This next statement also tests to see if **myArray** is equal to **nil**:

```
if (!myArray) //myArray hasn't been initialized.
```

"Modern" WebScript Syntax

WebScript supports two syntax styles. The style that you've been reading about up until now is "classic" syntax, which is based on the syntax of Objective-C. If you're more familiar with languages such as Visual Basic or Java, you may be more comfortable with the alternate syntax style, called "modern" syntax.

The differences between classic and modern WebScript syntax are summarized below:

- Method Definition

  Classic:

  ```
  - submit {
      // <body>
  }
  ```

  Modern:

  ```
  function submit() {
      // <body>
  }
  ```

- Method Definition With Arguments

  Classic:

  ```
  - takeValuesFromRequest:(WORequest *)request
      inContext:(WOContext *)context {
  // <body>
}
  ```

  Modern:

  ```
  //Note: no static typing allowed.
  function takeValues(fromRequest:= request inContext:=
context){
      // <body>
  }
  ```

- Method Invocation — No Argument

  Classic:

  ```
  [self doIt];
  ```

  Modern:

  ```
  self.doIt();
  ```

- Method Invocation — One Argument

  Classic:

  ```
  [guests addObject:newGuest];
  ```

  Modern:

  ```
  guests.addObject(newGuest);
  ```

- Method Invocation — Two or More Arguments

  Classic:

  ```
  [guests insertObject:newGuest atIndex:anIndex];
  ```

  Modern:

  ```
  guests.insert(object := newGuest, atIndex := anIndex);
  ```

Note that in this last example the left parenthesis should occur at a break between words when the modern message maps to an existing Objective-C method (which, of course, follows classic WebScript syntax). When WebScript transforms modern to classic syntax internally, it capitalizes this character before concatenating the keywords of the selector. Thus, any of the following is correct:

```
super.takeValuesFrom(request := request, inContext :=
context);
super.takeValues(fromRequest := request, inContext :=
context);
super.take(valuesFromRequest := request, inContext :=
context);
```

If you choose to use modern WebScript, there is another important caveat: You cannot have methods with variable-length argument lists. Thus, you cannot use methods such as **logWithFormat:** and **stringWithFormat:**. You can, however, mix classic and modern WebScript in the same script file.

Advanced WebScript

In WebScript, you create subclasses of WOComponent, WOSession, WOApplication, and WODirectAction and you use declared variables of classes defined in the Foundation Framework, Enterprise Objects Framework, or the WebObjects Framework. For most WebScript applications, this is sufficient.

Sometimes, however, you might want to subclass some other class or at least extend the behavior of that class without having to resort to compiled code. For these cases, WebScript allows you to do two things: create a *scripted class*, which is a scripted subclass of anything other than WOApplication, WOSession, WOComponent, or WODirectAction; or create a *category*, which is a way to extend the behavior of a class without subclassing it.

Scripted Classes

The syntax for creating a scripted class is very similar to the syntax for creating a class in Objective-C. The instances of a class created in such a manner behave like any other Objective-C object.

To create a scripted class, you specify the class interface in an **@interface...@end** block and the class implementation in an **@implementation...@end** block. To ensure the class is loaded properly, the scripted class code should be in its own **.wos** file, with the class name

matching the filename. The following example is in a file named **Surfshop.wos**:

```
@interface Surfshop:NSObject {
  id name;
  NSArray *employees;
}
@end

@implementation Surfshop
- (Surfshop *)initWithName:aName employees:theEmployees {
  name = [aName copy];
  employees = [theEmployees retain];
  return self;
}
@end
```

Do not use separate files for the **@interface** and **@implementation** blocks. They must both be in the same file.

To use the class, you locate it in the application, load it, and then allocate and initialize instances using the class object. Here's an example:

```
NSMutableArray *allSurfshops;
- init {
  id scriptPath;
  id surfshopClass;

  [super init];
  scriptPath = [[[self application] resourceManager]
      pathForResourceNamed:@"Surfshop.wos"
inFramework:nil
      languages:nil];
  surfshopClass = [[self application]
      scriptedClassWithPath:scriptPath];
  allSurfshops = [NSMutableArray array];
  [allSurfshops addObject:[[[surfshopClass alloc]
initWithName:
      "Banana Surfshop" employees:@("John Popp", "Jenna de
Rosnay")]
      autorelease]];
  [allSurfshops addObject:[[[surfshopClass alloc]
initWithName:
      "Rad Swell" employees:@("Robby Naish", "Nathalie
Simon")]
      autorelease]];

  return self;
}
```

Categories

A category is a set of methods you add to an existing class. You can add a category to any custom or WebObjects-provided Objective-C class. Because the methods added by the category become part of the class type, you can invoke them on any object of that type within an application. That is, you don't have to instantiate a special subclass.

To create a category, you must implement it within an **@implementation** block, which is terminated by the **@end** directive. Place the category name in parentheses after the class name.

The following example is a simple category of WORequest that gets the sender's Internet e-mail address from the request headers ("From" key) and returns it (or "None").

```
@implementation WORequest(RequestUtilities)
- emailAddressOfSender {
  NSString *address = [self headerForKey:@"From"];
  if (!address) address = @"None";
  return address;
}
@end
```

Elsewhere in your WebScript code, you invoke this method on WORequest objects just as you do with any other method of that class. Here's an example:

```
- takeValuesFromRequest:request inContext:context {
  [super takeValuesFromRequest:request
inContext:context];
  [self logWithFormat:@"Email address of sender: %@",
      [request emailAddressOfSender]];
}
```

Note: If your category is at the end of a scripted component, you must restart your app each time you change that file.

