I n s i d e   W e b O b j e c t s

Web Applications

For WebObjects 5.2

November 2002

Contents

Figures, Listings, and Tables 9

Chapter 1 About This Document 13

Why Read This document 13
What You Should Already Know 14
Contents 15
Related Documentation 16

Chapter 2 Introduction to WebObjects 17

WebObjects Features 18
  Database Access and Independence 18
  Scalability 18
  Object Orientation 19
  Dynamic Publishing 20
  User Input 22
  Client-Server Applications 23
Development Tools 23
  Project Builder 24
  WebObjects Builder 24
  EOModeler 24

Chapter 3 Your First Project 25

Launch Project Builder 26
Using the New Project Assistant 26
The Project Builder Main Window 33
Modifying the Main Component 36
Building the Project 37
Running the Project 38

Chapter 4 Developing Dynamic Content 41

The WebObjects Builder Toolbar 42
Components and Classes 44
The Main Component 45
  Adding Methods 45
  Adding a WOString 47
HTML and WOD Files 50
Build and Run the Application 50
Response Generation 51
Maintaining State in the Component 53
  Adding the Variable to Count Method Calls 54
  Displaying the Count 55
  Increasing the Variable's Value 56
  Refreshing the Page 56
  The Counter in Action 58
Further Exploration 60

Chapter 5 Managing User Input 61

Request Processing 62
  Processing the Request 65
  Generating the Response 67
  Backtracking Cache 68
User Interface 69
Tracing the Request-Response Loop 73
Conditional Display With WOConditional Elements 77
Derived Properties 81

Chapter 6 Component Communication 85

Custom Objects 85
  Adding the Custom Class 86
  Following a Keypath 89
Defining a New Component 89
Modifying the Main component 94
Running the Application 97

Chapter 7 Managing State 99

The Session 100
Displaying and Editing Lists of Objects 101
The NSArray and NSMutableArray Classes 103
NSArray 104
NSMutableArray 104
Adding the NSMutableArray to the Session 104
Adding the WORepetition to Main 107
Editing the Users 109
Adding Users 110
Deleting Users 111
Running the Application 112
Benefits of Encapsulation 112
The NSArray and NSMutableArray Classes 103

Chapter 8 Database Basics 115

Database Structure 115
Tables 116
Rows 116
Uniquing 116
Not Null 117
Relationships 117
To-One Relationships 118
To-Many Relationships 119

Chapter 9 Introduction to Enterprise Objects 121

System Architecture 121
Enterprise Objects 124
EOControl 125
The Object Graph 125
The Editing Context 126
EOAccess 126
The Adaptor Level 126
The Database Level 127

Chapter 10 Creating an Enterprise Objects Application 129

Creating the Authors Database 129
Creating the Authors Model 132
  Adding the Author Entity to the Model 136
  The EOModeler Window 141
  Creating the AUTHOR Table 144
Creating the Authors Project 145
Customizing the Main Component 146
  Customizing Main.wo 147
  Customizing Main.java 149
Running the Authors Application 155
Browsing the Database 156
Further Exploration 158

Chapter 11 Using Custom Enterprise Objects 161

Generating a Custom Class 161
  Generating a Java Class From a Model Entity 162
  Adding a Java Class to the Project 163
Modifying the Authors Project 165
Adding Custom Logic 169
Using Custom Logic 170
Setting Default Values 173

Chapter 12 Working With Relationships 175

Completing the Authors Model 175
  Defining the Book Entity 176
  Creating the BOOK Table 177
  Defining the Model's Relationships 178
    What Are Delete Rules? 179
    Delete Rules in the Authors Model 180
Using Relationships in Your Code 181
  Update the Project's Author.java File 182
  Add the Book.java file to the Authors Project 184
  To-One Relationships in Java 185

To-Many Relationships in Java 185
Create the AuthorBookEdit Component 186
AuthorBookEdit.wo 186
AuthorBookEdit.java 189
Modify Session.java 190
Modify the Main.wo Component 193
Main.wo 193
Main.java 194
Run the Application 197
Deleting Authors 197
Create the ConfirmAuthorDelete Component 198
Modify ConfirmAuthorDelete.java 201
Modify Main.java 201
Run the Application 202
Sorting a Fetch 203

