

WEBOBJECTS DEVELOPER'S GUIDE

Apple, NeXT, and the publishers have tried to make the information contained in this manual as accurate and reliable as possible, but assume no responsibility for errors or omissions. They disclaim any warranty of any kind, whether express or implied, as to any matter whatsoever relating to this manual, including without limitation the merchantability or fitness for any particular purpose. In no event shall they be liable for any indirect, special, incidental, or consequential damages arising out of purchase or use of this manual or the information contained herein. NeXT or Apple will from time to time revise the software described in this manual and reserves the right to make such changes without obligation to notify the purchaser.

Copyright © 1997 by Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014.
All rights reserved.
[7010.01]

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher or copyright owner. Printed in the United States of America. Published simultaneously in Canada.

NeXT, the NeXT logo, OPENSTEP, Enterprise Objects, EOF, Enterprise Objects Framework, ProjectBuilder, Objective-C, Portable Distributed Objects, Workspace Manager, Database Wizard, WEBSOCKET, and WEBOBJECTS are trademarks of NeXT Software, Inc. PostScript is a registered trademark of Adobe Systems, Incorporated. Windows NT is a trademark of Microsoft Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. ORACLE is a registered trademark of Oracle Corporation, Inc. SYBASE is a registered trademark of Sybase, Inc. All other trademarks mentioned belong to their respective owners.

Restricted Rights Legend: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 [or, if applicable, similar clauses at FAR 52.227-19 or NASA FAR Supp. 52.227-86].

This manual describes WebObjects, version 3.5.

Written by Terry Donoghue, Katie McCormick, Matt Morse, Jean Ostrem, Kelly Toshach
With help from Eric Bailey, Craig Federighi, Patrice Gautier, Francois Jouaux, Charles Lloyd, and Dan Peknik
Developmental Editing by Jeanne Woodward
Proofread by Laurel Rezeau
Book Design by Karin Stroud
Technical illustrations by Karin Stroud
Print and Online Production Editing by Gerri Gray
Art, Production, and Editorial management by Gary Miller
Technical publications management by Greg Wilson

Contents

Table of Contents

Introduction 9

About This Book 11
Other Useful Documentation 12

Part I WebObjects Essentials

What Is a WebObjects Application? 17

The Ingredients of a WebObjects Application 19
 Components 20
 Template 21
 Code or Script File 21
 Bindings 22
 Application Code 23
 Session Code 23
 A Note on WebObjects Classes 23
 Application Directory 24
Running a WebObjects Application 25
 WebObjects Adaptors 26
 The WebObjects Application Executable 27

Dynamic Elements 29

Server-Side Dynamic Elements 31
 How Server-Side Dynamic Elements Work 33
 Binding Values to Dynamic Elements 35
 Declarations File Syntax 36
Client-Side Java Components 37
 Deciding When to Use Client-Side Components 37
 How Client-Side Components Work 38

Common Methods 41

Action Methods 43
Initialization and Deallocation Methods 46
 The Structures of init and awake 46
 Application Initialization 47
 Session Initialization 48
 Component Initialization 49
Request-Handling Methods 50
 Taking Input Values From a Request 51
 Invoking an Action 52
 Limitations on Direct Requests 53
 Generating a Response 53

Debugging a WebObjects Application 55

Launching an Application for Debugging 57
 Debugging WebScript 57
 Debugging Java 57
 Debugging Objective-C 58
 Debugging Mixed Applications 58
Debugging Techniques 58
 Writing Debug Messages 58
 Using Trace Methods 59
 Isolating Portions of a Page 60
Programming Pitfalls to Avoid 60
 WebScript Programming Pitfalls 60
 Java Programming Pitfalls 61

WebObjects Viewed Through Its Classes 63

The Classes in the Request-Response Loop 65
 Server and Application Level 65
 Session Level 66
 Request Level 68
 Page Level 69
 Database Integration Level 71

How WebObjects Works—A Class Perspective 72

- Starting the Request-Response Loop 72
- Taking Values From the Request 73
 - Accessing the Session 74
 - Creating or Restoring the Request Page 76
 - Assigning Input Values 78
- Invoking an Action 79
- Generating the Response 80

How HTML Pages Are Generated 82

- Component Templates 82
- Associations and the Current Component 84
- Associations and Client-Side Java Components 85
- Subcomponents and Component References 86

Part II Special Tasks

Creating Reusable Components 91

- Benefits of Reusable Components 93
 - Centralizing Application Resources 93
 - Simplifying Interfaces 96
- Intercomponent Communication 98
 - Synchronizing Attributes in Parent and Child Components 102
- Sharing Reusable Components Across Applications 104
- Search Path for Reusable Components 105
- Designing for Reusability 106

Managing State 109

- Why Do You Need to Store State? 111
- When Do You Need to Store State? 112
- Objects and State 113
 - The Application Object and Application State 113
 - The Session Object and Session State 115
 - Component Objects and Component State 118
- State Storage Strategies 120
 - Comparison of Storage Options 121
 - A Closer Look at Storage Strategies 122
 - State in the Server 123

- State in the Page 124
- State in Cookies 126
- Custom State-Storage Options 128

Storing State for Custom Objects 131

- Archiving Custom Objects in a Database Application 131
- Archiving Custom Objects in Other Applications 132

Controlling Session State 133

- Setting Session Time-Out 134
- Using awake and sleep 135

Controlling Component State 135

- Managing Component Resources 135
 - Adjusting the Page Cache Size 136
 - Using awake and sleep 137
 - pageWithName: and Page Caching 138
- Client-Side Page Caching 139
 - Page Refresh and WODisplayGroup 140

Creating Client-Side Components 141

- Choosing a Strategy 143
- When You Have an Applet's Source Code 144
- When You Don't Have an Applet's Source Code 146

Deployment and Performance Issues 149

- Recording Application Statistics 151
 - Maintaining a Log File 151
 - Accessing Statistics 152
 - Recording Extra Information 153
- Error Handling 154
- Automatically Terminating an Application 155
- Performance Tips 156
 - Cache Component Definitions 156
 - Compile the Application 157
 - Control Memory Leaks 157
 - Limit State Storage 158
 - Limit Database Fetches 158
 - Limit Page Sizes 158
- Installing Applications 159

Part III WebScript

The WebScript Language 163

Objects in WebScript 165

WebScript Language Elements 166

Variables 166

Variables and Scope 167

Assigning Values to Variables 168

Methods 170

Invoking Methods 171

Accessor Methods 171

Sending a Message to a Class 172

Creating Instances of Classes 173

Data Types 174

Statements and Operators 175

Control-Flow Statements 176

Arithmetic Operators 176

Logical Operators 176

Relational Operators 176

Increment and Decrement Operators 177

Reserved Words 178

“Modern” WebScript Syntax 179

Advanced WebScript 181

Scripted Classes 181

Categories 182

WebScript for Objective-C Developers 183

Accessing WebScript Methods From Objective-C Code 185

WebScript Programmer’s Quick Reference to Foundation Classes 187

Foundation Objects 189

Representing Objects as Strings 189

Mutable and Immutable Objects 189

Determining Equality 190

Writing to and Reading From Files 190

Writing to Files 190

Reading From Files 191

Working With Strings 191

Commonly Used String Methods 192

Creating Strings 192

Combining and Dividing Strings 193

Comparing Strings 194

Converting String Contents 194

Modifying Strings 195

Storing Strings 195

Working With Arrays 196

Commonly Used Array Methods 196

Creating Arrays 197

Querying Arrays 197

Sorting Arrays 198

Adding and Removing Objects 198

Storing Arrays 200

Representing Arrays as Strings 200

Working With Dictionaries 200

Commonly Used Dictionary Methods 201

Creating Dictionaries 202

Querying Dictionaries 203

Adding, Removing, and Modifying Entries 204

Representing Dictionaries as Strings 205

Storing Dictionaries 205

Working With Dates and Times 206

The Calendar Format 206

Date Conversion Specifiers 206

Commonly Used Date Methods 207

Creating Dates 207

Adjusting a Date 207

Representing Dates as Strings 208

Retrieving Date Elements 208

Index 211

Introduction

WebObjects is an object-oriented environment for developing and deploying World Wide Web applications. A WebObjects application runs on a server machine, receives requests from a user's web browser on a client machine, dynamically generates the appropriate HTML page in response to those requests, and returns that page to the user. WebObjects provides you with a web application server, prebuilt application components, and a suite of tools for rapid development of World Wide Web applications.

WebObjects is flexible enough to suit the needs of any web programmer. You can, for instance, write code using one of three programming languages: Java, Objective-C, or WebScript. You can write simple WebObjects applications in a matter of minutes. And if your programming task is more complex, WebObjects still makes it as easy as possible by performing common web application tasks automatically and by allowing you to reuse objects you've written for other applications.

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.

Other Useful Documentation

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.

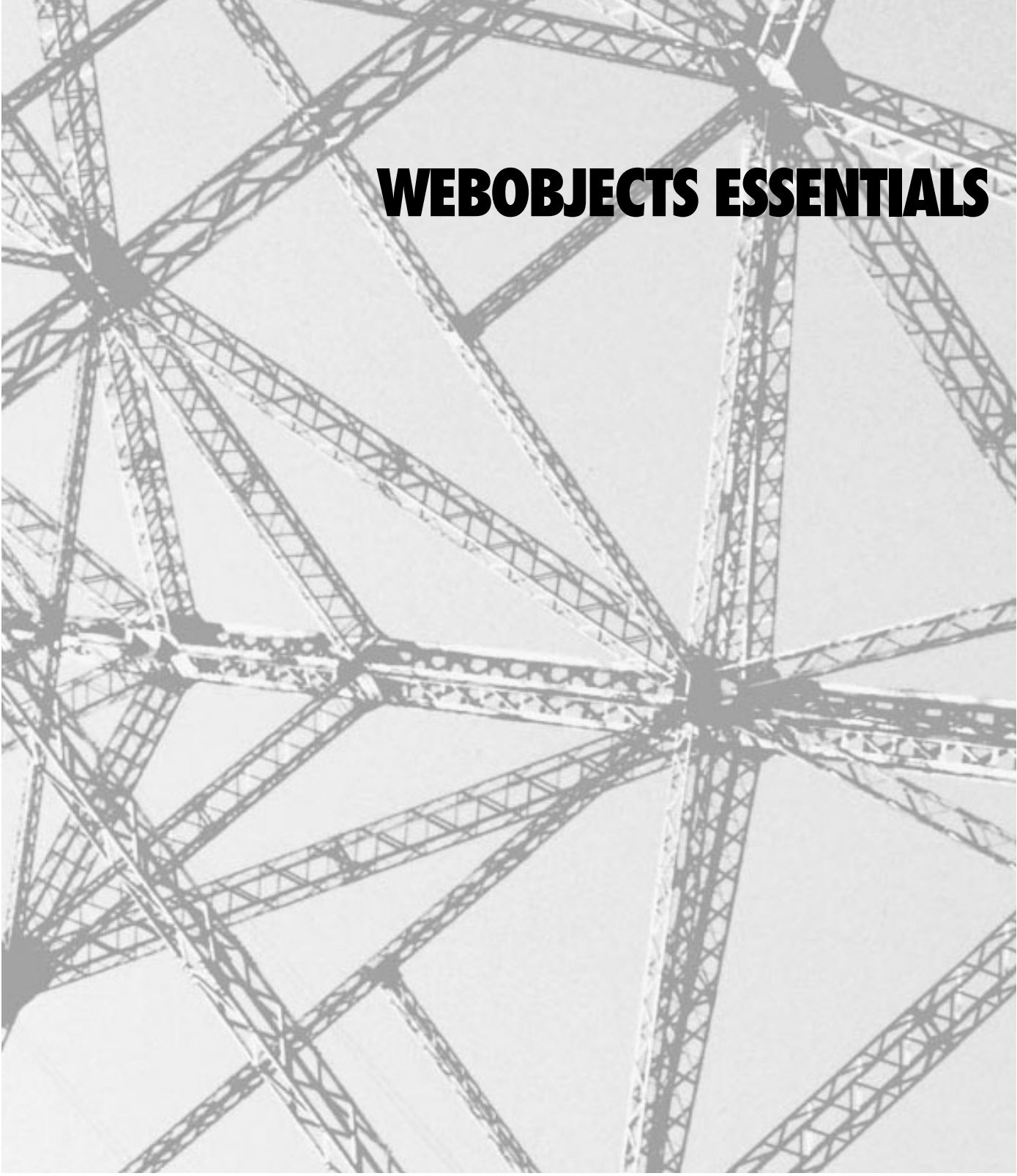
If you want to write an application that accesses a database, you’ll need to use Enterprise Objects in conjunction with WebObjects. And although database access is covered in the tutorials in *Getting Started With WebObjects*, you’ll probably also want to read the *Enterprise Objects Framework Developer’s Guide* for more in-depth information.

Other valuable information can be found online. To access online documentation, use the WebObjects Home Page. The WebObjects Home Page is in your web server’s document root, and you can access it at this URL:

`http://localhost/WebObjects/Documentation/WOHomePage.html`

In particular, WOHomePage’s Documentation link gives you access to some books that are available only online:

- *WebObjects Tools and Techniques* describes the development tools WebObjects Builder and Project Builder and shows how to use them to create WebObjects applications.
- *Serving WebObjects* describes how to administer and deploy WebObjects applications after you've written them.
- The *WebObjects Class 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.
- The *Client-Side Applet Controls Reference* lists and describes the client-side Java components provided with WebObjects.



WEBOBJECTS ESSENTIALS

What Is a WebObjects Application?

WebObjects is a product that makes it easy for you to write dynamic web-based applications (or *WebObjects applications*). Before you start programming, however, you need to understand what a WebObjects application is.

This chapter answers the question what is a WebObjects application in two ways: first by showing you the pieces of a simple WebObjects application, and then by explaining what happens when a WebObjects application runs. In the rest of Part 1, you'll learn how to construct these pieces and you'll get more in-depth information about how they work.

Read this chapter if you want a very high-level overview. The rest of this book provides much more detailed information about how WebObjects applications work and how to write one.

When you're ready to start programming, read the book *Getting Started With WebObjects*. It provides a series of tutorials that help you understand the tasks and tools involved in writing a WebObjects application.

The Ingredients of a WebObjects Application

WebObjects applications reside within a directory named **WebObjects** in your web server's document root (`<DocumentRoot>/WebObjects`). Look in `<DocumentRoot>/WebObjects/Examples/WebScript`, and you'll see several directories. These are WebObjects application projects, provided with the WebObjects package as examples that you can use when learning WebObjects. These examples range from simple to highly complex. For now, you might want to focus on two of the simplest applications, named **HelloWorld** and **Visitors**.

When you look inside any of these project directories, you may see these pieces:

- **.wo** directories, which are called *components*. Components are dynamic HTML pages.
- An application code file (**Application.wos**), which creates and manages applicationwide resources.
- A session code file (**Session.wos**), which creates and manages sessionwide resources.
- Standard project files, such as makefiles.

The following sections describe the WebObjects application ingredients in more detail.

Components

To write a WebObjects application, you create components and then connect them. 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.” However, remember, that not all components represent an entire page. For example, a component might represent only a header or footer of a page, and you can nest that component inside of a component that does represent the entire page.

Each component is located in its own directory, named *Component.wo*, and generally contains these parts:

- 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

Figure 1 shows the contents of the **Main.wo** component from the HelloWorld example. (**Main.wo** is almost always the name of the first page of a WebObjects application.) In this example, the **Main.wo** component contains three files: a template in the form of an HTML file (**Main.html**), the code file (**Main.wos**), and the declarations file (**Main.wod**), which contains the bindings between the template and the code.

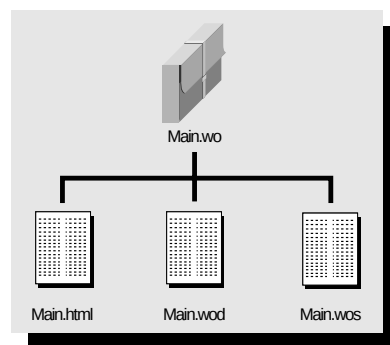


Figure 1. The Contents of a WebScript Component Directory

Typically, components contain some form of the three files shown in Figure 1; however, any given component might contain more or fewer files. For example, components whose code is written in a compiled language do not contain code

files. A component may not need a code file at all; it may need only a template file and a declarations file. Another component might have a code file but no template file or declarations file. 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.

The next three sections describe more completely these template, code, and declarations files.

Template

You use a *template* (*Main.html*) 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 runtime.

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* (*Main.wos*) to define your component's attributes and actions. The attributes are called *variables* or *instance variables*, and the actions are called *methods*.

With WebObjects, you can write your code file in one of three programming languages: Java, Objective-C, or WebScript. Java is the language of choice for many people; others prefer Objective-C. Because both of these languages require compiling, they aren't as well suited to rapid prototyping as a scripting language is. For this reason WebObjects provides a scripting language named WebScript, described in the chapter "The WebScript Language" (page 163). You may have noticed that the examples directory mentioned previously offers examples in all three languages.