Exception Handling

WebScript supports exception handling in a fashion very similar to that of Objective-C. An exception is a special condition that interrupts the normal flow of program execution. Each application can interrupt the program for different reasons. For example, one application might interpret saving a file in a directory that's write-protected as an exception. In this sense, the exception is equivalent to an error. Another application

might interpret the user's input as an exception, perhaps as an indication that a long-running process should be aborted.

Raising an Exception

Once an exception is detected, it must be propagated to code that will handle it, called the exception handler. This entire process of handling an exception is referred to as “raising an exception.” Exceptions are raised by instantiating an NSException object and sending it a raise message.

NSException objects provide:

- a name - a short string that is used to uniquely identify the exception.
- a reason - a longer string that contains a “human-readable” reason for the exception.
- userInfo - a dictionary used to supply application-specific data to the exception handler. For example, if the return value of a method causes an exception to be raised, you could pass the return value to the exception handler through userInfo.

Handling an Exception

Where and how an exception is handled depends on the context where the exception was raised. In general, a raise message is sent to an NSException object within the domain of an exception handler. An exception handler is contained within a control structure created by the keywords NS_DURING, NS_HANDLER, and NS_ENDHANDLER, as shown here:

```
.
.
NS_DURING
  [some code];
  [some more code];
NS_HANDLER
  [exception handler code];
  [more exception handler code];
NS_ENDHANDLER
.
.
```

The section of code between NS_DURING and NS_HANDLER is the *exception handling domain*; the section between NS_HANDLER and NS_ENDHANDLER is the *local exception handler*. The normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if an exception is raised. Sending a **raise** message to an exception object causes program control to jump to the first executable line following NS_HANDLER.

Although an exception can be raised directly within the exception handling domain, exceptions more often arise indirectly from a method invoked from the domain. No matter how deep in a call sequence the exception is raised, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in the next section). In this way, exceptions raised at a low level can be caught at a high level.

You may leave the exception handling domain (the section of code between NS_DURING and NS_HANDLER) by:

- Raising an exception.
- Calling NS_VALUERETURN(), which returns a value from the DURING block.
- Calling NS_VOIDRETURN, which returns void from the DURING block.
- "Falling off the end."

"Falling off the end" is simply the normal path of execution-after all statements in the exception handling domain are executed, execution continues on the line following NS_ENDHANDLER.

Note: You can't use return to exit an exception handling domain—errors will result.

Nested Exception Handlers

Exception handlers can be nested so that an exception raised in an inner domain can be treated by the local exception handler and any number of encompassing exception handlers. The following diagram illustrates the use of nested exception handlers, and is discussed in the text that follows.

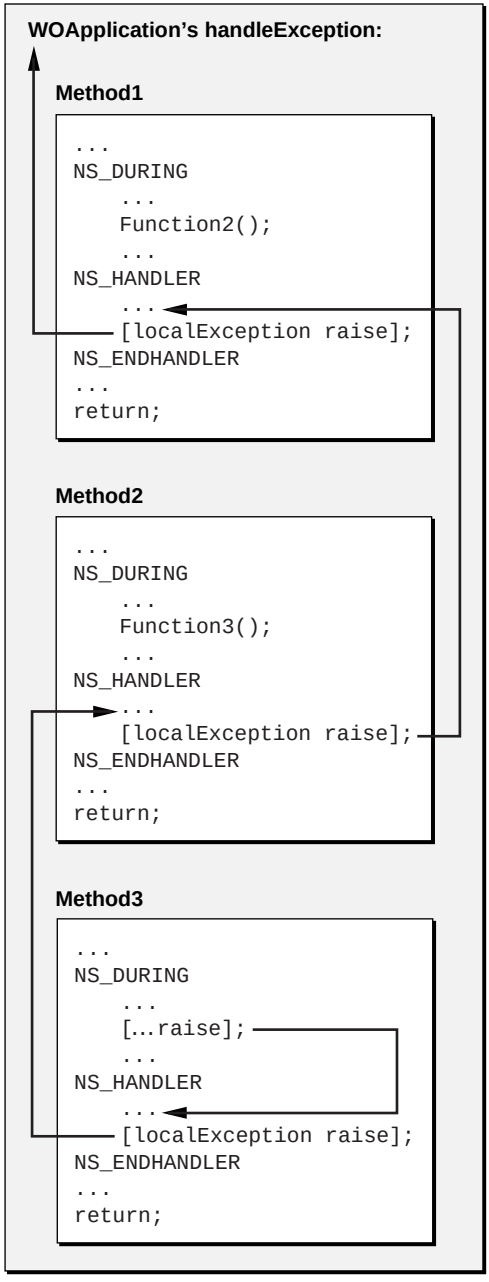

Figure 38. Nested Exception Handlers

An exception raised within Method3's domain causes execution to jump to its local exception handler. In a typical application, this exception handler checks the object **localException** to determine the nature of the exception. For exception types that it recognizes, the local handler responds and then may send **raise** to **localException** to pass notification of the exception to the handler above, the handler in Method2. (An exception that's re-raised appears to the next higher handler just as if the initial exception had been raised within its own exception handling domain.) Method2's exception handler does the same and then re-raises the exception to Method1's handler. Finally, Method1's handler re-raises the exception. Since there's no exception handling domain above Method1, the exception is transferred to the uncaught exception handler as described below.

WebScript for Objective-C Developers

WebScript uses a subset of Objective-C syntax, but its role within an application is significantly different. The following table summarizes some of the differences.

Objective-C	WebScript
Is compiled	Is interpreted
Supports primitive C data types	Supports only the class type
Performs type checking at compiled time	Never performs type checking
Requires method prototyping	Doesn't require method prototyping (that is, you don't declare methods before you use them)
Usually involves a .h and a .m file	Stands alone (unless part of a component)
Supports all C language features	Has limited support for C language features; for example, doesn't support structures, pointer, enumerations, or unions
Methods not declared to return void must include a return statement	Methods aren't required to include a return statement. Such methods return **nil**.