Appendix A Document Revision History 207

Glossary 209

Index 215

Figures, Listings, and Tables

Chapter 2 Introduction to WebObjects 17

Figure 2-1	Dynamic page generation in WebObjects	21

Chapter 3 Your First Project 25

Figure 3-1	The New Project Assistant	27
Figure 3-2	Choosing a location for the project	28
Figure 3-3	The Enable J2EE Integration pane of the Assistant	29
Figure 3-4	The Enable Web Services pane of the Assistant	30
Figure 3-5	The Choose EOAdaptors pane of the Assistant	31
Figure 3-6	The Choose Frameworks pane of the Assistant	32
Figure 3-7	Project Builder's main window	34
Figure 3-8	HelloWebObjects project folder	38
Figure 3-9	The HelloWebObjects application in action	39

Chapter 4 Developing Dynamic Content 41

Figure 4-1	WebObjects Builder toolbar	42
Figure 4-2	Web page displaying the Main component	51
Listing 4-1	URL that generates a new Session object	59
Listing 4-2	URL with session ID	60

Chapter 5 Managing User Input 61

Figure 5-1	The request-response loop	62
Figure 5-2	Structure of a component action URL	63
Figure 5-3	Binding the Favorite Food text field to personName	72
Figure 5-4	WOConditional elements	80
Figure 5-5	Adding a derived property	83

Listing 5-1 Example of a component action URL 63
Listing 5-2 Overriding the awake method 66
Listing 5-3 Overriding the sleep method 68
Listing 5-4 Tracing the request-response loop—the awake method 73
Listing 5-5 Tracing the request-response loop—the accessor and action methods 74
Listing 5-6 Implementation of entryIncomplete as a derived property 84
Table 5-1 Request-response processing phases 64
Table 5-2 Request-response processing time line 65

Chapter 6 Component Communication 85

Figure 6-1 The UserEdit.wo component in WebObjects Builder 94
Figure 6-2 The Main.wo component in WebObjects Builder 97
Listing 6-1 Instantiating the user instance variable in the constructor of the Main.java class 87
Listing 6-2 The submitChanges method of EditUser.java 93
Listing 6-3 The noDataEntered method of the Main.java class 95
Listing 6-4 Main component's editUser action method 96

Chapter 7 Managing State 99

Figure 7-1 Relationship between application and session 100
Figure 7-2 The Main.wo component with a WORepetition 109
Figure 7-3 The SessionState application in action 112
Listing 7-1 The Session.java class of the SessionState project 106
Listing 7-2 The editUser method of the Main.java class 109
Listing 7-3 The submitChanges method of the UserEdit.java class 110
Listing 7-4 The addUser method of the Main.java class 110
Listing 7-5 The deleteUser method of the Main.java class 111

Chapter 9 Introduction to Enterprise Objects 121

Figure 9-1 The Enterprise Objects approach 123

Chapter 10 Creating an Enterprise Objects Application 129

Figure 10-1 The OpenBase Manager window with the Authors database 132
Figure 10-2 Authors model with Authors entity 141
Figure 10-3 The Authors project in Project Builder 146
Figure 10-4 The Main.wo component with elements to maintain author information 149
Figure 10-5 The Authors application 155
Figure 10-6 The Data Browser window in EOModeler 157
Figure 10-7 Data Browser using filtering 158
Listing 10-1 The constructor in Main.java 150
Listing 10-2 The addAuthor method in Main.java 151
Listing 10-3 The deleteAuthor method in Main.java 152
Listing 10-4 The editAuthor method in Main.java 153
Listing 10-5 The updateAuthor method in Main.java 153
Listing 10-6 The saveChanges method in Main.java 154
Listing 10-7 The revertChanges method in Main.java 154
Listing 10-8 Fetch specification that uses sort orderings 159