Note: Java support is not available on the Mach or HP-UX platform.

The *Main.wo* component shown in Figure 1 uses a WebScript file to define its behavior. (The *.wos* extension signifies WebScript.) If you want to use Java or Objective-C, the code file resides at the same level as the *Main.wo* directory

as shown in Figure 2. (In Project Builder, Java and Objective-C code files are shown under Classes instead of with the component under Web Components.)

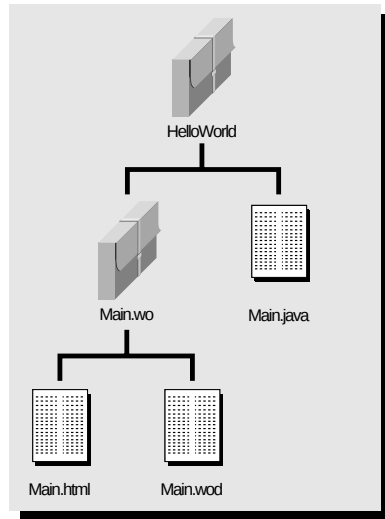


Figure 2. Location of Code File for Java Component

You can mix languages. It's common to use WebScript to write your interface logic (that is, the files described in this chapter) 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.

Bindings

You use a *declarations file* (**Main.wod**) to define the bindings, or mapping, between the methods and variables you defined in your code and the dynamic elements in your template. For example in the HelloWorld application, the HTML template for the Main component contains two dynamic elements. The declarations file specifies that the first dynamic element represents a text field whose value maps to the **visitorName** variable in the component's script. When the user types a name in the text field, WebObjects assigns it to the **visitorName** variable. The declarations file also specifies that the second dynamic element is a submit button and that when the user clicks the button, WebObjects invokes the **sayHello** method.

Application Code

In addition to having one or more components, your application can also include *application code*. In application code, you declare and initialize application variables and perform tasks that affect the entire application.

The application code file is named `Application` and its extension is based on the programming language you use to create it (`Application.wos`, `Application.java`, or `Application.m`).

Session Code

Sessions are periods during which one user is accessing your application. Because users on different clients may be accessing your application at the same time, a single application may have more than one session accessing it at a time (see Figure 3). Each session has its own copy of the components that its user has requested.

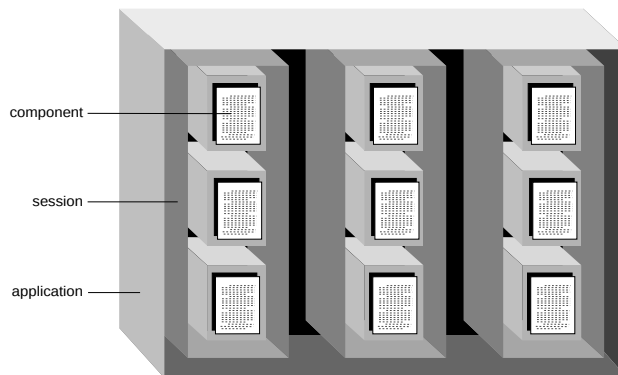


Figure 3. Application, Sessions, and Components

You perform tasks that affect a single session and store variables that persist throughout a session in the *session code file*. The session code file is named `Session` and has the appropriate extension (`Session.wos`, `Session.java`, or `Session.m`).

A Note on WebObjects Classes

True to its name, WebObjects is an object-oriented environment for writing web applications. Therefore, when you write a component, an application file, or a session file, you are really writing a class. This is true whether you use WebScript, Java, or Objective-C. You can learn about these classes and

where they are used by reading the chapter “WebObjects Viewed Through Its Classes” (page 63).

Components are subclasses of a class named `WOComponent`. For example, in Figure 1 the component directory creates a `WOComponent` subclass named `Main`. Application files create subclasses of a class named `WOApplication`, and session files create subclasses of a class named `WOSession`.

`WOComponent`, `WOApplication`, and `WOSession` are defined, along with other classes, in the WebObjects Framework in `NeXT_ROOT/NextLibrary/Frameworks/WebObjects.framework`. (`NeXT_ROOT` is an environment variable defined at installation time. On Windows NT systems, it is `C:\NeXT` by default. On Mach systems, the `NeXT_ROOT` environment variable is undefined, but you can think of it as being the root directory.)

In Java, WebObjects classes have different names. The names shown previously are the WebScript names. (Objective-C uses the same names as WebScript.) In Java, `WOComponent` is called `Component`, `WOApplication` is `WebApplication`, and `WOSession` is `WebSession`. The Java classes are contained in the package `next.wo`.

Note: This book generally uses the WebScript names for classes and methods. Usually, you can easily discern the Java name from the WebScript name, and vice versa. The following table tells you how to do so.

	WebScript/Objective-C Name	Java Name
Class names	<i>WOClass</i>	<i>Class</i> (or <i>next.wo.Class</i>)
Zero-argument methods	<i>method</i>	<i>method</i>
Single-argument methods	<i>method:</i>	<i>method</i>
Multiple-argument methods	<i>methodWithArg1:arg2:</i>	<i>methodWithArg1</i>

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

Application Directory

When you build a WebObjects application project, the result is a directory with the extension `.woa`. This directory contains the application executable, copies of the component files, copies of the script files (if any), and any resources that the application or the HTTP server needs access to. When you’re ready to deploy an application to your users, you use the `.woa` to run the application.

Now you've learned what a WebObjects application looks like and seen the pieces that you'll have to write. The next section tells you how to run a WebObjects application.

Running a WebObjects Application

WebObjects applications run on a web server. Your users connect to a WebObjects application using web browsers that they run on their own (client) machines. How does a user start a WebObjects application, and how does the application communicate with the browser?

Users run a WebObjects application using a Uniform Resource Locator (URL) similar to the one shown in Figure 4. (Of course, you'd probably provide a button or a link on a static web page that would take users to this URL rather than forcing your users to type such a long string.)

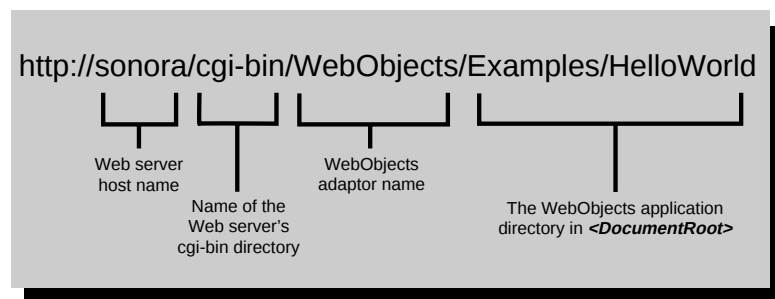


Figure 4. A URL to Start a WebObjects Application

To start your own applications, you open a command shell window, go to the directory that contains your application, and enter the application command. WebObjects starts up your application, opens the web browser, and enters the URL in the web browser for you. For example, to start the Java version of HelloWorld, go to the directory

`<DocRoot>/WebObjects/Examples/Java/HelloWorldJava/HelloWorldJava.woa`, which contains the executable file, and enter **HelloWorld** on the command line. On Windows NT, you can simply navigate to this directory in the Explorer and double-click the **HelloWorld.exe** file.

When you run a WebObjects application, it communicates with the web browser through the chain of processes shown in Figure 5.

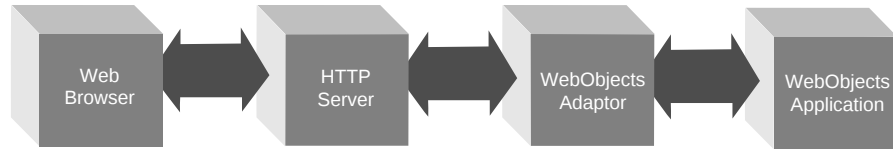


Figure 5. Chain of Communication Between the Browser and Your WebObjects Application

Here is a brief description of these processes:

- **An HTTP server.** Any HTTP server that uses the Common Gateway Interface (CGI), the Netscape Server API (NSAPI), or the Internet Server API (ISAPI).
- **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 executable.** The application executable receives incoming requests and responds to them, usually by returning a dynamically generated HTML page.

Two of these, WebObjects adaptors and WebObjects application executables, are described next.

WebObjects Adaptors

A WebObjects adaptor receives requests from the server, repackages the requests in a standard format, and forwards them to an appropriate WebObjects application (see Figure 6).



Figure 6. 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, the Netscape Interface adaptor uses the Netscape Server API (NSAPI), and the Internet Server adaptor uses the

Internet Server API (ISAPI). Thus, WebObjects adaptors can 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—including those that are publicly available. As demands on performance increase, switch to one of the other adaptors with a server that supports the corresponding API (Netscape Server API or Internet Server API). Such servers are capable of dynamically loading the adaptor, eliminating the overhead of starting a new process for each request. As shown in Figure 7, the communication between the adaptor and the HTTP server occurs inside a single process.

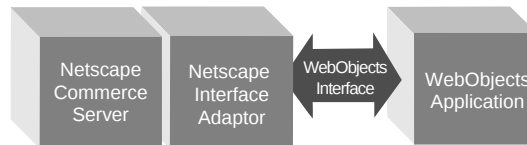


Figure 7. The Netscape Interface Adaptor

The online document *Serving WebObjects* describes how to configure a WebObjects adaptor.

The WebObjects Application Executable

An application executable is an executable file, provided by you or by WebObjects, that receives incoming requests from the adaptor and responds to them, usually by returning a dynamically generated HTML page.

If your application is written entirely in WebScript, it can use the default application executable, `NeXT_ROOT/NextLibrary/Executables/WODefaultApp`, provided as part of the WebObjects package. If your application contains compiled code, you build your own executable and use it in place of `WODefaultApp`.

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

associated with the user's action, and generates a response—usually an HTML page (see Figure 8).

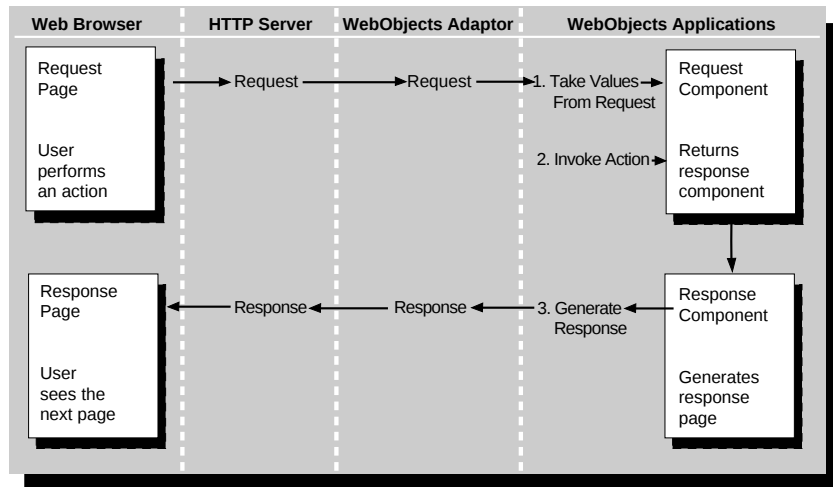


Figure 8. The Request-Response Loop

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 runtime.

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 runtime. 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, look at the CyberWind sample application. It's located in `<DocRoot>/WebObjects/Examples/Java/CyberWindJava`, where `<DocRoot>` is your web server's document root. When you run CyberWind, its first page contains a list of hyperlinks, shown in Figure 9.

Choose between the following menu options:

[See surfshop information](#)
[Buy a new sailboard](#)

Figure 9. CyberWind Main Page

This list is not hard-coded into the page. Instead, it is produced by several dynamic elements. Figure 10 shows how this same part of the page looks in WebObjects Builder.

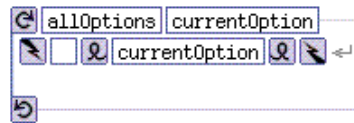


Figure 10. CyberWind Main Page in WebObjects Builder

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 runtime, and the WOString is a string whose contents are determined at runtime.

When you run CyberWind, 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 returns 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.

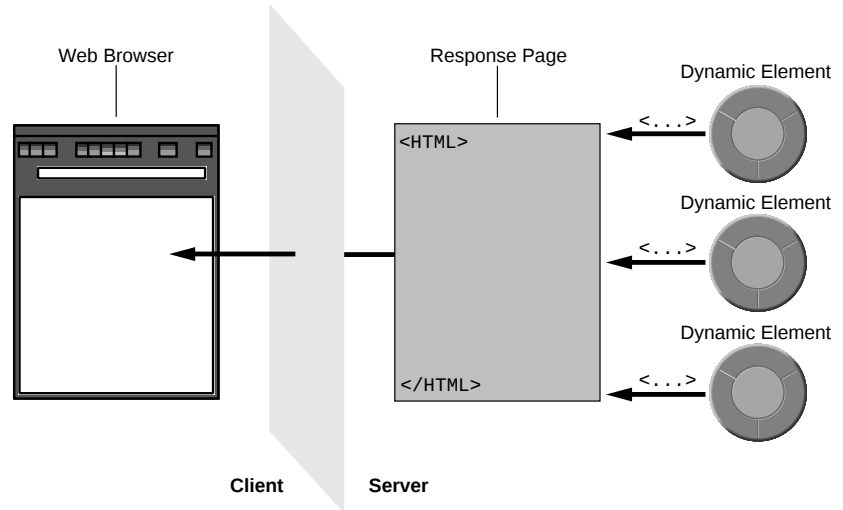


Figure 11. Server-Side Dynamic Elements

How Server-Side Dynamic Elements Work

To learn how server-side dynamic elements work, look once more at the Main page of the CyberWind example in WebObjects Builder. If you switch over to raw mode, you see that the Main page contains this HTML code in its template:

```
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).

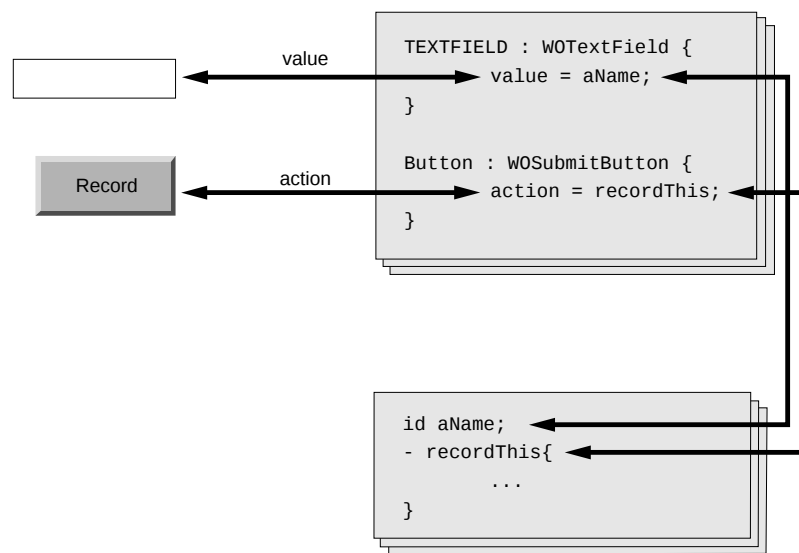


Figure 12. Dynamic Element Bindings

In the CyberWind 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). CyberWind's 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 CyberWind 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. Look at CyberWind's Footer component in WebObjects Builder. This component displays, among other information, the date and time the CyberWind application was started. This date is stored in the application object, not in the Footer component. The Footer component's **.wod** file contains 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 **takeValue** 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 Footer 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 (Component in Java) defines the **application** method, which returns the WOApplication object (WebApplication in Java).

- 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 };
```