Objective-C	WebScript
Has preprocessor support	Has no preprocessor support—that is, doesn't support the #define, #import, or #include statements
Uses reference counting to determine when to release instance variables	Automatically retains all instance variables for the life of the object that owns them. Automatically releases instance variables when the object is released.

Here are some of the more subtle differences between WebScript and Objective-C:

- You don't need to retain instance variables in the **init** method or release them in the **dealloc** method. In general, you never have to worry about releasing variables. One exception: if you perform a **copy** or a **mutableCopy** on an object, you must release that copy.
- Categories must not have an **@interface** declaration in WebScript.
- The @ in WebScript signifies the initialization of an NSString, NSDictionary, or NSArray.
- Instead of using operators like **@selector**, you simply enclose the selector in double quotes ("").
- Certain operators from the C language aren't available in WebScript, notably the postdecrement, postincrement, and cast operators.
- Boolean expressions never short-circuit.

Of course, the most significant difference between Objective-C and WebScript is that in WebScript, all variables must be objects. Some of the less obvious implications of this are:

- You can't use methods that take non-object arguments (unless those arguments are integers or floats, which WebScript converts to NSNumbers). For example, in WebScript the following statement is invalid:

```
// NO!! This won't work.NSRange is a structure.
string = [NSString substringWithRange:aRange];
```

- You can only use the “at sign” character (@) as a conversion character with methods that take a format string as an argument:

```
// This is fine.
[self logWithFormat:@"The value is %@", myVar];

// NO!! This won't work. It prints the address of var1.
[self logWithFormat:@"The values are %d and %s", var1,
var2];
```

- You need to substitute integer values for enumerated types.

  For example, suppose you want to compare two numeric values using the enumerated type NSComparisonResult. This is how you might do it in Objective-C:

```
result = [num1 compare:num2];
if(result == NSOrderedAscending) /* This won't work in
WebScript */
        /* num1 is less than num2 */
```

  But this won't work in WebScript. Instead, you have to use the integer value of NSOrderedAscending, as follows:

```
result = [num1 compare:num2];
if(result == -1)
        /* num1 is less than num2 */
```

  For a listing of the integer values of enumerated types, see the “Types and Constants” section in the *Foundation Framework Reference*.

Accessing WebScript Methods From Objective-C Code

As stated previously, you can mix WebScript and Objective-C code. Often, programmers use WebScript for component logic, and then supply the bulk of the application (the “business logic”) in compiled code.

To access Objective-C code from a WebScript file, you simply use the Objective-C class like any other class:

```
id myObject = [[MyCustomObjCClass alloc] init];
```

To access a WebScript object from Objective-C code, you simply get the object that implements the method and send it a message. If you're accessing a method in the application script, you can use this WOApplication method to access the object:

```
[[WOApplication application] applicationScriptMethod];
```

To avoid compiler warnings when accessing WebScript objects from Objective-C, create a header file for the scripted component as though it were compiled. This step isn't strictly required, of course—your code will still build, you'll just get warnings.

Chapter 10

WebScript Programmer's Quick Reference to Foundation Classes

As you learned in the previous chapter, when you write an application in WebScript, all values are objects, even simple values such as numbers or strings. The objects that represent strings, numbers, arrays, dictionaries, and other basic constructs are defined by classes in the Foundation framework. You use classes from the Foundation framework in virtually all WebScript applications.

This chapter gives you an overview of the Foundation framework classes most commonly used in WebScript. More detailed descriptions are provided by the class specifications in the *Foundation Framework Reference*. However, not all methods in these class specifications are available in WebScript. For example, some classes define methods that take structures as arguments or return structures, but because structures are not supported in WebScript, you cannot use these methods. For more information, see "WebScript for Objective-C Developers" (page 228) in the previous chapter.

Foundation Objects

This section provides an overview of some of the topics, techniques, and conventions you use when programming with Foundation objects.

Representing Objects as Strings

You can obtain a human-readable string representation of any object by sending it a **description** message. This method is particularly useful for debugging. In some cases, the string returned from **description** contains only the class name of the object that received the message (the *receiver*). Most objects, however, provide more information. For class-specific details, see the **description** method descriptions later in this chapter.

Mutable and Immutable Objects

Some objects are immutable; that is, once they are created, they can't be modified. Other objects are mutable. They can be modified at any time. When you create an object, you can often choose to create it as either immutable or mutable. Three kinds of objects discussed in this

chapter—strings, arrays, and dictionaries—have both immutable and mutable versions.

It's best to use immutable objects whenever possible. Use a mutable object only if you need to modify its contents after you create it.

Determining Equality

You can determine if two objects are equal using the **isEqual:** method. This method returns YES if the receiver of the message and the specified object are equal, and NO otherwise. The definition of equality depends on the object's type. For example, array objects define two arrays as equal if they contain the same contents. For more information, see the **isEqual:** method descriptions later in this chapter.

Writing to and Reading From Files

Strings, arrays, and dictionaries—three of the classes discussed in this chapter—provide methods for writing to and reading from files. The method **writeToFile:atomically:** writes a textual description of the receiver's contents to a specified path name, and corresponding class-specific creation methods—**stringWithContentsOfFile:**, **arrayWithContentsOfFile:**, and **dictionaryWithContentsOfFile:**—create an object from the contents of a specified file.