Chapter 11 Using Custom Enterprise Objects 161

Figure 11-1 The Main.wo component after adding the fullName derived attribute to Author.java 171
Figure 11-2 The Main.wo component using the fullName derived attribute 172
Figure 11-3 The Authors application using the fullName method to display author information 173
Listing 11-1 Author.java generated by EOModeler 164
Listing 11-2 The Main.java file modified to use Author class instead of EOGenericRecord 166
Listing 11-3 The fullName method in Author.java 169
Listing 11-4 The constructor in Author.java—setting default value for lastName 174

Chapter 12 Working With Relationships 175

Figure 12-1 Relationships in the Authors model 179
Figure 12-2 The AuthorBookEdit.wo component in WebObjects Builder 188
Figure 12-3 Main.wo with the editBooks action and the Books WOHyperlink 194
Figure 12-4 The ConfirmAuthorDelete.wo component 200
Figure 12-5 The ConfirmAuthorDelete page in action 202
Listing 12-1 The methods that implement the author relationship in Book.java 185
Listing 12-2 The methods that implement the books relationship in Author.java 185
Listing 12-3 The deleteBook method in AuthorBookEdit.java 189
Listing 12-4 The addBook method in AuthorBookEdit.java 190
Listing 12-5 The constructor in Session.java 191
Listing 12-6 The fetchAuthorList method in Session.java 191
Listing 12-7 The addAuthor method in Session.java 191
Listing 12-8 The deleteAuthor method in Session.java 192
Listing 12-9 The constructor in Main.java 195
Listing 12-10 The addAuthor method in Main.java—uses the addAuthor method in Session.java 195
Listing 12-11 The deleteAuthor method in Main.java—uses the deleteAuthor method in Session.java 195
Listing 12-12 The editBooks method in Main.java—sends Author object to AuthorBookEdit component 196
Listing 12-13 The revertChanges method in Main.java—uses default editing context and the fetchAuthorList method in Session.java 196
Listing 12-14 The deleteAuthor method in ConfirmAuthorDelete.java 201
Listing 12-15 The deleteAuthor method in Main.java—returns the ConfirmAuthorDelete component 201
Listing 12-16 The sortAuthorList method in Session.java 203
Listing 12-17 The addAuthor method in Session.java 204

Appendix A Document Revision History 207

Table A-1 Document revision history 207

C H A P T E R 1

About This Document

WebObjects is an application server with tools, technologies, and capabilities to create Internet and intranet applications. It has an object-oriented architecture that promotes quick development of reusable components. WebObjects is extremely scalable and supports high-transaction volumes.

This document teaches you how to build websites that are backed by a robust and flexible Web applications.

Why Read This document

This document introduces developers to WebObjects application development using the Web-based approach. There are other approaches for developing WebObjects applications. For more information on them, see *Inside WebObjects: WebObjects Overview*.

You should read this document if you want to learn to maintain or develop WebObjects applications that users interact with using a Web browser and you are currently are at a beginner or intermediate level of experience with the WebObjects system.

This document will lead you through a hands-on exploration of the WebObjects programming environment. Through examples paired with explanations of the theory behind them, you'll learn to construct dynamic applications that allow your users to view and modify data from your databases.

This document has two main parts. The first part shows you how to use the WebObjects's tools to develop a Web application. You learn how to

- use the WebObjects development environment, including Project Builder and WebObjects Builder
- compile and run WebObjects applications
- manipulate user input in your programs
- create your own components and reuse them in your applications

The second part introduces the Enterprise Object technology and the use of databases as a persistent storage mechanism. In it you learn about

- basic database architecture
- the object-to-database connection
- designing your database schema
- using editing contexts to collect changes

What You Should Already Know

The WebObjects runtime is implemented entirely in Java. This document assumes you are familiar with Java and with the basic principles of object-oriented programming. While object-oriented programming experience is not necessary, WebObjects is an extensively object-based system. Familiarity with the Sun standard Java libraries beyond the basic object types like String and Integer is not necessary, because the Foundation libraries in WebObjects provide most of the functionality your WebObjects applications utilize.

Some familiarity with database architecture and OODBs (object-oriented databases) is beneficial, but not necessary. The Enterprise Object technology abstracts most of the specific details of databases away from your job as a developer, but an understanding of the underlying structure is always useful. "Database Basics" (page 115) provides a basic explanation of database architecture and usage.