- 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 };
```

Notice that the last attribute/value pair before a closing brace {} does not end with a semicolon (;).

As described in the previous section, *value* can be a constant, variable, or method. It can also be a string of messages 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 can 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. This is not the case with the client-side Java components provided with WebObjects. While these applets run on the client, they 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.

To add a client-side component to your application, you drag it from the palette provided in WebObjects Builder. For a list of the client-side Java components that WebObjects provides, see the online book *Client-Side Applet Controls Reference*.

Deciding When to Use Client-Side Components

You should use client-side components whenever you want greater control over the appearance of your application. In general, client-side components have these advantages over server-side dynamic elements:

- Client-side components define a state-synchronization phase that does not reload the page.

As you learned in the first chapter, 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 resynchronization of state or the return of a new page. Thus, state can be synchronized without the page having to be redrawn (see Figure 13).

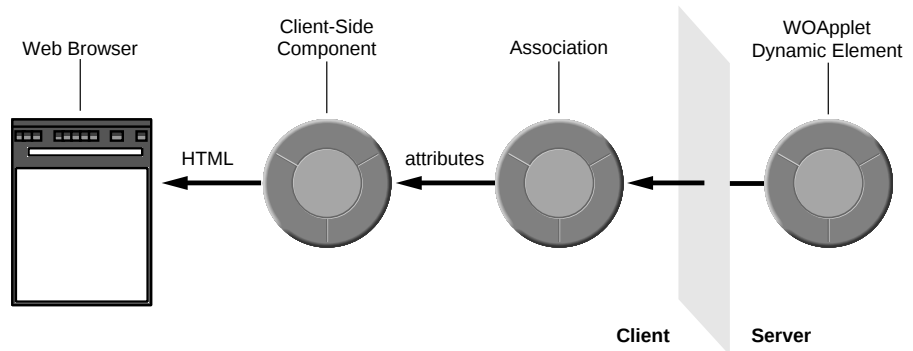


Figure 13. Client-Side Java Components

- 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. To learn how to create a client-side component, see the chapter “Creating Client-Side Components” (page 141).

The 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 use 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";
    width = "200";
    height = "20";
    associationClass = "next.wo.client.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 preloaded 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 **associationClass** attribute differentiates the client-side components from any other applet you might include in your application. This attribute specifies an object (a subclass of **next.wo.client.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. For the WebObjects-provided client-side components, this attribute is always **next.wo.client.SimpleAssociation**. If you create your own client-side components, you provide your own Association subclass.

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.

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, and WOComponent (in Java, WebApplication, WebSession, and Component), 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 63).

As you're writing methods, refer to the class specifications for WOApplication, WOSession, and WOComponent 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. To associate your method to a user action, you map it to a dynamic element that has an attribute named **action**. (In the examples just given, the dynamic elements associated with the user actions are WOSubmitButton, WOActiveImage, or WOHyperlink.) When the user performs the associated action, your method is invoked.

For example, in the HelloWorld example application (in `<DocRoot>/WebObjects/Examples/WebScript/HelloWorld`, where `<DocRoot>` is your web server's document root), the submit button is mapped to a method named **sayHello** in the Main component. When users see this page, they type in a

name and click the button. This 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 = [WObj pageWithName:@"Hello"];
    [nextPage setVisitorName:visitorName];
    return nextPage;
}
```

If you're programming in Java, you can look at the HelloWorldJava example, which is identical to HelloWorld but written in Java. Its **sayHello** method looks like this:

```
//Java HelloWorld Main.java
public Component sayHello() {
    Hello nextPage = (Hello)application().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 application] pageWithName:@"Error"];
        [errorPage setErrorMessage:@"Invalid part number %@.",
            partnumber];
        return errorPage;
    }
    inventoryPage = [[self application] pageWithName:@"Inventory"];
    [inventoryPage setPartNumber:partnumber];
    return inventoryPage;
}
```

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

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

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, in the Visitors application, the `recordMe` action method in the Main page records the name of the last visitor, clears the text field, and redraws itself:

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

// Java Visitors Main.java
public Component 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`.

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 methods are `init` and `awake`. In Java, the initialization methods are the constructor for the class and `awake`.

Both `init` and `awake` perform initialization tasks, but they are invoked at different times during an object's life. The `init` message (or the constructor in Java) 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` and `init` are the `sleep` and `dealloc` methods. These methods let objects deallocate their instance variables and perform other clean-up tasks. The `sleep` method is invoked at the end of each cycle of the request-response loop, whereas the `dealloc` method is invoked at the end of the object's life.

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 Structures of `init` and `awake`

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

```
- init {
    [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 */
}
```

The `awake` method has no such structure. In it, you don't need to send a message to `super` or return anything.

```
- awake {  
    /* initializations go here */  
}  
  
public void awake () {  
    /* 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, the Visitors application has this **init** method in **Application.wos**:

```
// WebScript Visitors Application.wos  
- init {  
    [super init];  
    lastVisitor = @"";  
    [self setTimeout:7200];  
    return self;  
}  
  
// Java Visitors Application.java  
public Application () {  
    super();  
    ...  
    lastVisitor = "";  
    setTimeout(7200);  
    ...  
}
```

This method begins by calling the application'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 application to terminate after it has been running 2 hours.

This example sets the application time-out value. 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 109).

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, the DodgeDemo example application keeps track of the number of requests the application has received. It increments and logs that number at the top of the request-response loop:

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

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

Session Initialization

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

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, in the CyberWind application, each session keeps track of which number it is. These values are changed in the session object's **init** method:

```
//From CyberWind Session.wos
- init {
    [super init];
    [self setTimeout:120]; // session idle time is 2 minutes.
    [[self application] setSessionCount:[self application]
        sessionCount + 1];
    sessionNumber = [[self application] sessionCount];
    return self;
}

//From CyberWindJava Session.java
public Session() {
    super();
    Application application = (Application)application();
    this.setTimeout(120);
    application.setSessionCount(application.sessionCount() + 1);
    sessionNumber = application.sessionCount();
}
```

The session object's **awake** method is invoked each time the user associated with the session makes a new request. After the application object has performed its own **awake** method, it restores the appropriate session object and sends it the **awake** message too.

The CyberWind application keeps track of the number of requests per session. It increments the number in the session's **awake** method.

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

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 63). 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:** method shown in the section “Action Methods” (page 43) always creates a new component object, even if page caching is turned on.

A component object's **init** method usually initializes component variables. For example, in the *EmployeeBook* example, the **Department.wos** script uses **init** to initialize the **departments** component variable:

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

    [super init];
    departmentsPath = [[self application]
        pathForResourceNamed:@"Departments" ofType:@"array"];
    departments = [NSArray arrayWithContentsOfFile:departmentsPath];
    return self;
}
```

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, in the *DodgeDemo* application, the **Car.wos** script uses **awake** to initialize the **shoppingCart** component variable:

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

In general, you use **init** to initialize component instance variables instead of **awake**. The reason is that **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 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 109).

Request-Handling Methods

Request-handling is performed in three phases, which correspond to three methods that 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` or `Request` in Java) 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, the URL, and form values.
- A context object (`WOContext` or `Context` in Java) is passed as an argument in all three phases. This object represents the current context of the application. It contains references to information specific to the application, such as the path to the request component’s directory, the version of WebObjects that’s running, the application name, and the request page’s name.
- A response object (`WOResponse` in Java) 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.

Taking Input Values From a Request

The **takeValuesFromRequest:inContext:** method is invoked during the first phase of the 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.

Override **takeValuesFromRequest:inContext:** when you want to do one of the following:

- Access information from the request or context object.
- Perform postprocessing on user input.

In the first case, you can place your code before the message to **super**. In the second case, you must place your code after the message to **super**. For example, the following implementation of **takeValuesFromRequest:inContext:** records the kinds of browsers—user agents—from which requests are made:

```
// WebScript example
- takeValuesFromRequest:request inContext:context {
    id userAgent = [request headerForKey:@"user-agent"];
    [self recordUserAgent:userAgent];
    [super takeValuesFromRequest:request inContext:context];
}
```

The following example performs postprocessing. It takes the values for the **street**, **city**, **state**, and **zipCode** variables and stores them in the **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(Request request, Context context) {
    super.takeValuesFromRequest(request, context);
    address = street + city + state + zipCode;
}
```

Invoking an Action

The second phase of the 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).

Use **invokeActionForRequest:inContext:** 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.

The following example, implemented in **Session.wos**, returns a “CreditCard” page if the user hasn’t supplied this information yet:

```
// WebScript example
- invokeActionForRequest:request inContext:context {
    id creditPage;
    id responsePage = [super invokeActionForRequest:request
        inContext:context];
    id nameOfNextPage = [responsePage name];

    if ([self verified]==NO &&
        [nameOfNextPage isEqual:@"Fulfillment"]) {
        creditPage = [[self application]
            pageWithName:@"CreditCard"];
        [creditPage setNameOfNextPage:nameOfNextPage];
        return creditPage;
    }
    return responsePage;
}

//Java example
public Element invokeActionForRequest(Request request, Context context)
{
    Component creditPage;
    Component responsePage = super.invokeActionForRequest(request,
        context);
    String nameOfNextPage = responsePage.name();

    if (verified()==false &&
        (nameOfNextPage.compareTo("Fulfillment") == 0) {
        creditPage = application().pageWithName("CreditCard");
        creditPage.setNameOfNextPage(nameOfNextPage);
        return creditPage;
    }
    return responsePage;
}
```

When the application receives a request for a new page (say, a fulfillment page), the session object determines whether or not the user has supplied valid credit-card data by checking the value of its **verified** variable. If the value of **verified** is **NO**, the session object returns the “CreditCard” component. As shown in the

following action method, the “CreditCard” component sets the **verified** session variable to YES when the user has supplied valid credit information and returns the user to the original request page to try again.

```
- verifyUser {
    if ([self isValidCredit]) {
        [[self session] setVerified:YES];
        return [[self application] pageWithName:nameOfNextPage];
    }
    return nil;
}
```

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/-/Hello.wo/
```

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, you must implement another strategy.

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

```
- appendToResponse:aResponse inContext:aContext
{
    [super appendToResponse:aResponse inContext:aContext];
    [aResponse setHeader:@"True"
        forKey:@"dshttpd-NoAutomaticFooter"];
}
```

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 appendContentHTMLAttributeValue:@"<p>"];
    if (isBold) {
        [aResponse appendContentHTMLAttributeValue:@"<b>"];
    }
    if (isItalic) {
        [aResponse appendContentHTMLAttributeValue:@"<i>"];
    }

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

    if (isItalic) {
        [aResponse appendContentHTMLAttributeValue:@"</i>"];
    }
    if (isBold) {
        [aResponse appendContentHTMLAttributeValue:@"</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(response, 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 63).

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. When you debug, you'll be using the Project Builder application. To learn how to use Project Builder, see the online book *WebObjects Tools and Techniques*.

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 58).

If you've written an application entirely in WebScript, you typically debug it by running `NeXT_ROOT/NextLibrary/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

The debugging strategy for Java applications is very similar to the strategy for debugging WebScript applications. Because the WebObjects Java bridge is incompatible with `jdb`, no Java debugger is supported for WebObjects. Instead, you can use the methods described in the section “Debugging Techniques” (page 58) 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 `logWithFormat:` and `WOApplication` trace statements as described in "Debugging Techniques" (page 58).

Debugging Mixed Applications

When you build a WebObjects application project, the result is a `.woa` file package inside of the project directory. You may notice that this file package contains all of the application's components (including scripted components), and all other resources need to run the application, as well as the application executable itself.

When you're debugging an application, the executable uses the components from the project directory instead of those in the `.woa` package, so you can safely ignore the components placed inside of the `.woa` package. When you need to make a change, change the component in the project directory. When you run an application, it checks to see if its `.woa` package is inside of a project directory (that is, a directory that contains a file named `PB.project`). If it is, the application takes its scripted components from the project directory. This way, you can make any necessary changes to your scripts in Project Builder, and (once you have saved the scripts) your application automatically picks up your changes without your having to rebuild.

Debugging Techniques

To debug WebScript and Java code, 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.

Writing Debug Messages

The method `logWithFormat:` (in Java, `logString`) writes a formatted string to standard error (`stderr`).

In WebScript and Objective-C, `logWithFormat:` 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 logWithFormat:@"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 `logWithFormat:` is `logString`, and you can send it only to WebApplication objects. Instead of using `printf` specifications, it uses concatenation. Here’s how you’d write the same lines of code in Java:

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

Perhaps the most effective debugging technique is to use `logWithFormat:` 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 logWithFormat:@"The contents of self in sayHello are %@", self];
```

produces output that resembles the following:

```
The contents of self in sayHello are
<<WOScriptedClass(/WebObjects/Examples/WebScript/HelloWorld.woa/Main
.wo/Main): 0x8cb08 name=Main subcomponents=0x0> visitorName=frank>
```

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

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

Using Trace Methods

WOApplication (in Java, WebApplication) 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.
– 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:

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

Isolating Portions of a Page

If a component is producing unexpected HTML output, you can try to isolate the problem by printing 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:` method.

```
NSString *string1, *string2;

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

// Right
if ([aString1 isEqual: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.
```

- WebScript always evaluates both sides of a Boolean expression (such as && and ||). You should make sure that the second half of an expression does not produce an error.