For example, the following code excerpt reads the contents of an error log stored in a file, appends a new error to the log, and saves the updated log to the same file:

```
id errorLog = [NSString
stringWithContentsOfFile:errorPath];
id newErrorLog = [errorLog stringByAppendingFormat:@"%@:
%@.\n",
     timeStamp, @"premature end of file."];
[newErrorLog writeToFile:errorPath atomically:YES];
```

Writing to Files

To write to a file, use the method **writeToFile:atomically:**. It uses the **description** method to obtain a human-readable string representation of the receiver and then writes the string to the specified file. The resulting file is suitable for use with *className***WithContentsOfFile:** methods. This method returns YES if the file is written successfully, and NO otherwise.

If the argument for **atomically:** is YES, the string representation is first written to an auxiliary file. Then the auxiliary file is renamed to the specified file name. If flag is NO, the object is written directly to the specified file. The YES option guarantees that the specified file, if it exists at all, won't be corrupted even if the system should crash during writing.

When **writeToFile:atomically:** fails, it returns NO. If this happens, check the permissions on the specified file and its directory. The most common cause of write failures is that the process owner doesn't have the necessary permissions to write to the file or its directory. If the argument for **atomically:** is NO, it's sufficient to grant write permissions only on the file.

Reading From Files

The string, array, and dictionary classes provide methods of the form *className***WithContentsOfFile:**. These methods create a new object and initialize it with the contents of a specified file, which can be specified with a full or relative pathname.

Working With Strings

NSString and NSMutableString objects represent static and dynamic character strings, respectively. They may be searched for substrings, compared with one another, combined into new strings, and so on.

The difference between NSStrings and NSMutableStrings is that you can't change an NSString's contents from its initial character string. While NSMutableString provides methods such as **appendString:** and **setString:** to add to or replace the string's contents, there are no such methods available for NSStrings. It's best to use NSStrings wherever possible. Only use an NSMutableString if you need to modify its contents after you create it.

You can create NSStrings with WebScript's "at sign" @ syntax for defining constant objects. For example, the following statement creates an NSString object:

```
id msg = @"This option is no longer available. Please
choose another.";
```

You can also create string objects with creation methods—methods whose names are preceded by a + and that return new objects. The strings created with the @ syntax are always NSStrings, so they can't be modified. If you use a creation method instead, you can choose to create either an NSString or a NSMutableString. The following code excerpt illustrates the creation of both NSString and NSMutableString objects:

```
// Create an immutable string
id message = [NSString stringWithString:@"Hi"];

// Create a mutable string
id message = [NSMutableString stringWithString:@"Hi"];
```

The methods provided by NSString and NSMutableString are described in more detail in the next section.

Commonly Used String Methods

The following sections list the most commonly used NSString and NSMutableString methods, grouped according to function.

Creating Strings

The methods for creating strings are class methods, denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSString and NSMutableString. For more information on class methods, see "Sending a Message to a Class" (page 211).

+ string

Returns an empty string. Usually used to create NSMutableStrings. NSStrings created with this method are permanently empty.

```
/* Most common use */
id mutableString = [NSMutableString string];

/* May not be what you want */
id string = [NSString string];
```

+ stringWithFormat:

Returns a string created by substituting arguments into a specified format string just as **printf()** does in the C programming language. In WebScript, only the "at sign" (@) conversion character is supported, and it expects a corresponding **id** argument.

```
  // These are fine
  id party = [NSString stringWithFormat:@"Party date:
%@", partyDate];
  id mailto = [NSString stringWithFormat:@"mailto:
%@",
      [person email]];
  id footer = [NSString stringWithFormat:
      @"Interaction %@ in session %@.",
      numberOfInteractions, sessionNumber];

  // NO! This won't work. Only %@ is supported.
  // (%d prints address, not value).
  id string = [NSString stringWithFormat:@"%d of %d
%s", x, y,
       cString];
```

+ stringWithString:

Returns a string containing the same contents as a specified string. This method is usually used to create an NSMutableString from an NSString. For example, the following statement creates an NSMutableString from a constant NSString object:

```
  id mutableString = [NSMutableString
stringWithString:@"Change me."];
```

+ stringWithContentsOfFile:

Returns a string created by reading characters from a specified file. For example, the following statement creates an NSString containing the contents of the file specified in **path**:

```
  id fileContents = [NSString
stringWithContentsOfFile:path];
```

See also **writeToFile:atomically:**.

Combining and Dividing Strings

– stringByAppendingFormat:

Returns a string made by appending to the receiver a string constructed from a specified format string and the arguments following it in the manner of **stringWithFormat:**. For example, assume the variable **guestName** contains the string "Rena". Then the following code excerpt produces the string **message** with contents "Hi, Rena!":

```
id string = @"Hi";
id message = [string stringByAppendingFormat:@",
%@!",
      guestName];
```

– stringByAppendingString:

Returns a string made by appending a specified string to the receiver. This code excerpt, for example, produces the string "Error: premature end of file.":

```
id errorTag = @"Error: ";
id errorString = @"premature end of file.";
id errorMessage = [errorTag
    stringByAppendingString:errorString];
```

– componentsSeparatedByString:

Returns an NSArray containing substrings from the receiver that have been divided by a specified separator string. For example, the following statements produce an NSArray containing the strings "wrenches", "hammers", and "saws":

```
id toolString = @"wrenches, hammers, saws";
id toolArray = [toolString
componentsSeparatedByString:@", "];
```

See also **componentsJoinedByString:** (NSArray and NSMutableArray).

– substringToIndex:

Returns a string object containing the characters of the receiver up to, but not including, the one at the specified index.

– substringFromIndex:

Returns a string containing the characters of the receiver from the character at the specified index to the end.

Comparing Strings

– compare:

Returns –1 if the receiver precedes a specified string in lexical ordering, 0 if it is equal, and 1 if it follows. For example, the following statements result in an NSString **result** with the contents "'hello' precedes 'Hello' lexicographically.":

```
if ([@"hello" compare:@"Hello"] == -1) {
    result = [NSString stringWithFormat:
        @"'%@' precedes '%@' lexicographically.",
        @"hello", @"Hello"];
}
```

– caseInsensitiveCompare:

Same as **compare:**, but case distinctions among characters are ignored.

– isEqual:

Returns YES if a specified object is equivalent to the receiver; NO otherwise. An object is equivalent to a string if the object is an NSString or an NSMutableString and **compare:** returns 0. For example, the following statements:

```
if ([string isEqual:newString]) {
    result = @"Found a match";
}
```

assign the contents "Found a match" to **result** if **string** and **newString** are lexicographically equal.

Converting String Contents

– doubleValue

Returns the floating-point value of the receiver's text as a double, skipping white space at the beginning of the string. Returns 0 if the string is not a number.

– floatValue

Returns the floating-point value of the receiver's text as a float, skipping white space at the beginning of the string. Returns 0 if the string is not a number.

– intValue

Returns the integer value of the string's text, assuming a decimal representation and skipping white space at the beginning of the string. Returns 0 if the string is not a number.

Modifying Strings

Warning: The following methods are not supported by NSString. They are available only to NSMutableString objects.

– appendFormat:

Appends a constructed string to the receiver. Creates the new string by using **stringWithFormat:** method with the arguments listed. For example, in the following code excerpt, if you assume the variable **guestName** contains the string "Rena", then **message** has the resulting contents "Hi, Rena!":

```
  id message = [NSMutableString
stringWithString:@"Hi"];
  [message appendFormat:@", %@!", guestName];
```

– appendString:

Adds the characters of a specified string to the end of the receiver. For example, the following statements create an NSMutableString and append another string to its initial value:

```
  id mutableString = [NSMutableString
stringWithFormat:@"Hello "];
  [mutableString appendString:@"world!"];
```

mutableString has the resulting contents "Hello world!".

– setString:

Replaces the characters of the receiver with those in a specified string. For example, the following statement replaces the contents of an NSMutableString with the empty string:

```
  [mutableString setString:@""];
```

Storing Strings

– writeToFile:atomically:

Writes the string to a specified file, returning YES on success and NO on failure. If YES is specified for **atomically:**, this method writes the string to an auxiliary file and then renames the auxiliary file to the specified path. In this way, it ensures that the contents of the specified path do not become corrupted if the system crashes during writing. The resulting file is suitable for use with **stringWithContentsOfFile:**. For example, the following code excerpt reads the contents of an error log stored in a file, appends a new error to the log, and saves the updated log to the same file:

```
    id errorLog = [NSString
stringWithContentsOfFile:errorPath];
    id newErrorLog = [errorLog
stringByAppendingFormat:@"%@: %@.\n",
        timeStamp, @"premature end of file."];
    [newErrorLog writeToFile:errorPath atomically:YES];
```

Working With Arrays

NSArray and NSMutableArray objects manage immutable and mutable collections of objects, respectively. Each has the following attributes:

- A count of the number of objects in the array
- The objects contained in the array

The difference between NSArray and NSMutableArray is that you can't add to or remove from an NSArray's initial collection of objects. That is, insertion and deletion methods provided for NSMutableArrays are not available for NSArrays. Although their use is limited to managing static collections of objects, it is best to use NSArrays wherever possible.

You can create NSArrays with WebScript's @ syntax for defining constant objects. For example, the following statements create NSArrays:

```
id availableQuantities = @(1, 6, 12, 48);
id shortWeekDays = @("Sun", "Mon", "Tue", "Wed", "Thu",
"Fri", "Sat");
```

You can also create NSArrays with creation methods. If you want to create a static array that contains variables, you have to use a creation method because you can't use variables in WebScript's @ syntax. The following statement creates an NSArray that contains variables:

```
id dinnerPreferences = [NSArray
arrayWithObjects:firstChoice,
      secondChoice, nil];
```

The variable **dinnerPreferences** is an NSArray, so its initial collection of objects can't be added to or subtracted from. When you need to create an array that can be modified, use a creation method to create an NSMutableArray. For example, the following statement creates an empty NSMutableArray to which you can add objects:

```
id mutableArray = [NSMutableArray array];
```

The methods provided by NSArray and NSMutableArray are described in more detail in the next section.

Commonly Used Array Methods

The following sections list the most commonly used NSArray and NSMutableArray methods. The methods covered are grouped according to function.

Creating Arrays

The methods in this section are class methods, as denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSArray and NSMutableArray. For more information on class methods, see "Sending a Message to a Class" (page 211).

+ array

Returns an empty array. Usually used to create NSMutableArrays. NSArrays created with this method are permanently empty.

```
// Most common use
id mutableArray = [NSMutableArray array];

// May not be what you want
id array = [NSArray array];
```

+ arrayWithObject:

Returns an array containing the single specified object.

+ arrayWithObjects:

Returns an array containing the objects in the argument list. The argument list is a comma-separated list of objects ending with **nil**.

```
id array = [NSMutableArray arrayWithObjects:
    @"Plates", @"Plasticware", @"Napkins", nil];
```

+ arrayWithArray:

Returns an array containing the contents of a specified array. Usually used to create an NSMutableArray from an immutable NSArray, or vice-versa. For example, the following statement creates an NSMutableArray from a constant NSArray object:

```
id mutableArray = [NSMutableArray
    arrayWithArray:@("A", "B", "C")];
```

+ arrayWithContentsOfFile:

Returns an array initialized from the contents of a specified file. The specified file can be a full or relative pathname; the file that it names must contain a string representation of an array, such as that produced by the **writeToFile:atomically:** method. See also **description**.

Querying Arrays

– count

Returns the number of objects in the array.

– isEqual:

Returns YES if the specified object is an array and has contents equivalent to the receiver; NO, otherwise. Two arrays have equal contents if they each hold the same number of objects and objects at a given index in each array satisfy the **isEqual:** test.