```
// WRONG! produces a divide by 0 if a is 0.
if ((a == 0) || (b / a) > 5) ...
```

For more information, see the chapter “The WebScript Language” (page 163).

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 `WebApplication`, `WebSession`, `Component`, 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 Component {
    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 **Component** 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 **WebSession** object. Because **WebSession** does not define a method named **verify**, your code won't compile unless you cast the return value of **session** to your **WebSession** subclass.

WebObjects Viewed Through Its Classes

As you learned at the end of the first chapter, WebObjects applications respond to HTTP requests from the server 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.

In the chapter “Common Methods” (page 41), 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 incoming message (URL) 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 14.

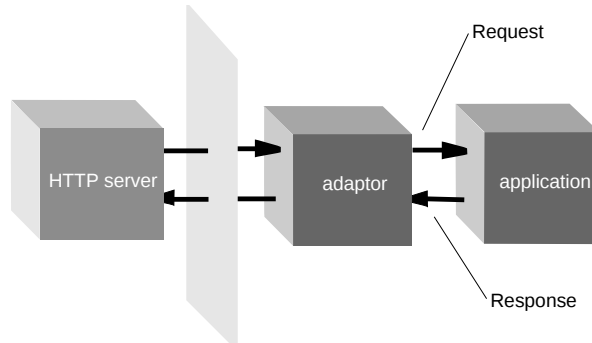


Figure 14. Request-Response Loop: Application and Server Level

The HTTP server sends 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 initiates and manages the process of request handling and returns the completed response 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** (in Java, **Adaptor**)
Defines the interface for objects mediating the exchange of data between an HTTP server and a WebObjects application. This is an abstract class.
- **WOApplication** (in Java, **WebApplication**)
Receives requests from the adaptor and initiates and coordinates the request-handling process, after which it returns a response to the adaptor. WOApplication also creates dynamic elements “on the fly” and manages adaptors, sessions, application resources, and components.

Session Level

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

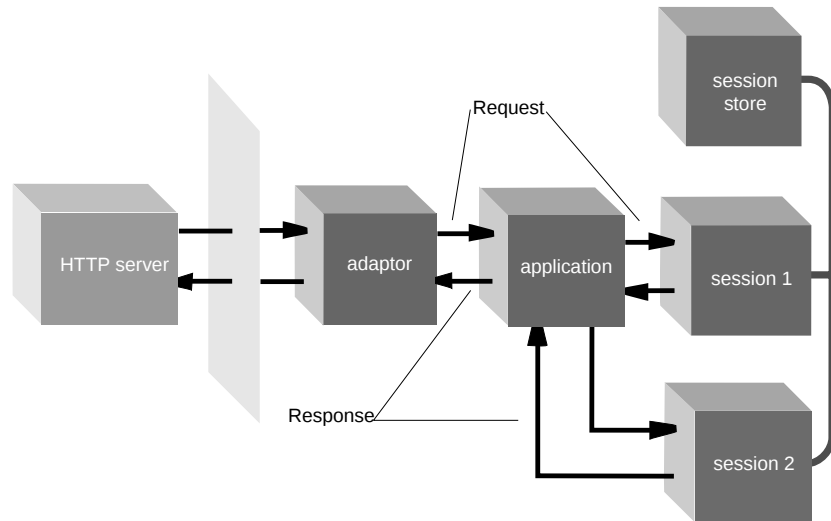


Figure 15. Request-Response Loop: Session Level

The objects dedicated to session management ensure that state with sessionwide scope persists between cycles of the request-response loop.

Two classes are involved at this level:

- WOSession (in Java, WebSession)

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 109). Each session object is identified by a unique session ID, which is reflected in the URL.

- WOSessionStore (in Java, SessionStore)

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 (which can include Netscape cookies), and restores them upon request by the application.

When a user makes an initial request to a WebObjects application, 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 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 109).

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 16 shows how WebObjects requests are handled at the transaction level.

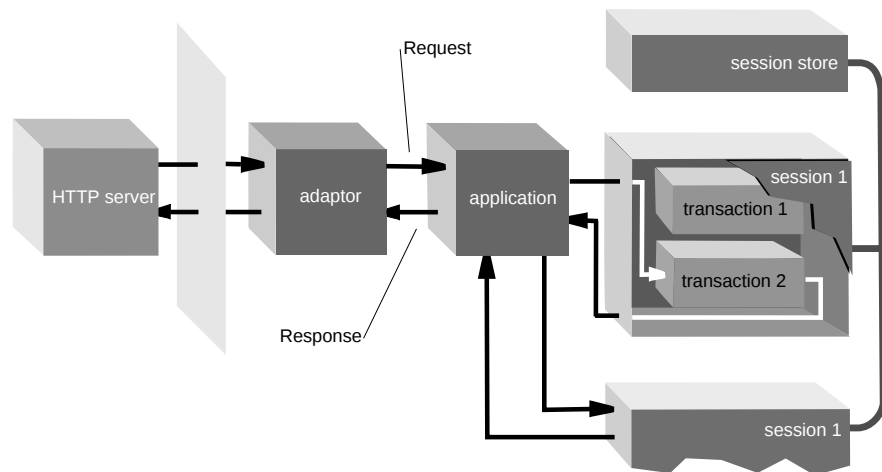


Figure 16. Request-Response Loop: Transaction Level

Three classes are involved at this level:

- WOREquest (in Java, Request)

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** (in Java, **Response**)

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** (in Java, **Context**)

Provides access to the objects involved in the current cycle, such as the current request, response, session, and application objects. It also stores the 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 association. See “How HTML Pages Are Generated” (page 82) for an explanation. The WOContext object acts as a “cursor,” traversing the object graph during each phase of the request-response loop. The WOContext for a cycle is identified by a unique context ID, which appears in the URL.

You rarely need to work directly with WORequest, WOResponse, and WOContext yourself. At the beginning of the request-response loop, the WOAdaptor and WOApplication objects create instances of these three classes. The application initiates each phase of the request-response loop by sending the messages **takeValuesFromRequest:inContext**, **invokeActionForRequest:inContext**, and **appendToResponse:inContext**: (in Java, **takeValuesFromRequest**, **invokeAction**, and **appendToResponse**). It passes in the WORequest, WOResponse, and WOContext objects as arguments to one or more of these methods. 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 72) 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 17). Many of the same objects also set their variable values from data entered into request pages and respond to user actions.

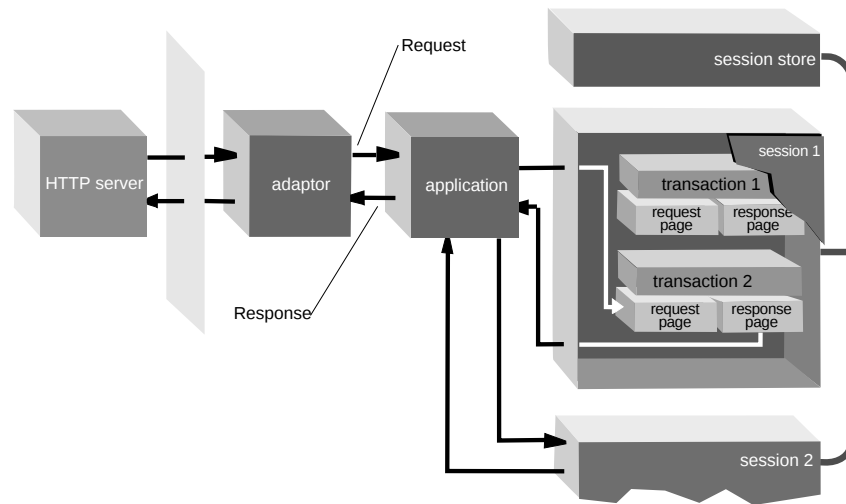


Figure 17. Request-Response Loop: Page Level

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 82).

Four classes are involved at this level:

- `WOComponent` (in Java, `Component`)
Represents an integral, reusable page (or portion of a page) for display in a web browser.
- `WOElement` (in Java, `Element`)
Declares the three 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` (in Java, `DynamicElement`)
An abstract class for subclasses that generate particular dynamic elements.

- WOAssociation (in Java, Association)

Knows how to find and set a value by reference to a key. Instance variables and action methods of dynamic elements are instances of this class.

Database Integration Level

Database integration is handled mainly by classes in the Enterprise Objects Framework (see Figure 18). 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.

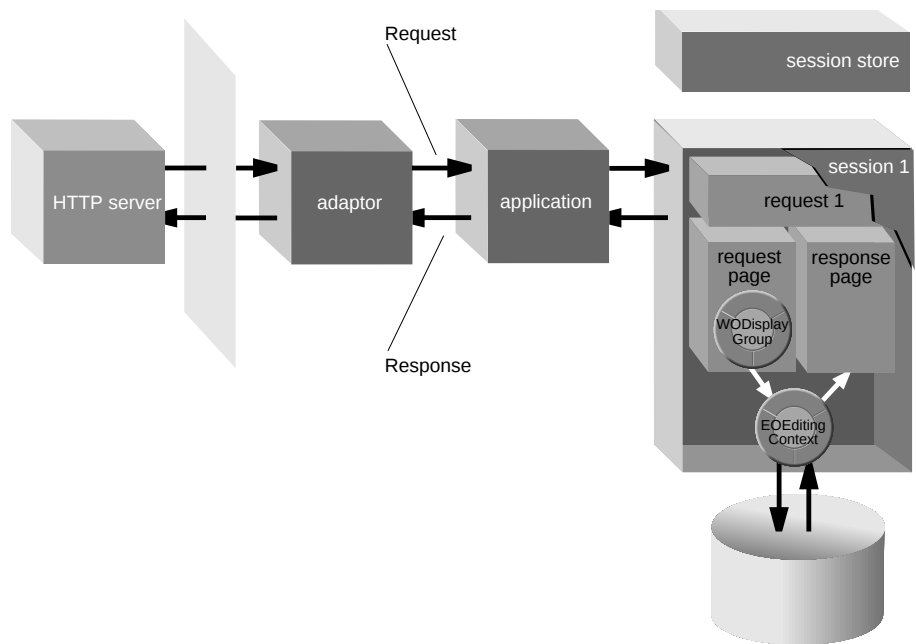


Figure 18. Request-Response Loop: Database Access

Two classes are involved at this level:

- WODisplayGroup (in Java, DisplayGroup)

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, `next.eo.EditingContext`)

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 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 can start up in one of two ways: automatically, when it receives a request (autostarting), or manually, when it's run from the command line. Either way, its 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 begins by creating an autorelease pool that's used for the automatic deallocation of objects that receive an `autorelease` message. It then calls a function that loads the Java Virtual Machine (VM) if necessary.

The next step is to create a WOApplication (or WebApplication) object. This seems fairly straightforward, but in the `init` method or constructor the application creates and stores, in an instance variable, one or more adaptors. These adaptors, all instances of a WOAdaptor subclass, handle communication between an

HTTP server and the WOApplication object. The application first parses the command line for the specified adaptors (with necessary arguments); if none are specified, as happens when the application is autostarted, it creates a suitable default adaptor.

The `run` method initiates 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. 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.

Taking Values From the Request

The first phase of the request-response loop (see Figure 19) 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.

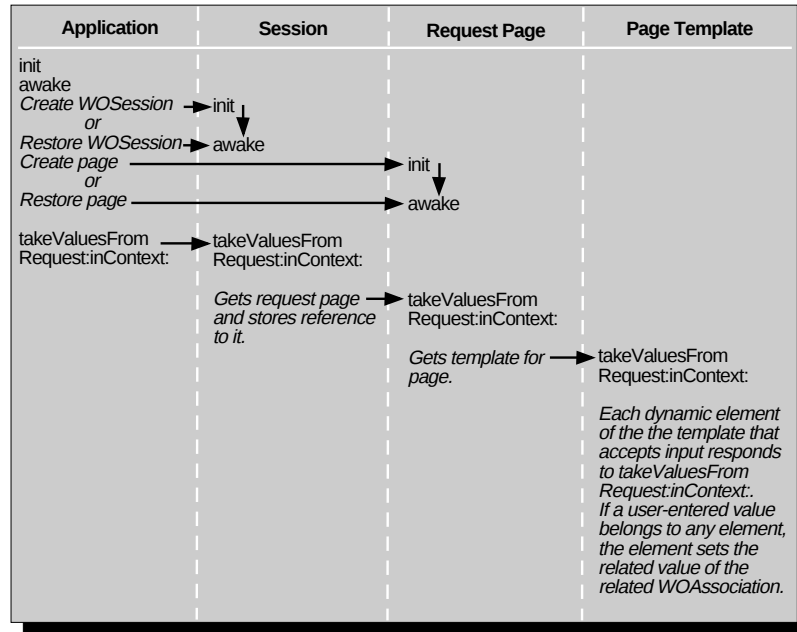


Figure 19. Taking Values From the Request

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` and forwards this object to the application object in a `handleRequest:` message. Upon receiving this message, the application object does the following:

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

Accessing the Session

The application determines whether to create a new session or access an existing session by searching the request URL (which was passed in as an argument to the `handleRequest:` method) 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.

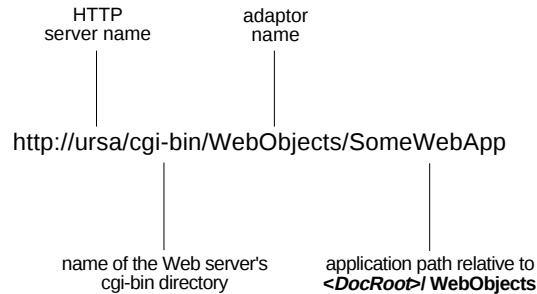


Figure 20. URL to Start a WebObjects Application

This URL does not contain a session ID, so the application object creates a new session by performing the following steps:

1. It sends itself a **createSession** message.
2. As part of the **createSession** method, it sends the **init** message or the constructor message to the **WOSession** (or **WebSession**) class to create a new session object.
3. It 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 21.

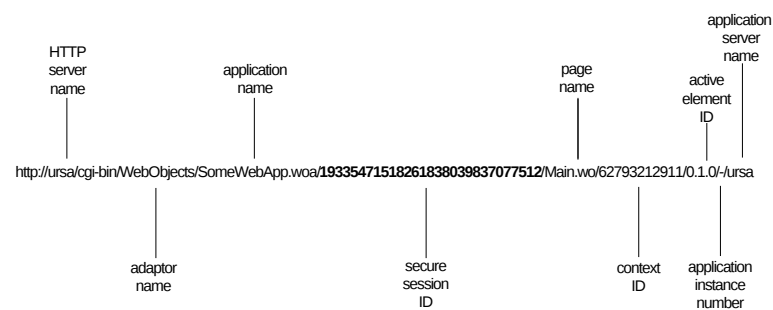


Figure 21. WebObjects URL in an Existing Session

This URL contains all of the information necessary to restore the state of the existing session. The session ID comes right after the application name in the URL. 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 32-digit integers as session IDs. (You can also override `WOSession`'s `sessionID` method and implement another security scheme if you'd like.)

The application keeps existing, active sessions in the `WOSessionStore` object. The application object uses the session ID to retrieve the appropriate session from the session store (see Figure 22). The appropriate session object is then sent the `awake` message to prepare it for the request.

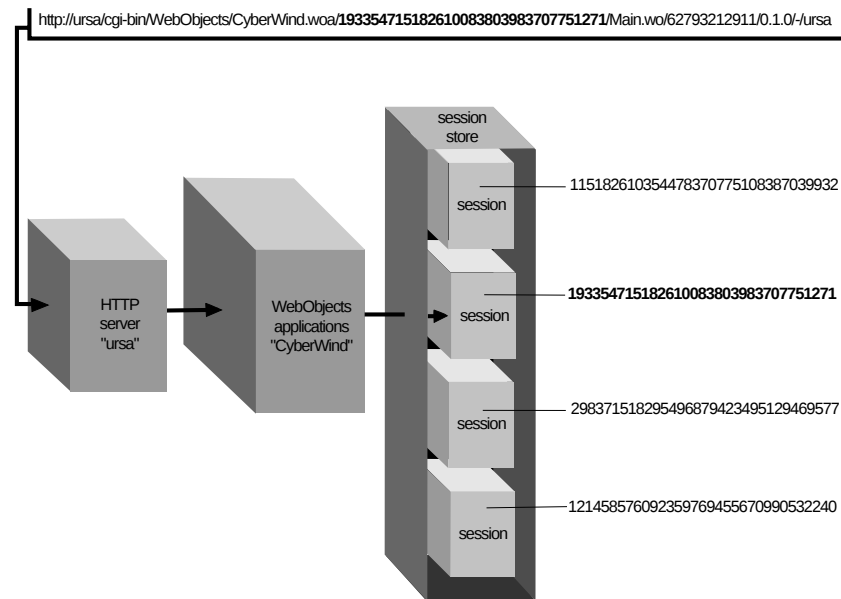


Figure 22. Associating a Request With a Session Object

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.)

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 runtime system, it looks for a scripted component with the name of the request page. 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 21). 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 in addition to the page name. 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.

Assigning Input Values

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. Here is the basic sequence of events in preparing for a request:

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 82).

Invoking an Action

In the second phase of the request-response loop (see Figure 23), 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.

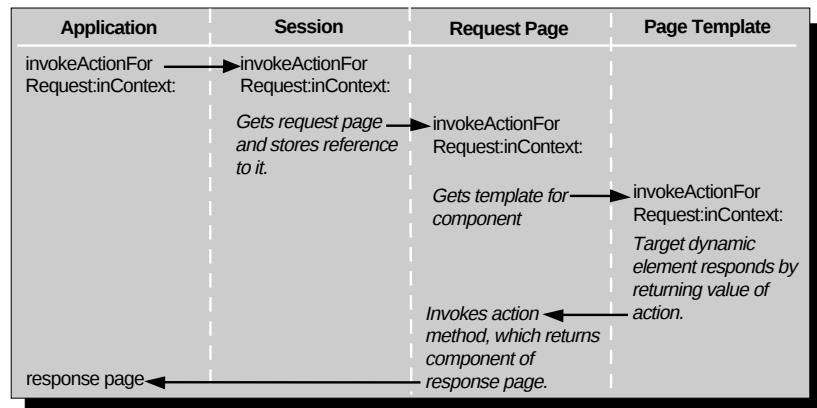


Figure 23. Invoking an Action

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.

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 82).

Generating the Response

In the final phase of request-response loop (see Figure 24), 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.

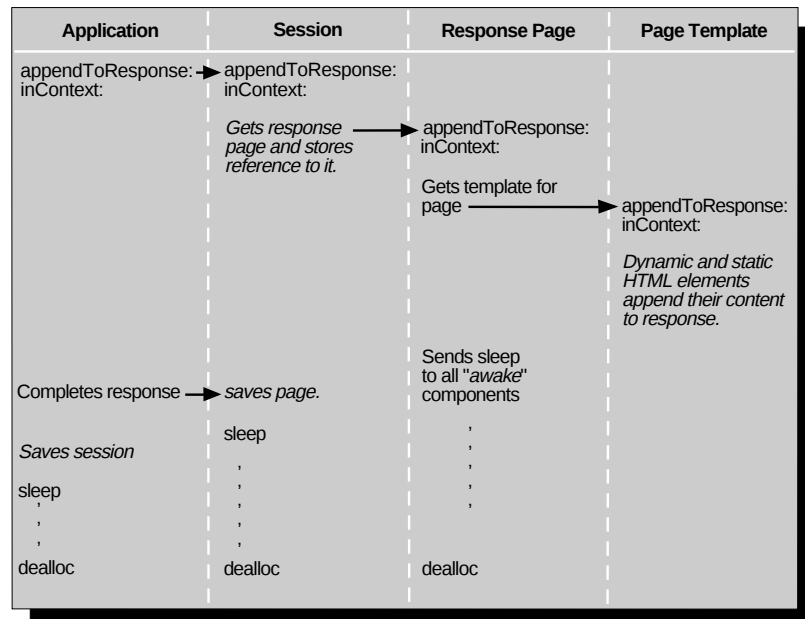


Figure 24. Generating the Response

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 the second phase of the request-response loop.)
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.

After the response has been generated, but before returning the response to the adaptor, the application object concludes request handling by doing the following:

1. It causes the **sleep** method—the counterpart of **awake**—to be invoked in all components involved in the cycle (request, response, and subcomponents). As described in the chapter “Managing State” (page 109), 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 24). 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.

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** (in Java, **Element**). **WOElement** declares, but does not implement, the three 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 25). 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.

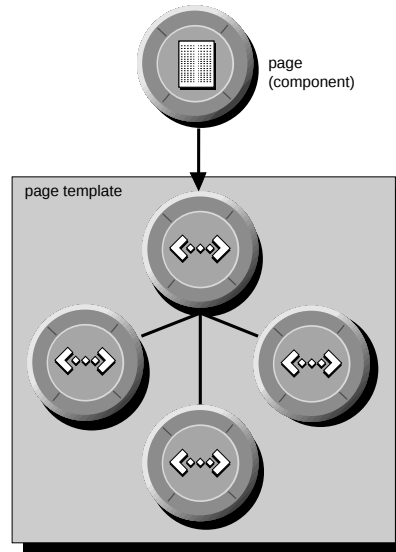


Figure 25. An Object Graph for a Page's Template

The template is created at runtime 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.

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. An element ID is implemented as an extension of the sender ID in the URL. 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` (in Java, `Association`). `WOAssociation` objects know how to obtain and set the value they represent. They generally do this using key-value coding.

The key to a value can be represented as a sequence of keys separated by periods. The resolution of a key by yielding its value makes possible the resolution of the next key. For instance:

```
self.aRepetition.list.item
```

means that `self` (identifying the current component) has a `WORepetition` named `aRepetition`. The `list` key denotes the list of elements displayed by the `WORepetition`, and the `item` is the key to the current item in that list. 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 86) for more on this. `WOContext` stores the current component on a stack, “pushing” and “popping” components onto and off of the stack as necessary.

Depending on the phase of the request-response loop, 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” `WOAssociation`, the appropriate action method in the current component 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 is two-way. This two-way binding allows the synchronization of state between the two components. Consider this declaration in `Main.wod` of the `TimeOff` example:

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

In this example, Main is the parent component and Calendar is the child component. The **startDate** variable 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 (**takeValuesFromRequest:inContext**, **invokeActionForRequest:inContext**, and **appendToResponse:inContext**). Synchronization is performed through the accessor methods of both components.

This aspect of synchronization has implications for developers. Because WebObjects invokes explicitly implemented accessor methods many times during the same 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 91) for more on state synchronization between child and parent components.

Associations and Client-Side Java Components

Client-side Java components, like server-side dynamic elements, use associations to synchronize state with the parent component. However, the association class they use is not the same. Instead of using **next.wo.Association** (the WOAssociation equivalent in Java), they use **next.wo.client.Association**, and this Association class is downloaded to the client along with the component itself.

Keys for a client-side component fall into two groups: state bindings and action bindings. *State bindings* form the basis for state synchronization by associating state in the applets with state in the server. *Action bindings*

associate particular events in the client applet (such as clicking a button) with the invocation of methods in the server.

State is synchronized between the client and the server in three phases:

1. When a page is first generated, the server sends the client all state for which there are bindings.
2. Before an action is invoked in the server, the client sends the server any of its state that has changed.
3. After the action is completed, the server sends the client any of its state that has changed.

This last synchronization occurs only if no new page is returned to the browser. When a method invoked remotely through an applet action binding returns `null`, it signals that, instead of returning a new page, the server should resynchronize its state with the applets on the page. WebObjects takes a snapshot of the changes in state in the server so that only the state that has changed is sent back to the client.

Note: The last two phases of the synchronization cycle can be initiated only on the browser side. That is, except for the first “initialization” phase, the server component can react only to an action triggered in an applet. The component cannot unilaterally update the state of an applet when its own state changes.

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 runtime, the component reference binds itself to the separate instances. Figure 26 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 the chain of request-handling messages traverses an object graph and reaches the component reference, it resolves references to its instances according to the element ID of each instance. Components keep track of all their subcomponents by storing them in an internal dictionary using element IDs as keys.

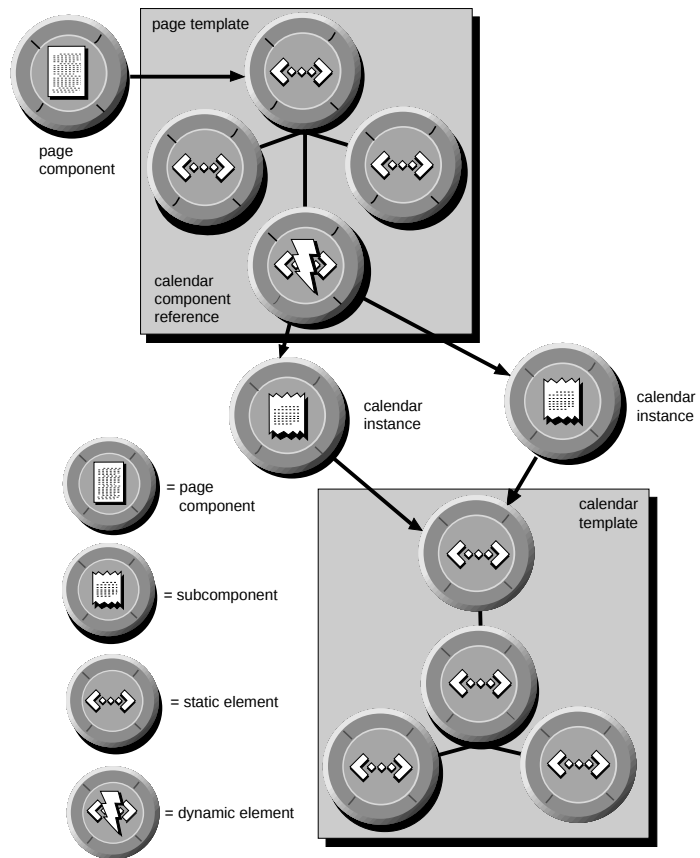


Figure 26. An Object Graph for a Page With a Subcomponent



SPECIAL TASKS

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 27 at the bottom of each page of your application.



Figure 27. A Navigational Control

The HTML code for one page might look like this:

```
<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.www.com/home.html"> Home </a></TD>
  <TD><A HREF = "http://www.www.com/sales.html"> Sales </a></TD>
  <TD><A HREF = "http://www.www.com/service.html"> Service </a></TD>
  <TD><A HREF = "http://www.www.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 = "WWW0314"};
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 28.

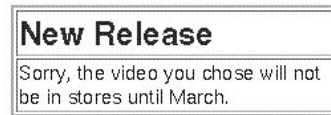


Figure 28. An Alert Panel

This panel is similar to the navigation table shown in Figure 27, 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.

`AlertPanel` is one of several components included in a sample application called `ReusableComponents`. This application demonstrates and documents how to create and use reusable components. If you look at the source code for `AlertPanel`, you'll notice that it's moderately complicated and, in fact, relies on other reusable components for its implementation. However, `WebObjects` lets you think of the `AlertPanel` component as a black box. You simply position the component in your HTML template, specify its attributes in the declarations file, and implement any associated dynamic behavior in your code.

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 28, 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 29.

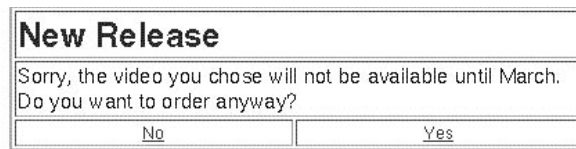


Figure 29. An Alert Panel That Allows User Input

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'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'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'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 30 illustrates the connections between the child and parent components.

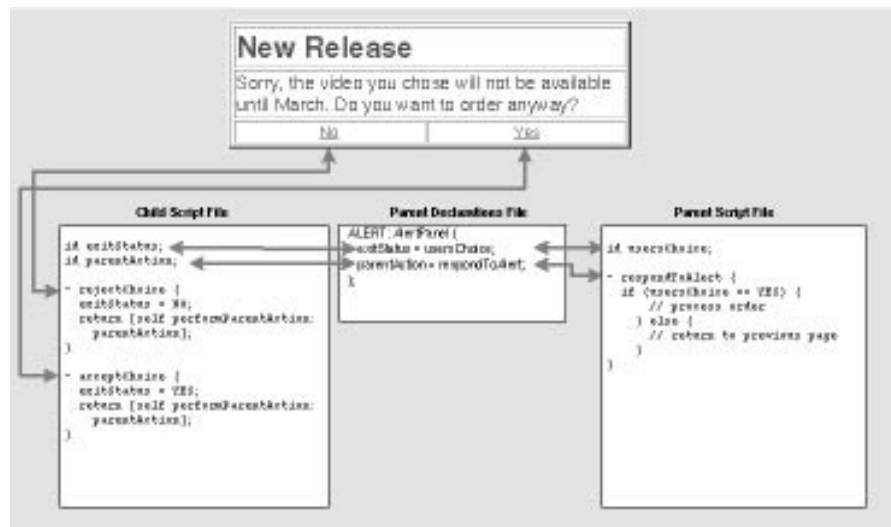


Figure 30. Parent and Child Component Interconnections

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 request-response loop. This synchronization mechanism has some implications for how you design components.

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 31).

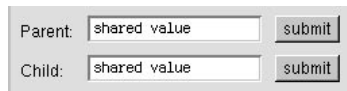


Figure 31. Synchronized Components

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 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 31 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.

Let's say the parent component in the example shown in Figure 31 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.

Sharing Reusable Components Across Applications

If a component is in the WebObjects application's directory, it can be used over and over again, but only within that application. This may be fine for some components, but others, you'll want the option to reuse in 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/NextLibrary/Frameworks* and the WebServer resources in *<DocRoot>/WebObjects/Frameworks*.

You must build the framework even if it contains only scripted components.

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.

The application's executable file must contain all components that your application references, as described in the next section. For this reason, you must build the application even if it contains only scripted components.

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 runtime system for a subclass of WOComponent (in Java, Component) 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**, so any components defined in either of these two frameworks 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 105). 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 unpredictable.

Reusable component names need to be systemwide 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 “Intercomponent Communication” (page 98) 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.

Take, for example, the `AlertPanel` example shown in Figure 28 (page 97). See the `ReusableComponents` example application to view the source code for this component. The `AlertPanel` lets you not only set the message displayed to the user, but also the message's font size and color. These font handling features aren't provided by the `AlertPanel` itself but by an embedded reusable component, `FontString`. `FontString` itself is a versatile component that's used in many other components.

- 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.

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 alternative 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 the chapter "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 online 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.

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. (In Java, these classes are called WebApplication, WebSession, and Component.) 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 application object in the DodgeLite example application (in `<DocRoot>/WebObjects/Examples` where `<DocRoot>` is your web server's document root), keeps information about the cars available and the possible prices:

```

// Java DodgeLite Application.java
public class Application extends WebApplication {
    public ImmutableHashtable dodgeData;
    public ImmutableVector prices;
    public ImmutableVector sortByS;

    public Application() {
        super();
        String filePath = resourceManager()
            .pathForResourceNamedInFramework("DodgeData.dict", null);
        if (null != filePath) {
            try {
                dodgeData = new ImmutableHashtable(new
                    java.io.File(filePath));
            }
            catch (Exception e) {
                //...
            }
        } else {
            // ...
        }
        int priceValues[] = { 8000, 10000, 12000, 14000, 16000, 18000,
            20000, 25000, 30000, 50000, 90000 };
        MutableVector a = new MutableVector();
        for (int i=0; i<priceValues.length; i++) {
            Number num = new Integer(priceValues[i]);
            a.addElement(num);
        }
        prices = (ImmutableVector) a;

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

        for (int i=0; i<sortByStrings.length; i++) {
            a.addElement(sortByStrings[i]);
        }
        sortByS = (ImmutableVector) a;
    }
}

// WebScript DodgeLite Application.wos
id dodgeData;
id prices;
id sortByS;

- init {
    id filePath;

    [super init];
    filePath = [[self resourceManager]
        pathForResourceNamed:@"DodgeData.dict" inFramework:nil];
    if (filePath)
        dodgeData = [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().get(
        "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;
}
```

Sessions can access application state using the same method defined in WOSession.

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();
```

The application object can also access session state using the same method defined in `WOApplication`.

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:@"%s-%s-%s", 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 63), 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 the DodgeLite sample application, which lists models, prices, and types of vehicles for the user to choose from (see Figure 32).

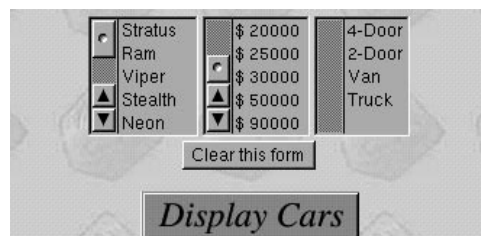


Figure 32. First Page of the DodgeLite Example

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 DodgeLite Main.java
ImmutableVector models, prices, types;
MutableVector selectedModels, selectedPrices, selectedTypes;
String model, price, type;

public Main() {
    super();
    java.util.Enumeration en;

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

// WebScript DodgeLite 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 DodgeLite 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 DodgeLite 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. Looking again at the DodgeLite application, consider what happens when the user makes a selection in the first page and clicks Display Cars. The **displayCars** method in the Main component is invoked:

```

// Java DodgeLite Main.java
public Component displayCars()
{
    SelectedCars selectedCarsPage =
        (SelectedCars)application().pageWithName("SelectedCars");

    ...

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

    return (Component)selectedCarsPage;
}

// WebScript DodgeLite Main.wos
- displayCars
{
    id selectedCarsPage = [[self application]
        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 the application 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 135).

State Storage Strategies

WebObjects gives you the option of storing state in various ways:

- **In the server.** State is maintained in memory within a WebObjects application.
- **In the page.** State is embedded in the HTML page that’s returned to the user.

- **In cookies.** State is embedded in name-value pairs (“cookies”) in the HTTP header and passed between the client and server. Like “state-in-the-page,” cookies store state on the client.
- **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 the next sections, “Comparison of Storage Options” and “A Closer Look at Storage Strategies.” If you decide to use one of the other state-storage solutions, 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 131).

You may find you need to control the amount of state that is stored. The sections “Controlling Session State” (page 133) and “Controlling Component State” (page 135) tell you how to do so.

Comparison of Storage Options

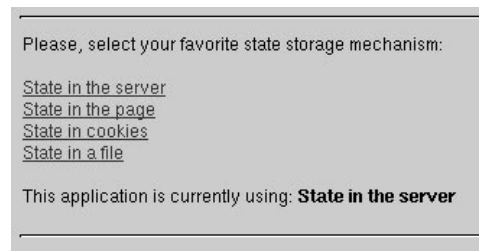
The following table summarizes the pros and cons of each of the state storage options. These options are discussed in more detail in the next section, “A Closer Look at Storage Strategies,” but seeing an overall comparison might save you time in deciding which options to explore.

	State in Server	State in Page	State in Cookies	Custom Storage
Simplicity	Simplest approach; WebObject’s default.	Relatively simple, but can involve design changes to application.	Relatively simple.	More complex.
Security	Secure since state is on server and accessed by encrypted session IDs.	Since data is passed to client, opens possibility that data could be modified by user.	Since data is passed to client, opens possibility that data could be modified by user.	If stored on server, can be as secure or more secure than state-in-server.
Scalability	Can consume lots of memory. Also, can’t use load balancing once state is established in a particular application instance.	More scalable since any application instance can handle a request (because state is bundled with each request). Applications don’t grow when new sessions are added.	Not very scalable. Cookie specification limits capacity to 4K bytes per cookie, but some browsers have further limitations.	Depends on design of storage. If file system or database used for storage, can scale to accommodate almost any need.
Reliability	Least reliable, since if the server crashes, state is lost.	More reliable, since a server crash doesn’t affect state stored on client.	More reliable, since a server crash doesn’t affect state stored on client.	Can be extremely reliable if state is stored in server file system or database.

	State in Server	State in Page	State in Cookies	Custom Storage
Other		Performance can suffer if lots of data is passed back and forth between client and server. State can become out of sync, especially when using frames.	Client can refuse to accept cookies.	

A Closer Look at Storage Strategies

To compare and further understand state-storage options, look at the SessionStores sample application. This application presents the user with a choice of storage strategies:



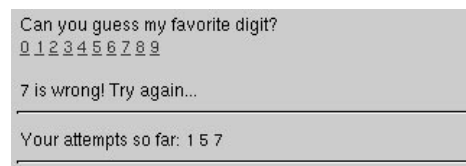
Please, select your favorite state storage mechanism:

- [State in the server](#)
- [State in the page](#)
- [State in cookies](#)
- [State in a file](#)

This application is currently using: **State in the server**

Figure 33. SessionStores: Storage Choices

Once a storage strategy has been chosen, the application plays a guessing game with the user:



Can you guess my favorite digit?

0 1 2 3 4 5 6 7 8 9

7 is wrong! Try again...

Your attempts so far: 1 5 7

Figure 34. SessionStores: Guessing Game

As you can see, the application keeps track of a user's previous guesses within a session—these guesses are part of the state that must be stored from cycle to cycle.

The SessionStores example was designed to illustrate WebObjects' support for various state storage strategies, so it lets you switch between strategies while the application is running. This is not a design you should emulate in your

applications—changing storage strategies midsession can cause errors. For example, imagine an application that stores state in the page during the first half of a session and stores state in cookies for the second. Now, suppose that the user backtracks from a page in the second half to one in the first and resubmits the page. The application’s strategy and the actual storage mechanism won’t match, and state will be lost.

In a normal WebObjects application, you should set the session storage mechanism as early as possible, usually in the application object’s initialization method. You set the mechanism by sending the application object a **setSessionStore:** message. This method takes a `WOSessionStore` (or `SessionStore` in Java) object as an argument. `WOSessionStore` declares these methods to create specific types of session stores:

- `serverSessionStore`
- `pageSessionStore`
- `cookieSessionStoreWithDistributionDomain:secure:`

The following sections describe each state-storage option in detail and show examples of setting the session store.

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
SessionStore sessionStore = SessionStore.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” (page 133))
- Setting the size of the page cache (see “Adjusting the Page Cache Size” (page 136))

- Page uniquing by implementing `pageWithName:` in the session object (see “`pageWithName:` and Page Caching” (page 138))

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.

State in the Page

To store state in the page, you must do two things:

- Specify page session store as the application's state storage mechanism.
- Include a `WOStateStorage` element inside of a form on each page.

You specify the page session store this way:

```
// Application.java
public Application() {
    super();
    this.setSessionStore(SessionStore.pageSessionStore());
}

// Application.wos.
- init {
    [super init];
    [self setSessionStore:[WOSessionStore pageSessionStore]];
    return self;
}
```

Next, you must add a form to each page of the application and place a `WOStateStorage` object within the form. You do this in WebObjects Builder by adding a Custom WebObject to the form and then using the Inspector panel to specify that the type of element is “`WOStateStorage`”.)