– objectAtIndex:

Returns the object located at a specified index. Arrays have a zero-based index. The first object in an array is at index 0, the second is at index 1, and so on. It is an error to specify an index that is out of bounds (greater than or equal to the array's count).

– indexOfObject:

Returns the index of the first object in the array that is equivalent to a specified object, or NSNotFound if an equivalent object isn't found. To determine equality, each element of the array is sent an **isEqual:** message.

– indexOfObjectIdenticalTo:

Returns the index of the first occurrence of the a specified object, or NSNotFound if the specified object isn't found. To determine equality, the **id**s of the two objects are compared.

Sorting Arrays

– sortedArrayUsingSelector:

Returns an NSArray that lists the receiver's elements in ascending order, as determined by a specified method. This method is used to sort arrays containing strings and/or numbers. For example, the following code excerpt creates the NSArray **sortedArray** containing the string "Alice" at index 0, "David" at index 1, and so on:

```
     id guestArray = @("Suzy", "Alice", "John", "Peggy",
"David");
     id sortedArray = [guestArray
sortedArrayUsingSelector:@"compare:"];
```

Adding and Removing Objects

Warning: The following methods are not supported by NSArray. They are available only to NSMutableArray objects.

When removing objects using NSMutableArray's **removeObject...** methods, note that the objects are sent a **release** message as they're removed. Thus, the following can cause an error:

```
id anObj = [aList objectAtIndex:0];
[aList removeObjectAtIndex:0];
[aList addObject:anObj];// anObj may already be dealloc'd
```

– addObject:

Adds a specified object at the end of the receiver. It is an error to specify **nil** as an argument to this method. You cannot add **nil** to an array.

– insertObject:atIndex:

Inserts an object at a specified index. If the specified index is already occupied, the objects at that index and beyond are shifted down one slot to make room. The specified index can't be greater than the receiver's count, and the specified object cannot be **nil**.

Array objects have a zero-based index. The first object in an array is at index 0, the second is at index 1, and so on. You can insert only new objects in ascending order—with no gaps. Once you add two objects, the array's size is 2, so you can insert objects at indexes 0, 1, or 2. Index 3 is illegal and out of bounds.

It is an error to specify **nil** as an argument to this method. You cannot add **nil** to an array. It is also an error to specify an index that is greater than the array's count.

– removeObject:

Removes all objects in the array equivalent to a specified object, and moves elements up as necessary to fill any gaps. Equivalency is determined using the **isEqual:** method. Removed objects are sent **release** as they're removed.

– removeObjectIdenticalTo:

Removes all occurrences of a specified object and moves elements up as necessary to fill any gaps. Removed objects are sent **release** as they're removed.

– removeObjectAtIndex:

Removes the object at a specified index and moves all elements beyond the index up one slot to fill the gap. Arrays have a zero-based index. The first object in an array is at index 0, the second is at index 1, and so on. Removed objects are sent **release** as they're removed.

It is an error to specify an index that is out of bounds (greater than or equal to the array's count).

– removeAllObjects

Empties the receiver of all of its elements. Removed objects are sent **release** as they're removed.

– setArray:

Empties the receiver of all its elements, then adds the contents of a specified array.

Storing Arrays

– writeToFile:atomically:

Writes the array's string representation to a specified file using the **description** method. Returns YES on success and NO on failure. If YES is specified for **atomically:**, this method attempts to write the file safely so that an existing file with the specified path is not overwritten, and it does not create a new file at the specified path unless the write is successful. The resulting file is suitable for use with **arrayWithContentsOfFile:**. For example, the following code excerpt creates **guestArray** with the contents of the specified file, adds a new guest, and saves the changes to the same file:

```
      id guestArray = [NSMutableArray
arrayWithContentsOfFile:path];
      [guestArray addObject:newGuest];
      [guestArray writeToFile:path atomically:YES];
```

Representing Arrays as Strings

– description

Returns a string that represents the contents of the receiver. For example, the following code excerpt produces the string "(Plates, Plasticware, Napkins)":

```
id array = [NSMutableArray arrayWithObjects:
    @"Plates", @"Plasticware", @"Napkins", nil];
id description = [array description];
```

– componentsJoinedByString:

Returns an NSString created by interposing a specified string between the elements of the receiver's objects. Each element of the array must be a string. If the receiver has no elements, an empty string is returned. See also **componentsSeparatedByString:** (in NSString and NSMutableString). For example, the following code excerpt creates the NSString **dashString** with the contents "A-B-C":

```
id commaString = @"A, B, C";
id array = [string
componentsSeparatedByString:@","];
id dashString = [array componentsJoinedByString:@"-
"];
```

Working With Dictionaries

NSDictionary and NSMutableDictionary objects store collections of key-value pairs. The key-value pairs within a dictionary are called *entries*. Each entry consists of an object that represents the key, and a second object that represents the key's value. Within a dictionary, the keys are unique. That is, no two keys in a single dictionary are equivalent.

The difference between NSDictionary and NSMutableDictionary is that you can't add, modify, or remove entries from an NSDictionary's initial collection of entries. Insertion and deletion methods provided for NSMutableDictionaries are not available for NSDictionaries. Although their use is limited to managing static collections of objects, it's best to use NSDictionaries wherever possible.

You can create NSDictionaries using WebScript's @ syntax for defining constant objects. For example, the following statements create NSDictionaries:

```
id sizes = @{"S" = "Small"; "M" = "Medium"; "L" = "Large";
"X" = "Extra Large"};
id defaultPreferences = @{
  "seatAssignment" = "Window";
  "smoking" = "Non-smoking";
  "aircraft" = "747"};
```

You can also create dictionaries using creation methods. For example, if you want to create an NSDictionary that contains variables, you have to use a creation method. You can't use variables with WebScript's @ syntax. The following statement creates an NSDictionary that contains variables:

```
id customerPreferences = [NSDictionary
dictionaryWithObjectsAndKeys:
  seatingPreference, @"seatAssignment",
  smokingPreference, @"smoking",
  aircraftPreference, @"aircraft", nil];
```

The variable **customerPreferences** is an NSDictionary, so its initial collection of entries can't be modified. To create a dictionary that can be modified, use a creation method to create an NSMutableDictionary. For example, the following statement creates an empty NSMutableDictionary:

```
id dictionary = [NSMutableDictionary dictionary];
```

The methods provided by NSDictionary and NSMutableDictionary are described in more detail next.

Commonly Used Dictionary Methods

The following sections list some of the most commonly used methods of NSDictionary and NSMutableDictionary, grouped according to function.

Creating Dictionaries

The methods in this section are class methods, as denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSDictionary and NSMutableDictionary. For more information on class methods, see "Sending a Message to a Class" (page 211).

+ dictionary

Returns an empty dictionary. Usually used to create NSMutableDictionaries. NSDictionaries created with this method are permanently empty.

```
// Most common use
id mutableDictionary = [NSMutableDictionary
dictionary];

// May not be what you want
id dictionary = [NSDictionary dictionary];
```

+ dictionaryWithObjects:forKeys:

Returns a dictionary containing entries constructed from the contents of a specified array of objects and a specified array of keys. The two arrays must have the same number of elements.

```
id preferences = [NSMutableDictionary
    dictionaryWithObjects:@("window", "non-
smoking", "747")
    forKeys:@("seatAssignment", "smoking",
"aircraft")];
```

+ dictionaryWithObjectsAndKeys:

Returns a dictionary containing entries constructed from a specified set of objects and keys. This method takes a variable

number of arguments: a list of alternating objects and keys ending with **nil**.

```
    id customerPreferences = [NSDictionary
dictionaryWithObjectsAndKeys:
        seatingPreference, @"seatAssignment",
        smokingPreference, @"smoking",
        aircraftPreference, @"aircraft", nil];
```

+ dictionaryWithDictionary:

Returns a dictionary containing the contents of a specified dictionary. Usually used to create an NSMutableDictionary from an immutable NSDictionary.

+ dictionaryWithContentsOfFile:

Returns a dictionary initialized from the contents of a specified file. The specified file can be a full or relative pathname; the file that it names must contain a string representation of a dictionary, such as that produced by the **writeToFile:atomically:** method.

See also **description**.

Querying Dictionaries

– allKeys

Returns an array containing the dictionary's keys or an empty array if the dictionary has no entries. This method is useful for accessing all the entries in a dictionary. For example, the following code excerpt creates the NSArray **keys** and uses it to access the value of each entry in the dictionary:

```
id index;
id keys = [dictionary allKeys];
for (index = 0; index < [keys count]; index++) {
    value = [dictionary objectForKey:[keys
    objectAtIndex:index]];
    // Use the value
}
```

– allKeysForObject:

Returns an array containing all the keys corresponding to values equivalent to a specified object. Equivalency is determined using the **isEqual:** method. If the specified object isn't equivalent to any of the values in the receiver, this method returns **nil**.

– allValues:

Returns an array containing the dictionary's values, or an empty array if the dictionary has no entries.

Note that the array returned from **allValues** may have a different count than the array returned from **allKeys**. An object can be in a dictionary more than once if it corresponds to multiple keys.

– keysSortedByValueUsingSelector:

Returns an NSArray containing the dictionary's keys such that their corresponding values are sorted in ascending order, as determined by a specified method. For example, the following code excerpt creates the NSArray **keys** containing the string "Pasta" at index 0, "Seafood" at index 1, and "Steak" at index *2*:

```
    id choices = @{"Steak" = 3; "Seafood" = 2; "Pasta" =
1};
    id keys = [choices
keysSortedByValueUsingSelector:@"compare:"];
```

– count

Returns the number of entries currently in the dictionary.

– isEqual:

Returns YES if the specified object is a dictionary and has contents equivalent to the receiver; NO, otherwise. Two dictionaries have equivalent contents if they each hold the same number of entries and, for a given key, the corresponding value objects in each dictionary satisfy the **isEqual:** test.

– objectForKey:

Returns the object that corresponds to a specified key. For example, the following code excerpt produces the NSString **sectionPreference** with the contents "non-smoking":

```
    id preferences = [NSMutableDictionary
        dictionaryWithObjects:@("window", "non-
smoking", "747")
        forKeys:@("seatAssignment", "section",
"aircraft")];
    id sectionPreference = [dictionary
objectForKey:@"section"];
```

Adding, Removing, and Modifying Entries

Warning: The following methods are not supported by NSDictionary. They are available only to NSMutableDictionary objects.

– setObject:forKey:

Adds an entry to the receiver, consisting of a specified key and its corresponding value object. If the receiver already has an entry for the specified key, the previous value for that key is replaced with the argument for **setObject:**. For example, the following code excerpt produces the NSMutableDictionary **dictionary** with the value "non-smoking" for the key "section" and the value "aisle" for the key "seatAssignment." Notice that the original value for "seatAssignment" is replaced:

```
      id dictionary = [NSMutableDictionary
dictionaryWithDictionary:
          @{"seatAssignment" = "window"}];
      [dictionary setObject:@"non-smoking"
forKey:@"section"];
      [dictionary setObject:@"aisle"
forKey:@"seatAssignment"];
```

It is an error to specify **nil** as an argument for **setObject:** or **forKey:**. You can't put **nil** in a dictionary as a key or as a value.