The `WOStateStorage` element maps to an HTML input element of type “hidden.” The “hidden” input type contains text that is not displayed in the user's browser. For example, using state in the page, the HTML source for the Guess page of the SessionStores example would look something like this:

```
<FORM METHOD=Post ACTION=someAction>
  Can you guess my favorite digit?<BR>
  <SELECT NAME="guesses">
    <OPTION>1
    <OPTION>2
    ...
    <OPTION>9
  </SELECT>
  <INPUT TYPE="hidden" NAME="hiddenState" VALUE="previousGuesses">
  <INPUT TYPE="submit" VALUE="Guess">
</FORM>
```

When WebObjects generates a response page containing a `WOSessionStore` element, it packages the session state by archiving the session object—and consequently, all the component objects that it contains—using classes and methods defined in the Foundation framework. The session and components are archived into an `NSData` object. (In Java, `NSData` is called `next.util.ImmutableBytes`.) The `NSData` object is then asked for its ASCII representation, which is written into the HTML page as hidden fields. (See the class specification for `NSArchiver` in the *Foundation Framework Reference* for more information on archiving.)

WebObjects writes as many hidden fields as are necessary to contain the state data. `WOStateStorage`'s `size` attribute specifies the maximum size of each of these hidden fields (500 bytes in the example above). The `size` attribute is provided because browsers differ in the amount of text that they allow within a single hidden field. Most browsers have no problem with the default value for `size` (1000 bytes).

When the user submits the HTML page to the server, the process is reversed. The application's page session store restores the session state by recombining the ASCII data it finds in the hidden fields into the original ASCII archive, converting the ASCII archive to its binary, `NSData`, representation, and then unarchiving the session object and its contents from the `NSData` object.

There are some limitations inherent in storing state in the page:

- **Forms are required.** Because state is stored in an input element—which according to the HTML specification must exist within a form element—you must structure your application around forms. If you want session state to be available at any point in the application, each page of the application must have a form, and that form must contain a `WOStateStorage` element.

If a page has multiple forms, you must include the page state data in each form. If a form lacking this data is submitted, the application will no longer have the state information it needs.

- **Backtracking.** Because each page carries a record of the state existing at the time of its creation, backtracking can make the page state and the actual state disagree. If, for example, the user make five guesses in the SessionStores example, backtracks two pages, and submits another guess, the application will claim that four guesses were made, when the actual number is six.
- **Frames.** Storing state in the page is a problem if the “pages” in question are frames. Your state can quickly get out of sync. For example, suppose you have a mail application with two frames. One of the frames shows a list of messages with one message selected, and the other frame shows the text of the selected message. If you delete the message in the top frame, the state of the bottom frame isn’t updated (unless you implement your own solution).
- **Archiving.** Because WOSTorage works by archiving the objects to be stored, only objects that can be archived using the Foundation framework’s archiving mechanism can be stored. That is, the objects must conform to the NSCoder protocol (or the `util.Coding` interface in Java). For scripted objects, you don’t need to worry about this. WebScript provides a default archiving implementation that will archive data stored in the object’s instance variables. For compiled objects (whether Java or Objective-C), on the other hand, you have to implement the archiving methods yourself, as described in “Storing State for Custom Objects” (page 131).

State in Cookies

A “cookie” is another way that a web application can store state information on the client machine. Instead of being part of the HTML page as with the state-in-the-page mechanism, a cookie is passed as part of the HTTP header information. Here is the syntax for the cookie header line:

```
Set-Cookie: NAME=VALUE; expires=DATE; domain=DOMAIN_NAME;  
path=PATH; secure
```

The *NAME=VALUE* association is the only required field. It holds the cookie’s data and the name by which it can be accessed. The other fields are optional and set limitations on when the data will be passed from the client back to the server, as shown in the following table:

Field	Description
expires	The date after which the cookie is no longer valid. Once a cookie expires, the client will no longer return it to the server, and the client is free to delete it.

Field	Description
domain	The Internet domain name for which the cookie is valid. If, for example, the specified domain is apple.com for a given cookie, that cookie is returned along with a request to any host whose domain ends in apple.com (for example, www.apple.com)—assuming the URL is within the directories specified by path .
path	The directories within a given domain for which this cookie is valid. If, for example, a cookie has a domain of www.apple.com and a path of /devDoc , the client returns the cookie to the server for any request that begins with http://www.apple.com/devDoc...
secure	Specifies that the cookie can be passed only using a secure communications channel, such as SHTTP.

See http://www.netscape.com/newsref/std/cookie_spec.html for a complete description of cookies.

To use cookies, all you need to do is set the application's session storage type in the application object's initialization method:

```
//Java Application.java
public Application() {
    super();
    setSessionStore(
        SessionStore.cookieSessionStoreWithDistributionDomain("",
            false));
}

//WebScript Application.wos
- init {
    [super init];
    [self setSessionStore:
        [WOSessionStore cookieSessionStoreWithDomain:@"",
            secure:NO]];
    return self;
}
```

In this example, we set the domain to the empty string so that cookies that this application sends to the user are valid for all domains. We also turn off the requirement for a secure communications channel. Note that the cookie store API doesn't allow for a path argument. WebObjects automatically restricts the path so that cookies that an application produces are valid only within the application directory. For example, if you set the SessionStores application to use a cookie session store, the client returns a cookie only if the request URLs have this prefix:

```
/cgi-bin/WebObjects/Examples/SessionStores.woa/
```

As with storing state in the page, the cookie session-storage mechanism uses archive objects that should be stored. WebObjects packages the session state by archiving the session object (and all the component objects that it contains) into an NSData (next.util.ImmutableBytes) object. The

NSData object is then asked for its ASCII representation. WebObjects pairs this data with names it generates and creates the Set-Cookie headers of the response page.

The process is reversed when a user submits a request containing cookies. The ASCII archive from the Set-Cookie headers is converted to its binary, NSData, representation. The session object and the components it contains are then unarchived from the NSData object, thus restoring the session state.

One of the big advantages of using cookies over state in the page is that you don't have to design your application around forms. As you recall, storing state in the page implies using hidden field elements, which must be located in HTML forms. Cookies, however, are stored in the HTTP header and so are independent of the HTML elements in the page. With a cookie session store you could, for example, let users navigate from page to page by using hyperlinks rather than by submitting forms. In addition (and for similar reasons), storing state in cookies works better with frames than does storing state in the page.

However, the cookie mechanism has a size restriction that limits its usefulness. Currently, cookie data is passed from the HTTP server to the WebObjects application either through environment variables that typically are limited to 4KB or through a server's own API that in some cases is even more restrictive. We recommend that cookie state data (that is the ASCII representation of the state data) be kept to 2KB or less. Given these limitations, cookies can be best used for such things as storing keys used to fetch information from a database.

Custom State-Storage Options

If the provided state-storage strategies are insufficient for your needs, you can implement your own state storage. For example, you might want to store state in a file or database. The SessionStores application provides an example of a state-storage mechanism that uses the file system. Let's take a look at how it's done.

In WebObjects, an application saves and restores sessions by sending the session store object these messages:

- `saveSession:`
- `restoreSession`

This is the minimum interface that a custom session store must present to the application object. In the WebScript version of the SessionStores example, the custom storage class `FileSessionStore` presents this interface:

```
// WebScript StateStorage FileSessionStore.wos
@interface FileSessionStore:NSObject {
    id archiveDirectory;
}
- init;
- archiveFileForSessionID:aSessionID;
- archiveForSessionID:aSessionID;
- restoreSession;
- saveSession:aSession;
@end
```

These methods have the following implementation:

```
@implementation FileSessionStore

- init {
    self = [super init];
    archiveDirectory = [WOApp pathForResourceNamed:@"SessionArchives"
        ofType:nil];
    return self;
}

- archiveFileForSessionID:aSessionID {
    return [NSString stringWithFormat:@"%s/%s", archiveDirectory,
        aSessionID];
}

- archiveForSessionID:aSessionID {
    id archiveFile = [self archiveFileForSessionID:aSessionID];
    return [NSData dataWithContentsOfFile:archiveFile];
}

- restoreSession {

    id request = [[WOApp context] request];
    id archivedSession;
    id restoredSession;

    // Allow requests in this session to go to any application instance.
    [[WOApp context] setDistributionEnabled:YES];

    // Get archived session (as an NSData object)
    archivedSession = [self archiveForSessionID:[request sessionID]];
```

```

        // Unarchive session
        restoredSession = [NSUnarchiver
            unarchiveObjectWithData:archivedSession];

        return restoredSession;
    }

    - saveSession:aSession {
        id request = [[WOApp context] request];

        // Store data corresponding to session only if necessary.
        if (![aSession isTerminating] && ![request isFromClientComponent]) {
            id sessionData = [NSArchiver
                archivedDataWithRootObject:aSession];
            id sessionFilePath = [self archiveFileForSessionID:[aSession
                sessionID]];

            [sessionData writeToFile:sessionFilePath atomically:YES];
        }
    }

@end

```

As you can see, when the `FileSessionStore` receives a **saveSession:** message, it checks whether the session object needs to be archived, and if so, it asks `NSArchiver` to create a binary archive of the session object and all of the components it contains. It then invokes its own **archiveFileForSessionID:** to determine the path for the archive file. Finally, it writes the data to the file. Notice that the session data is written to a file whose name is the session ID itself.

In this implementation, **restoreSession** restores the state for a particular session. An interesting point in the **restoreSession** method implementation is the **setDistributionEnabled:** message to the `WOContext` object (Context in Java). This method enables application load balancing. As you learned in the earlier section “State in the Server” (page 123), when state is stored in the server, all requests from a particular session must access the same application instance on the same machine. In this example, because session state is stored in the file system and not in the application’s memory, any application instance can handle any request. The **setDistributionEnabled:** method enables application load balancing by allowing any application instance to respond to any request.

Note that the Foundation classes `NSArchiver` and `NSUnarchiver` aren’t provided in the Java `next.util` package and that the `WebSession` and `Component` classes don’t implement the `Serializable` or the `Externalizable` interface. For these reasons, you can’t implement storing state in the file system in Java. If you want to store Java objects in the file system, you can do so if they implement the `Coding` interface, but you must write your equivalent of the `FileSessionStore` class in WebScript or Objective-C.

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, when state is stored in the page, in cookies, or in the file system, objects are asked 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 (`EditingContext` in Java) 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 (`EditingContext` in Java) 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 OpenStep 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;
}
```

Even though the Java packages provide a different archiving mechanism, your Java classes should use the Foundation archiving mechanism. In Java, the `NSCoding` protocol is called the `Coding` interface, and it declares only one method, **`encodeWithCoder`**. If your class conforms to the `Coding` interface, it should also implement a constructor that takes a `Coder` object as an argument. (This is the equivalent of the **`initWithCoder:`** method.)

```
// Java example
public void encodeWithCoder(Coder aCoder) {
    EditingContext.encodeObjectWithCoder(this, aCoder);
}

public MyClass(Coder aDecoder) {
    EditingContext.initObjectWithCoder(this, aDecoder);
}
```

The enterprise object simply passes on responsibility for archiving and unarchiving itself to the `EOEditingContext` class, by invoking the **`encodeObjectWithCoder:`** and **`initObjectWithCoder:`** class methods and passing a reference to itself (**`self`**) as one of the arguments. The editing context takes care of the rest. (See the `EOEditingContext` class specification in the *Enterprise Objects Class 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.

You can see implementations of `encodeWithCoder:` and `initWithCoder:` in the DodgeDemo application, in the class `ShoppingCart`.

```
- 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 DodgeDemo's `ShoppingCart` implements these methods instead:

```
public void encodeWithCoder(Coder coder) {
    coder.encodeObject(leaseTerm);
    coder.encodeObject(downPayment);
    // DodgeDemoJava defines a custom Car object that contains all
    // info about the car.
    coder.encodeObject(car);
}

public ShoppingCart(Coder 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:

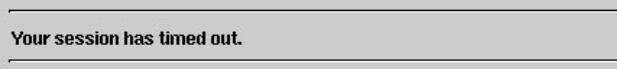


Figure 35. A Session Time-Out Error Message

By default, a session object's time-out value is so large that sessions effectively never expire. You should set the session time-out for your application to the shortest period that seems reasonable. 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 reinitialize the session state) and a **sleep** message at the end (where you can release it).

Controlling Component State

Component objects exist within a particular session and are stored along with the session object between each cycle of the 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" (page 139) looks at the consequences of page caching on the client.

There are three common techniques for controlling component resources:

- Adjusting the page cache size
- Using **awake** and **sleep** to initialize and release resources
- Controlling page instantiation by implementing **pageWithName:**

Adjusting the Page Cache Size

As noted in “WebObjects Viewed Through Its Classes” (page 63), except for the first request, a 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 like this:



Figure 36. Backtracking Error Message

To keep users from encountering this error, your application should maintain a moderate sized cache of pages. (Another strategy is to limit the number of identical page instances that your application creates; see “pageWithName: and Page Caching” (page 138) for one way to do this.) 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.

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 for DodgeLite's Main component could be changed to:

```
// rewritten DodgeLite 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];
    [types release];
    [prices release];
}
```

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.

pageWithName: and Page Caching

When the application object receives a **pageWithName:** message, it creates a new component. For example, in the HelloWorld example a user enters a name in the first page (the Main component), clicks Submit, and is presented with a personal greeting on the second page (the Hello component). Clicking the Submit button in the first page invokes the **sayHello** method in the Main component. As part of its implementation **sayHello** sends a **pageWithName:** message to the application object:

```
// Java HelloWorld
String visitorName;

public Component sayHello() {
    //Create the next page
    Hello nextPage = (Hello)application().pageWithName("Hello");

    // Set state in the Hello page
    nextPage.setVisitorName(visitorName);

    // Return the Hello page
    return nextPage;
}
```

Each time the **sayHello** method is invoked, a new Hello component is created. For example, if the user backtracks to the main page and clicks the Submit button again, another Hello page is created. It's unlikely this duplication of components will be a problem for the HelloWorld application, since users quickly tire of its charms. But, depending on design, some applications may benefit by modifying the operation of **pageWithName:** so that an existing component can be reused.

If you want to extend WebObjects' page caching mechanism to include pages returned by **pageWithName:**, you must implement your own solution. Fortunately, it's easy. One approach is to have the session maintain a dictionary that maps page names to page objects. Here's the code you would add to the session object:

```
// example Session.java
MutableHashtable pageDictionary;

public Session() {
    super();
    pageDictionary = new MutableHashtable();
}

public Component pageWithName(String aName) {
    return (Component)pageDictionary.get(aName);
}

public void storePage(String aName, Component aPage) {
    pageDictionary.put(aName, aPage)
}
```

```
// example Application.java
public Component pageWithName(String aName) {
    Component aPage;

    if (aName == null)
        aName = "Main";
    aPage = ((Session)session()).pageWithName(aName);
    if (aPage == null) {
        aPage = super.pageWithName(aName);
        ((Session)session()).storePage(aName, aPage);
    }
    return aPage;
}
```

Note that we store pages in the session object because we want to cache these pages on a per-session basis. (Implementing the dictionary in the application object would cache pages on a per-application basis.) Our override of `WOApplication`'s **pageWithName**: first attempts to retrieve the page from the current session's dictionary before creating a new copy of the page.

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.

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 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.

In earlier chapters, you learned about client-side Java components. Client-side components are Java applets that your application can use instead of server-side dynamic elements to interact with users. WebObjects comes with several premade client-side components. To use them you simply add them to your application in much the same way that you add a server-side dynamic element.

You can create your own client-side component if the premade components don't suit your needs or if you already have a Java applet that you would like to use as a client-side component. This chapter describes how to do so.

Choosing a Strategy

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 `next.wo.client.Association`. As explained in "How Client-Side Components Work" (page 38), client-side components use an Association object 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). The benefit of using SimpleAssociation is that it uses an AppletGroupController object. AppletGroupController is a hidden Java applet (on the client) that controls the visible applets and handles communication back to the server (see Figure 37). 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 Association fires its applet's action, 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.

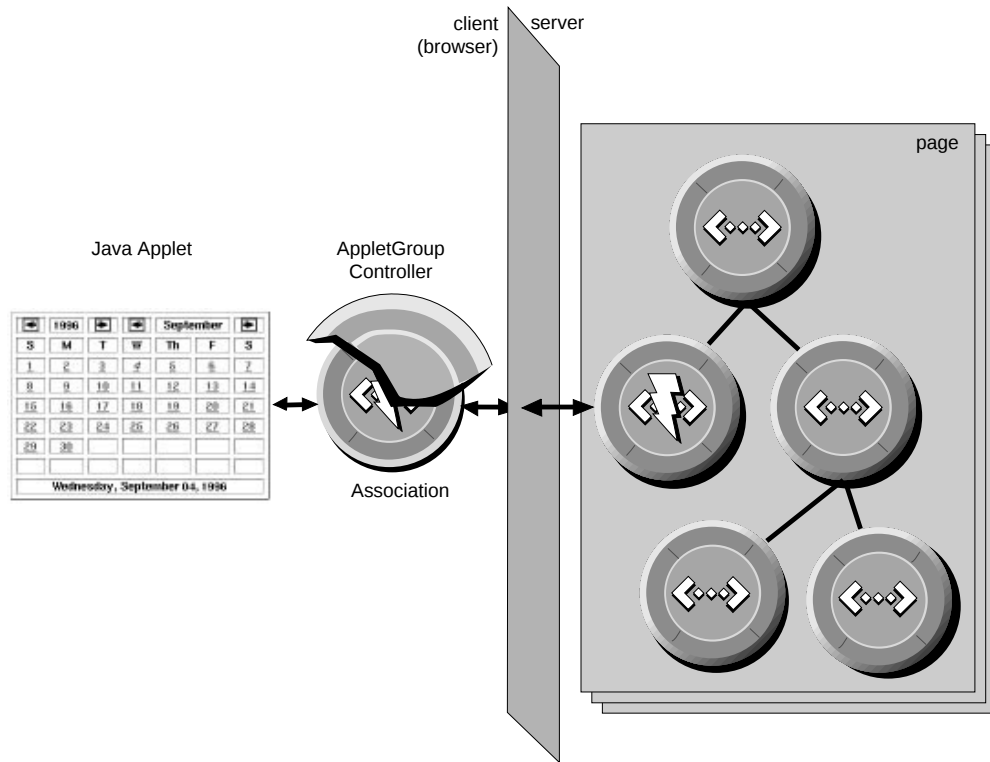


Figure 37. The Principal Objects Involved in Client-Side Components

SimpleAssociation objects don't get or set values themselves; instead, they rely on the actual client-side components 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.

When you build your project, Project Builder builds both the individual Java **.class** files and a **.jar** file containing the entire ClientSideJava subproject. This way, you have the option of using WOApplet's **archive** binding for browsers that support **.jar** files.

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 144) 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.