– addEntriesFromDictionary:

Adds the entries from a specified dictionary to the receiver. If both dictionaries contain the same key, the receiver's previous value for that key is replaced with the new value.

– removeAllObjects

Empties the dictionary of its entries.

– removeObjectForKey:

Removes the entry for a specified key. Releases the object and the key.

– removeObjectsForKeys:

Removes the entries for each key in a specified array. Releases the object and the key.

– setDictionary:

Removes all the entries in the receiver, then adds the entries from a specified dictionary.

Representing Dictionaries as Strings

– description

Returns a string that represents the contents of the receiver. For example, the following code excerpt produces the string "{"seatAssignment" = "Window"; "section" = "Non-smoking"; "aircraft" = "747"}":

```
    id preferences = [NSMutableDictionary
        dictionaryWithObjects:@("window", "non-
smoking", "747")
        forKeys:@("seatAssignment", "section",
"aircraft")];
    id description = [preferences description];
```

Storing Dictionaries

– writeToFile:atomically:

Writes the dictionary's string representation to a specified file using the **description** method. Returns YES on success and NO on failure. If YES is specified for **atomically:**, this method attempts to write the file safely so that an existing file with the specified path is not overwritten. It does not create a new file at the specified path unless the write is successful. The resulting file is suitable for use with **dictionaryWithContentsOfFile:**. For example, the following excerpt creates an NSMutableDictionary from the contents of the file specified by **path**, updates the object for the key @"Language", and saves the updated dictionary back to the same file:

```
    id defaults = [NSMutableDictionary
        dictionaryWithContentsOfFile:path];
    [defaults setObject:newLanguagePreference
forKey:@"Language"];
    [defaults writeToFile:path atomically:YES];
```

See also **description**.

Working With Dates and Times

NSCalendarDate objects represent dates and times. These objects are especially suited for representing and manipulating dates according to Western calendrical systems, primarily the Gregorian.

The methods provided by NSCalendarDate are described in more detail in "Commonly Used Date Methods" on page 257.

The Calendar Format

Each NSCalendarDate object has a calendar format associated with it. This format is a string that contains date-conversion specifiers very similar to those used in the standard C library function **strftime()**. NSCalendarDate interprets dates that are represented as strings conforming to this format. You can set the default format for an NSCalendarDate object either at initialization time or by using the **setCalendarFormat:** method. Several methods allow you to specify formats other than the one bound to the object.

Date Conversion Specifiers

The date conversion specifiers cover a range of date conventions:

Conversion Specifier	Argument Type
%%	a '%' character
%A, %a	full and abbreviated weekday name, respectively
%B, %b	full and abbreviated month name, respectively
%c	date and time designation for the locale
%d	day of the month as a decimal number (01–31)
%F	milliseconds as a decimal number (000-999)
%H, %I	hour based on a 24-hour or 12-hour clock as a decimal number, respectively. (00–23 or 01–12)
%j	day of the year as a decimal number (001–366)
%M	minute as a decimal number (00–59)
%m	month as a decimal number (01–12)

Conversion Specifier	Argument Type
%p	AM/PM designation for the locale
%S	second as a decimal number (00–59)
%w	weekday as a decimal number (0–6), where Sunday is 0
%x	date using date representation for the locale
%X	time using time representation for the locale
%Y, %y	year with century (such as 1990) and year without century (00-99), respectively
%Z, %z	time zone abbreviation (such as PDT) and time zone offset in hours and minutes from GMT (HHMM), respectively

Commonly Used Date Methods

The following sections list some of the most commonly used methods of NSCalendarDate, grouped according to function.

Creating Dates

The methods in this section are class methods, denoted by the plus sign (+). You use class methods to send messages to a class—in this case, NSCalendarDate. For more information on class methods, see "Sending a Message to a Class" (page 211).

+ calendarDate

Returns an NSCalendarDate initialized to the current date and time.

+ dateWithString:calendarFormat:

Returns an NSCalendarDate initialized to the date in a provided string, and sets the new NSCalendarDate's calendar format to the specified format. The date string must match the provided format exactly. See "Date Conversion Specifiers" for more detailed information on formats used by NSCalendarDate.

Adjusting a Date

– **dateByAddingYears:months:days:hours:minutes:seconds:**

Returns an NSCalendarDate derived from the receiver by adding a specified number of years, months, days, hours, minutes, and seconds.

Representing Dates as Strings

– **description**

Returns a string representation of the NSCalendarDate formatted according to the NSCalendarDate's default calendar format.

– **descriptionWithCalendarFormat:**

Returns a string representation of the receiver formatted according to the provided format string.

– **calendarFormat**

Returns a string that indicates the receiver's default calendar format. See "Date Conversion Specifiers" for more detailed information on formats used by NSCalendarDate.

– **setCalendarFormat:**

Set the receiver's default calendar format to the provided string.

Retrieving Date Elements

– **dayOfWeek**

Returns a number that indicates the NSCalendarDate's day of the week (0–6).

– **dayOfMonth**

Returns the NSCalendarDate's day of the month (1–31).

– **dayOfYear**

Returns a number that indicates the NSCalendarDate's day of the year (1–366).

– **dayOfCommonEra**

Returns the NSCalendarDate's number of days since the beginning of the Common Era. The base year of the Common Era is 1 A.C.E. (which is the same as 1 A.D.).

– **monthOfYear**

Returns a number that indicates the NSCalendarDate's month of the year (1–12).

– **yearOfCommonEra**

Returns the NSCalendarDate's year value (including the century).

– **hourOfDay**

Returns the NSCalendarDate's hour value (0–23).

– **minuteOfHour**

Returns the NSCalendarDate's minutes value (0–59).

– **secondOfMinute**

Returns the NSCalendarDate's seconds value (0–59).

Index

O

P, Q

R

S

Y, Z