```
// fictitious 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.

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. For example, if you included any `logWithFormat` (or `logString`) statements in your code for debugging purposes, you'll probably want to remove them before you deploy.

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 (`StatisticsStore` in Java) 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.

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/MyWebApp.woa/-/WOSTats
```

Note: You can access any component directly using a URL with this form.

Figure 38 shows a WOSTats page.



Figure 38. WOSTats Page

For more information about the statistics presented on the WOSTats page, see *Serving WebObjects*.

If you want access to statistics programmatically, send the WOStatisticsStore a **statistics** message. For example:

```
// WebScript
NSDictionary *myDict = [[[self application] statisticsStore]
    statistics];

// Java
ImmutableHashTable myDict =
    this.application().statisticsStore().statistics;
```

For a list of keys to this dictionary, see the `WOStatisticsStore` class specification in the *WebObjects Class Reference*.

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.

To record extra information about a page, override **`descriptionForResponse:inContext:`** in your component.

For example, the `HelloWorld` example's `Hello` component could return the value of its `visitorName` instance variable along with the component name:

```
// WebScript HelloWorld Hello.m
- (NSString *)descriptionForResponse:(WOResponse *)response
inContext:(WOContext *)context {
    return [NSString stringWithFormat:@"%s/%s",
        [self name], visitorName];
}

//Java HelloWorld Hello.java
public String descriptionForResponse(Response response, Context
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 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.

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 (`WebApplication` in Java) provides the following methods that you can override to show your own error page.

Method	Invoked When
<code>handleSessionCreationError</code>	The application needs to create a new session but can't.
<code>handleSessionRestorationError</code>	The application receives a request from a session that has timed out.
<code>handlePageRestorationError</code>	The application tries to access an existing page but cannot. Usually, this occurs when the user has backtracked beyond the limit set by <code>setPageCacheSize</code> : and <code>setPageRefreshOnBacktrackEnabled</code> : is NO.
<code>handleException:</code>	The application receives an exception; that is, any general type of error has occurred.

For example, the following implementation of `handleException:` returns a component named `ErrorPage` whenever an error occurs in the application.

```
public Response handleException(java.lang.Throwable anException) {
    Response response = new Response();
    Request request = context().request();
    String newURL = "http://" + request.applicationHost() +
        request.adaptorPrefix() + "/" + request.applicationName() +
        ".woa/-/ErrorPage.wo";

    response.setHeader(newURL, "location");
    response.setHeader("text/html", "content-type");
    response.setHeader("0", "content-length");
    response.setStatus(302);

    return response;
}
```

Notice that this method, and all of the error-handling methods, return a `WOResponse` object instead of a `WOComponent` object. It creates the response by directly setting the URL in the HTTP header to point to the component that it wants to return (in this case, the component is named `ErrorPage`). Notice how you can retrieve much of the information about the application URL through the current request object, which you can access from the current context 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.** If no users are accessing the application, you might want to shut it down until a user requests it. To do so, use `WOApplication`'s `setTimeout:` method. This method shuts down the application after it has been idle for a given number of seconds.

```
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 24 hours.

```
public Application() {
    super();
    this.terminateAfterTimeInterval(24*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 24 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
    {
        [[[self statisticsStore] statistics]
         objectForKey:@"StartedAt"];
    }
    // Compare start date to current date. If the difference is
    // greater than 24 hours, refuse any new sessions.
    if ((([NSDate date] timeIntervalSinceReferenceDate) -
         [startDate timeIntervalSinceReferenceDate]) > 86400)
    {
        [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 online 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 63), 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 once per new instance of that component. 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 include the `-c` option on the command line to perform component-definition caching.

```
WODefaultApp -c
```

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. OpenStep provides some tools that can help you check your code for memory leaks. For more information, see the Debugging section of Project Builder's online help.

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 155).

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 109). 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 155).

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 the chapter “Managing State” (page 109).

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*.

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:

- **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 is 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 Class Reference*.

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 `$(NEXT_ROOT)/NextLibrary/WOApps`.

If you're installing a framework, type `$(NEXT_ROOT)/NextLibrary/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/NextLibrary/WOApps/MyApp.woa
MyApp[.exe]
Resources/
WebServerResources/

<DocRoot>/WebObjects/MyApp.woa
WebServerResources/
```

When the WebObjects adaptor first receives an HTTP request, the adaptor looks for an executable in `<DocRoot>/WebObjects` and

NeXT_ROOT/NextLibrary/WOApps. Thus, you can install the entire directory under *<DocRoot>/WebObjects*, but doing so presents a security problem if you have scripted components. Any client can access any file under the document root, which means that if you install scripted components under the document root, you are exposing source code to outside users.

Instead, it is recommended that you install most of the application in *NeXT_ROOT/NextLibrary/WOApps* and install only the web server resources under the document root. It is also recommended that you install the application directly in the *<DocRoot>/WebObjects* directory rather than in a subdirectory. If you install in a subdirectory, your application can still run but cannot find image files unless you provide the application path on the command line. For more information, see *Serving WebObjects*.



WEBSCRIPT

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 183), 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. (This is actually the script file for the Visitors example’s Main component. You can look at the entire Visitors example in `<DocRoot>/WebObjects/Examples/WebScript/Visitors.woa`.)

```

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. 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 three kinds of script files: a component script inside a **.wo** directory, an application script, and a session script. These three kinds of scripts create subclasses of the WebObjects classes **WOComponent**, **WOApplication**, and **WOSession**, 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 *myString1, *myString2;
```

For more information on specifying class names in variable declarations, see the section “Data Types” (page 174).

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 171).

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 41), a `WOApplication` is created when you started a WebObjects application, a `WOSession` is created each time a different user accesses that application, and a `WOComponent` is created the first time a user accesses that page in the application. 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. Finally, the variables you declare at the top of a component script are created and released as that component is created and released.

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 63).

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 173) 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 187).

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, this is the definition of a method from the Main component in the Visitors 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 the following:

- Type information is optional. When there is no type, **id** is assumed.
- Explicitly specifying the **id** type is not allowed:

```
// NO!! This won't work.
- (id)aMethod:(id)anArg { ... }
```

- A return type of **void** is not allowed:

```
// This won't work either.  
- (void) aMethod:(NSString *)anArg { ... }
```

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 (and, as stated above, even if a method returns no value you shouldn't declare it as returning **void**).

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 **NSDate** 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:index] 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, the **Application.wos** script in the Visitors example declares 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 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 178).

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 166) 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 168). 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 a mutable object 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 `NSDate` objects, which are always immutable:

```

// Using the creation method date, create an NSDate instance
// 'now' that contains the current date and time
now = [NSDate date];

// Return a string representation of 'now' using a format string
dateString = [now descriptionWithCalendarFormat:@"%B %d, %Y"];

// Using the creation method dateWithString:, create an NSDate
// instance 'newDate' from 'dateString'
newDate = [NSDate 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 187).

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;
```

WebScript handles logical operators slightly differently from C. In C, Boolean expressions *short-circuit*. For example, in this statement:

```
(!a || (a && !i))
```

In C, an 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.

In WebScript, Boolean expressions never short-circuit. The entire expression is always evaluated.

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 unique 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, or the WOComponent 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, or WOComponent 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** word 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 are 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`, and `WOApplication` 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`, or `WOComponent`; 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. 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];
    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]];
}
```

The category must be included either at the end of a component's script file (that is, a script file within a `.wo`) or it must be included in a scripted class's stand-alone script file. Do not place categories in the application or session script.

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 <code>.h</code> and a <code>.m</code> file	Stands alone (unless inside of a component directory)
Supports all C language features	Has limited support for C language features; for example, doesn't support structures, pointers, enumerations, or unions

Objective-C	WebScript
Methods not declared to return void must include a return statement	Methods aren't required to include a return statement
Has preprocessor support	Has no preprocessor support—that is, doesn't support the <code>#define</code> , <code>#import</code> , or <code>#include</code> 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 `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 `NSNumber`s). 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 or session script, you can use `WOApplication` methods to access the object:

```
[[WOApplication application] applicationScriptMethod];
[[WOApplication application] session] sessionScriptMethod];
```

To access a component's methods, you must store the component in the session and then access it through the session. For example, suppose you wanted to rewrite the SelectedCars component of the DodgeDemo so that its database fetch code was in a compiled object and that object directly set the value of the **message** variable in the SelectedCars component. You'd add the following statement to the **init** method **SelectedCars.wos**:

```
/* Store the component in the session. */
[self.session setObject:self forKey:@"SelectedCars"];
```

and then you can access it in you custom object's **fetchSelectedCars** method this way:

```
/* Get the component from the session. */
WOComponent *selectedCarsPage = [[[WOApplication application]
    session] objectForKey:@"SelectedCars"];

/* Send it a message. */
[selectedCarsPage setMessage:@"You must supply a name and address"];
```

To avoid compiler warnings, you should declare the scripted method you want to invoke in your code. This is because scripted objects don't declare methods—their methods are parsed from the script at runtime. If you don't declare their methods in your code, the compiler issues a warning that the methods aren't part of the receiver's interface.

Note: This step isn't strictly required—your code will still build, you'll just get warnings.

For the example above, you'd add the following declaration to your object's implementation (.m) file:

```
@interface WOComponent (SelectedCarsComponent)
- (void)setMessage:(NSString *)aMessage;
@end
```

While it's certainly straightforward to access a scripted object's methods from Objective-C code, you may not want to have that degree of interdependence between your scripts and your compiled code. You may want to minimize the interdependence to facilitate reusability.

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 183) 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 **classNameWithContentsOfFile:** 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 filename. If the argument 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.

Note: The configuration of your HTTP server determines the user who owns autostarted applications.

Reading From Files

The string, array, and dictionary classes provide methods of the form *classNameWithContentsOfFile:*. 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 172).

+ 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:** (page 195).

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:** (page 200).

– 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 that has 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.

– **floatValue**

Returns the floating-point value of the receiver’s text as a float, skipping white space at the beginning of the string.

– **intValue**

Returns the integer value of the string’s text, assuming a decimal representation and skipping white space at the beginning of the string.

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 the **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 NSMutableArray are not available for NSArray. Although their use is limited to managing static collections of objects, it is best to use NSArray wherever possible.

You can create NSArray with WebScript's @ syntax for defining constant objects. For example, the following statements create NSArray:

```
id availableQuantities = @(1, 6, 12, 48);
id shortWeekDays = @("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat");
```

You can also create NSArray 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` or `NSMutableArray`. For more information on class methods, see “Sending a Message to a Class” (page 172).

+ array

Returns an empty array. Usually used to create `NSMutableArray`s. `NSArray`s 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`. 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](#) (page 200) .

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. To determine equality, each element of the array is sent an **isEqual:** message.

– **indexOfObjectIdenticalTo:**

Returns the index of the first occurrence of a specified object. To determine equality, the **ids** 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.

– **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.

– **insertObjectAtIndex:**

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.

– **removeObjectIdenticalTo:**

Removes all occurrences of a specified object and moves elements up as necessary to fill any gaps.

– **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.

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.

– **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:** (page 193). 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 `NSMutableDictionary`s are not available for `NSDictionary`s. Although their use is limited to managing static collections of objects, it's best to use `NSDictionary`s wherever possible.

You can create `NSDictionary`s using WebScript's `@` syntax for defining constant objects. For example, the following statements create `NSDictionary`s:

```
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 172).

+ `dictionary`

Returns an empty dictionary. Usually used to create `NSMutableDictionary`s. `NSDictionary`s 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` (page 205) .

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 sortedByValueUsingSelector:@"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.

– **removeObjectsForKeys:**

Removes the entries for each key in a specified array.

– **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

`NSDate` objects represent dates and times. These objects are especially suited for representing and manipulating dates according to Western calendrical systems. `NSDate` performs date computations based on Western calendrical systems, primarily the Gregorian.

The methods provided by `NSDate` are described in more detail in “Commonly Used Date Methods” (page 207).

The Calendar Format

Each `NSDate` 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()`. `NSDate` interprets dates that are represented as strings conforming to this format. You can set the default format for an `NSDate` 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)
%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

Conversion Specifier	Argument Type
%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 `NSDate`, 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, `NSDate`. For more information on class methods, see “Sending a Message to a Class” (page 172).

+ `calendarDate`

Returns an `NSDate` initialized to the current date and time.

+ `dateWithString:calendarFormat:`

Returns an `NSDate` initialized to the date in a provided string, and sets the new `NSDate`’s calendar format to the specified format. The date string must match the provided format exactly. See “Date Conversion Specifiers” (page 206) for more detailed information on formats used by `NSDate`.

Adjusting a Date

– `dateByAddingYears:months:days:hours:minutes:seconds:`

Returns an `NSDate` 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 `NSDate` formatted according to the `NSDate`'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" (page 206) for more detailed information on formats used by `NSDate`.
- **setCalendarFormat:**
Set the receiver's default calendar format to the provided string.

Retrieving Date Elements

- **dayOfWeek**
Returns a number that indicates the `NSDate`'s day of the week (0–6).
- **dayOfMonth**
Returns the `NSDate`'s day of the month (1–31).
- **dayOfYear**
Returns a number that indicates the `NSDate`'s day of the year (1–366).
- **dayOfCommonEra**
Returns the `NSDate`'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 `NSDate`'s month of the year (1–12).
- **yearOfCommonEra**
Returns the `NSDate`'s year value (including the century).
- **hourOfDay**
Returns the `NSDate`'s hour value (0–23).

– **minuteOfHour**

Returns the NSDate's minutes value (0–59).

– **secondOfMinute**

Returns the NSDate's seconds value (0–59).

Index

%@ 59
 .api file 20
 .html file 20, 21
 .wo directory, *See* wo directory
 .woa directory, *See* woa directory
 .wod file, *See* wod file
 .wos file, *See* code files
 @end 182
 @implementation 182
 @interface 181

A

accessing existing session 75–76
 accessing variables 171–172
 accessor methods 35, 85, 103
 automatic 171–172
 actions 43–45
 invoking 79–80, 171–174
 return value 45
 adaptor 26–27
 object 65–66
 See also WOAdaptor
 addEntriesFromDictionary: method 204
 addObject: method 198
 allKeys method 203
 allKeysForObject: method 203
 allValues: method 203
 .api file 20
 appendFormat: method 195
 appendToResponse:inContext:
 method 53–54, 80–82
 declaration 70
 example 53–54
 AppletGroupController 143
 applets, *See* client-side components
 application 19, 25–26
 code file, *See* application code file
 communication with server 65–66
 compiled 27
 creating 19
 debugging 57–62
 executable 26, 27–28
 initialization 47–48
 install 159–160
 object, *See* WOApplication

 parts 19–25
 running 25–26
 shutdown 155–156
 starting 25–26
 state 113–115
 variables 23, 113–115
 application code file 19, 23
 Application.java 23
 Application.m 23
 Application.wos 19, 23
 architecture 65–87
 archiving 131–133
 arithmetic operators 176
 array method 197
 arrays 196–200
 See also NSArray
 constant 169–170
 arrayWithArray: method 197
 arrayWithContentsOfFile:
 method 190–191, 197
 arrayWithObject: method 197
 arrayWithObjects: method 197
 assignment statements in
 WebScript 168–170
 Association 85–86
 subclassing 146–147
 associations 84–85
 automatic deallocation 72, 82
 autorelease pool 72
 awake method 46–50
 for application 74
 for component 77, 137
 for session 75, 76, 135

B

backtracking 135–140
 binding to reusable component 95
 bindings
 rules 35–36
 browser
 caching pages 139–140
 communicating with WebObjects
 application 26

C

caching pages 77–78, 81, 135–140
 adjusting size 136–137
 client-side 139–140
 database applications 140
 disabling 136
 calendarDate method 207
 calendarFormat method 208
 caseInsensitiveCompare: method 194
 categories (WebScript) 182–183
 CGI adaptor 26
 class object 172–173
 class, definition 165
 classes, scripted 181–182
 CLFF log file 151
 client-side components 37–39
 conceptual overview 85–86
 creating 143–147
 applets without source
 code 146–147
 state storage 86
 synchronization 85–86
 code files 20, 21–22
 communication between components
 parent and child 98–102
 peer 119
 compare: method 194
 Component class (Java), *See*
 WOComponent
 component definition 83
 component reference 86
 components 19, 20–22, 82–87
 See also page
 accessor methods 85, 171–172
 caching, *See* caching pages
 client-side, *See* client-side components
 creating 76–77
 initialization 49–50
 object, *See* WOComponent
 parts 20–22
 request, *See* request page
 response 44
 reusable, *See* reusable components
 sharing 104
 state 118–120, 135–140

- synchronization 85, 102–104
 - client-side 85–86
- template 82–83
- URL, specifying 53
- variables 118–120
 - in reusable components 97
- `componentsJoinedByString:` method 200
- `componentsSeparatedByString:` method 193
- conceptual overview of WebObjects 65–87
- constants, WebScript 169–170
- constructors 46–50
- context ID 68, 77
- context, accessing 51
- cookies 126–128
- count method 197, 203
- `createSession` method 75
- creating new component 76–77
- creating new session 74–75
- creating objects 173–174
- current component 84
- custom objects, storing 131–133
- cycle, request-response *See* request-response loop

D

- data types 174–175
 - id 167
- database integration 71–72
 - and backtracking 140
 - performance 158
 - storing state 131–132
- `dateByAddingYears:months:days:hours:minutes:seconds:` method 207
- dates 206–209
- `dateWithString:calendarFormat:` method 207
- `dayOfCommonEra` method 208
- `dayOfMonth` method 208
- `dayOfWeek` method 208
- `dayOfYear` method 208
- `dealloc` method 46, 82
- deallocation, automatic 46, 72, 82
- debugging 57–62

- isolating errors 60
- declarations file 20, 22, 33–35, 84
 - rules and syntax 35–37
- decrement operator 177
- deployment issues 151–160
- description method 189, 190, 200, 205, 208
- `descriptionForResponse:inContext:` method 81, 151, 153
- `descriptionWithCalendarFormat:` method 208
- dictionaries
 - See* NSDictionary
- dictionary method 202
- `dictionaryWithContentsOfFile:` method 190–191, 202
- `dictionaryWithDictionary:` method 202
- `dictionaryWithObjects:forKeys:` method 202
- `dictionaryWithObjectsAndKeys:` method 202
- display groups 71–72
- displaying statistics 152–153
- `doubleValue` method 194

E

- elements 82–87
 - See also* WOElement
- dynamic 31–39, 70
 - See also* WODynamicElement
- associations 84–85
- request-response loop 73
- server side 31–37
- ID 83, 86
- `encodeObject:withCoder:` method 132
- `encodeWithCoder:` method 132–133
 - example 133
- @end 182
- ending sessions 135
- Enterprise Objects Framework 71–72
- EOEditingContext 72, 131–132
- EOKeyValueCoding 103
- equality of objects, determining 190
- error handling 154

F

- files, reading and writing 190–191
- floating point in WebScript 169
- `floatValue` method 194
- Foundation framework 189–209
 - framework 65–87
 - Enterprise Objects 71–72
 - Foundation 189–209

G

- garbage collection 46
- generating response 80–82

H

- `handleException:` method 154
- `handlePageRestorationError` method 154
- `handleRequest:` method 66, 74
- `handleSessionCreationError` method 154
- `handleSessionRestorationError` method 154
- host name, storage 68
- `hourOfDay` method 208
- HTML content
 - composition 69–71, 82–87
- HTML template 20, 21
- HTTP headers, modifying 53–54
- HTTP request, *See* WORequest
- HTTP response, *See* response
- HTTP server 26
 - communication with application 65–66
- HTTP version, storage 68

I

- id data type 167
- immutable 173, 189
- @implementation 182
- increment operator 177
- `indexOfObject:` method 198
- `indexOfObjectIdenticalTo:` method 198
- `init` method 46–50
 - for application 47–48
 - for component 49–50
 - for session 48

initObject:withCoder: method 132
 initWithCoder: method 132–133
 example 133
 input postprocessing 51
 insertObject:atIndex: method 199
 install 159–160
 instance variables, definition 165
 instance, definition 165
 integers in WebScript 169
 @interface 181
 intValue method 194
 invokeActionForRequest:inContext:
 method 52–53, 79–80
 declaration 70
 example 52–53
 ISAPI adaptor 26
 isEqual: method 190, 194, 198, 204
 isolating errors 60
 isTerminating method 135

J

Java support
 AppletGroupController class 143
 applets, *See* client-side components
 Association class 85–86
 subclassing 146–147
 debugging 57–58, 58–60, 61–62
 mapping of Foundation classes 146
 SimpleAsocationDestination
 interface 144–146

K

keys method 145
 keysSortedByValueUsingSelector:
 method 203
 key-value coding 35, 84, 103

L

lifetime of variables 167–168
 log file 151
 logic operators 176
 logString method 58–59
 logWithFormat: method 58–59

M

main() function or method 72–73
 managing state 111–140

client-side components 86
 memory leaks 157
 messages 171–174
 See also methods
 methods
 action 43–45
 return value 45
 automatic 171–172
 class 172–173
 common 43–54
 creation 173–174
 initialization 46–50
 invoking 171–174
 modern syntax 180
 multiple arguments
 modern syntax 180
 nesting 171
 request-handling 50–54
 writing 170–171
 modern syntax 180
 methods, definition 165
 minuteOfHour method 209
 modern syntax, WebScript 179–181
 monthOfYear method 208
 mutable 173, 189

N

naming reusable components 106
 nesting messages 171
 next.wo.client.controls 37–39
NeXT_ROOT 24
 nil 178–179
 NSAPI adaptor 26, 27
 NSArchiver 130
 NSArray 196–200
 See also arrays
 creating 169–170, 173–174
 Java 146
 methods 196–200
 NSCalendarDate 206–209
 conversion specifiers 206
 methods 207–209
 NSCoding protocol 132–133
 NSDictionary 200–205
 creating 169–170, 173–174

Java 146
 methods 201–205
 NSMutableArray 196–200
 methods 196–200
 NSMutableDictionary 200–205
 methods 201–205
 NSMutableString 191–195
 methods 192–195
 NSNumber 169
 NSString 191–195
 See also strings
 creating 169–170, 173–174
 Java 146
 methods 192–195

O

object, definition 165
 objectAtIndex: method 198
 objectForKey: method 116, 204
 Objective-C
 debugging 58
 and WebScript 183–186
 operators 176–178

P

page
 See also components
 caching, *See* caching pages
 composition 69–71, 82–87
 name, storage 68
 requesting directly in URL 53
 storing custom objects 131–133
 storing state 124–126
 page size
 limiting 158
 pageCacheSize method 136
 pageWithName: method 52
 overriding 138–139
 performance issues 156–160
 performParentAction: method 86, 100
 prefetching 158
 processes involved in running
 applications 25–28

R

- recording statistics 151–153
- refuseNewSessions: method 155
- regenerating request page 45
- registerForEvents method 73
- relational operators 176
- release method (Objective-C only) 46
 - example 137
- removeAllObjects method 199, 204
- removeObject: method 199
- removeObjectAtIndex: method 199
- removeObjectForKey: method 205
- removeObjectIdenticalTo: method 199
- removeObjectsForKeys: method 205
- request component 44
- request page 44, 73, 76–78
 - regenerating 45
- request URL 75
- request, accessing 51
- request-handling methods 50–54
- request-response loop 27
 - application level 66
 - conceptual overview 72–82
 - methods 50–54
 - page level 70
 - request 68–69
 - session level 67
 - starting 72–73
 - transaction level 68
- reserved words, WebScript 178
- response 69, 80–82
 - See also* WOResponse
 - generating 79–80, 80–82
 - manipulating 53–54
 - returning a page other than requested 52
- response component 44
- response page 44, 69, 79–80, 81
 - composition 69–71
 - returning a page other than requested 52
- restoreSession method 128–130
 - example 129
- reusable components 93–108
 - accessor methods 103

- binding to 95
- communication with parent 98–102
 - synchronization 102–104
- design tips 106–108
- HTML tags in 106
- location 105–106
- naming 106
- setting values in 97
- run method 73
- running applications 25–26

S

- saveSession: method 128–130
 - example 130
- scope of variables 167–168
- script file, *See* code files
- scripted classes 181–182
- scripting 165–186
- secondOfMinute method 209
- self 178–179
- sender ID 68
- session code file 19, 23
- Session.java 23
- Session.m 23
- Session.wos 19, 23
- sessionID method 76
- sessions 23, 66–68
 - accessing existing 75–76
 - code file, *See* session code file
 - creating 74–75
 - ending 135
 - ID 68, 75–76, 117
 - initialization 48
 - object, *See* WOSession
 - saving 82
 - setting minimum 155
 - state 115–117
 - timeout 134–135
 - variables 23, 115–117
- setArray: method 199
- setCachingEnabled: method 156
- setCalendarFormat: method 208
- setDictionary: method 205
- setDistributionEnabled: method 130
- setLogFile: method 151
- setMinimumActiveSessionsCount: method 155
- setObject:forKey: method 116, 204
- setPageCacheSize: method 136
- setPageRefreshOnBacktrackEnabled: method 139
- setSessionStore: method 123
- setString: method 195
- setTimeOut: method 134–135, 155
- sharing components 104
- sharing variables 171–172
- shutting down application 155–156
- SimpleAssociation 39
- SimpleAssociationDestination interface 144–146
- sleep method 46, 81
 - Objective-C example 137
 - and state storage 135, 137
- sortedArrayUsingSelector: method 198
- starting applications 25–26
- starting the request-response loop 72–73
- state 111–140
 - application 113–115
 - classes that manage 113–120
 - client-side components 86
 - component 118–120, 135–140
 - session 115–117
 - storage, *See* storing state
 - synchronizing 73
- statistics 151–153
 - displaying 152–153
- statistics method 152
- storing state
 - cookies 126–128
 - custom implementation 128–130
 - custom objects, storing 131–133
 - page 124–126
 - server 123–124
 - summary table 121–122
- string method 192
- stringByAppendingFormat: method 193
- stringByAppendingString: method 193
- strings 191–195
 - See also* NSString
 - constant 169–170
 - representing objects as 189

-
- stringWithContentsOfFile:
 - method 190–191, 193
 - stringWithFormat: method 192
 - stringWithString: method 192
 - subclass, definition 166
 - substringFromIndex: method 194
 - substringToIndex: method 193
 - super 178–179
 - superclass, definition 166
 - synchronization of components 85, 102–104
 - client-side 85–86
 - syntax, WebScript 166–181
 - modern 179–181
 - T**
 - takeValuesFromRequest:inContext:
 - method 51, 78–79
 - declaration 70
 - examples 51
 - terminate method 135
 - template 82–83
 - terminateAfterTimeInterval:
 - method 155
 - terminating application 155–156
 - time 206–209
 - timeout for session 134–135
 - timeOut method 134–135
 - trace methods 59–60
 - U**
 - URL
 - request, *See* request URL
 - requesting a page directly in 53
 - startup 25–26
 - V**
 - variables 166–170
 - accessing 171–172
 - application 23
 - assigning 168–170
 - scope 167–168
 - session 23
 - W**
 - web browser, *See* browser
 - WebApplication, *See* WOApplication
 - WebObjects adaptor, *See* adaptor
 - WebObjects application, *See* application
 - WebObjects architecture 65–87
 - WebObjects Builder
 - creating reusable components 97
 - data types 175
 - WebObjects framework 65–87
 - WebScript 165–186
 - %@ 59
 - @end 182
 - @implementation 182
 - @interface 181
 - assignment statement 168–170
 - categories 182–183
 - class object 172–173
 - constants 169–170
 - creating objects 173–174
 - data types 174–175
 - debugging 57, 58–60, 60–61
 - Foundation classes, using in 189–209
 - instantiating 173–174
 - messages 171–174
 - methods
 - automatic 171–172
 - class 172–173
 - invoking 171–174
 - modern syntax 180
 - nesting 171
 - writing 170–171, 180
 - nil 178–179
 - and Objective-C 183–186
 - operators 176–178
 - reserved words 178
 - self 178–179
 - statements 171–174, 175
 - modern syntax 180
 - static typing 174–175
 - super 178–179
 - syntax 166–181
 - modern 179–181
 - variables 166–170
 - accessing 171–172
 - scope 167–168
 - WebSession, *See* WOSession
 - .wo directory 19, 20–22, 77
 - See also* components
 - .woa directory 24–25
 - See also* application
 - WOAdaptor 65–66
 - See also* adaptor
 - instantiating 72
 - registerForEvents method 73
 - request-response loop 74
 - WOApplet 39
 - WOApplication 23–24, 66
 - See also* application
 - appendToResponse:inContext:
 - method 81
 - createSession method 75
 - handleException: method 154
 - handlePageRestorationError
 - method 154
 - handleRequest: method 66, 74
 - handleSessionCreationError
 - method 154
 - handleSessionRestorationError
 - method 154
 - init method 47–48
 - instantiating 72
 - invokeActionForRequest:inContext:
 - method 79
 - pageCacheSize method 136
 - pageWithName: method 52
 - overriding 138–139
 - refuseNewSessions: method 155
 - run method 73
 - setCachingEnabled: method 156
 - setDistributionEnabled: 130
 - setMinimumActiveSessionsCount:
 - method 155
 - setPageCacheSize: method 136
 - setPageRefreshOnBacktrackEnabled:
 - method 139
 - setSessionStore: method 123
 - setTimeOut: method 155
 - sleep method 82
 - state 113–120
 - takeValuesFromRequest:inContext:
 - method 78

- terminateAfterTimeInterval:
 - method 155
- trace methods 59–60
- WOAssociation 71, 84–85
- WOComponent 23–24, 69–71, 82–87, 105
 - See also* components
- appendToResponse:inContext:
 - method 81
- awake method 77
- descriptionForResponse:inContext:
 - 81
- init method 49–50, 77
- invokeActionForRequest:inContext:
 - method 79
- performParentAction: method 86, 100
- sleep method 81
- state 113–120, 135–140
- takeValuesFromRequest:inContext:
 - method 78
- WOContext 50, 69, 83
 - current component 84
 - request-response loop 74
- .wod file 20, 22, 33–35, 84
 - rules and syntax 35–37
- WODefaultApp 27–28
- WODisplayGroup 71–72
 - page refresh 140
- WODynamicElement 69–71
 - See also* elements, dynamic
- appendToResponse:inContext:
 - method 81
- invokeActionForRequest:inContext:
 - method 79
- takeValuesFromRequest:inContext:
 - method 78
- WOElement 69–71, 82–87
 - See also* elements
- WOHyperlink 32–35
- WORepetition 32–35
 - large repetitions 158
- WORequest 50, 68
- WOResponse 50, 65, 68, 69
 - See also* response
 - request-response loop 74
- .wos file, *See* code files
- WOSession 23–24, 66–68
 - See also* sessions
 - and EOEditingContext 72
 - appendToResponse:inContext:
 - method 81
 - dictionary 116
 - init method 48
 - instantiating 75
 - invokeActionForRequest:inContext:
 - method 79
 - isTerminating method 135
 - objectForKey: method 116
 - sessionID method 76
 - setObject:forKey: method 116
 - setTimeout: method 134–135
 - sleep method 81
 - state 113–120
 - takeValuesFromRequest:inContext:
 - method 78
 - terminate method 135
 - timeout method 134–135
- WOSessionStore 66–68, 76, 82
 - types 123
- WOStateStorage dynamic element 124
- WOStatisticsStore 151–153
 - setLogFile: method 151
 - statistics method 152
- WOStats component 152–153
- WOString 32–35
- writeToFile:atomically:
 - method 190–191, 195, 200, 205
- writing methods 43–54, 170–171
 - using modern syntax 180
- writing script files 165–186

Y

- yearOfCommonEra method 208