Contents

This document has the following chapters:

- "Introduction to WebObjects" (page 17) introduces WebObjects's technologies and explains how they fit together.
- "Your First Project" (page 25) guides you through the creation of a simple WebObjects application project.
- "Developing Dynamic Content" (page 41) introduces the use of WebObjects elements to display dynamic data.
- "Managing User Input" (page 61) shows you the steps WebObjects takes when processing a request from a Web browser. It also introduces the use of conditional-display WebObjects elements and derived properties.
- "Component Communication" (page 85) shows how to send data from one component to another to maintain state.
- "Managing State" (page 99) introduces the use of the Session object to maintain state in a centralized location. It also shows the benefits of designing reusable components.
- "Database Basics" (page 115) provides a brief introduction to essential database concepts.
- "Introduction to Enterprise Objects" (page 121) shows the layers of an Enterprise Objects application and explains the role of the model.
- "Creating an Enterprise Objects Application" (page 129) guides you through the creation of a database with OpenBase Manager and a model with EOModeler. It shows you how to define an entity and how to create a database table based on an entity's definition. The chapter also shows how to perform data access using the Enterprise Object technology.
- "Using Custom Enterprise Objects" (page 161) guides you through the process of generating Java class files to add custom logic to enterprise objects.

- "Working With Relationships" (page 175) shows you how to create relationships between entities in EOModeler and how WebObjects implements those relationships in your application. It also shows you how to define a fetch specification and how to sort fetched data.

Related Documentation

If you need to learn the basics about developing WebObjects applications, you can find that information in the following documents:

- *Inside WebObjects: WebObjects Overview* provides you with a survey of WebObjects technologies and capabilities.
- *Inside WebObjects: Developing WebObjects Applications With Direct to Web* explores using rapid development technology to create dynamic Web applications.
- *Inside WebObjects: Java Client Desktop Applications* explains how to develop Swing-based applications with WebObjects.
- *Inside WebObjects: Using EOModeler* provides detailed information on the use of EOModeler to create models of your data.
- *Inside WebObjects: Enterprise JavaBeans* shows you how to develop Enterprise JavaBeans–based applications and how to use third-party–developed beans in your applications.
- *Inside WebObjects: JavaServer Pages and Servlets* describes how you can leverage WebObjects components in JSP-based applications. It also shows how you can deploy WebObjects applications as servlets.

For additional WebObjects documentation and links to other resources, visit http://developer.apple.com/webobjects.

C H A P T E R   2

Introduction to WebObjects

The Web started out as a means of disseminating static documents interconnected via hyperlinks. With its steady commercialization have come much greater demands on website developers. Today, it's not uncommon for a website to connect to a database, display dynamic data, take user input, and offer a reasonable representation of a desktop application.

Typically, each of these features is added by a developer at the behest of the customers or site owner. Dozens of incompatible mechanisms for solving the same problems exist, and any given site is a house of cards held together by expensive and frequent programmer intervention.

Another issue, all too real for many IS (Information Systems) managers, is the need to access data stored in databases from different vendors. Traditionally, developers have had to include custom code in their applications to be able to communicate efficiently with each database. Even if an organization standardizes on one database, if the need ever arises to upgrade due to performance or business reasons, then the custom code used to access the database becomes obsolete, slowing the transition process.

WebObjects solves all the common problems—dynamic page generation, user input, state management, interface with databases—that usually consume most of a developer's time, instead freeing the developer to spend her time constructing the logic that actually makes the application different.

In this chapter, you

- learn how WebObjects saves you from reinventing the wheel
- discover the features of WebObjects that make it a superior application development system
- learn about WebObjects's development tools

WebObjects Features

WebObjects solves many of the basic problems required for developing Web applications. Frequently, programmers reinvent the wheel to provide required features and capabilities to their applications or invest a lot of work integrating partial solutions. WebObjects comes with much of the logic needed by a Web application and provides an infrastructure that enables developers to work both effectively *and* efficiently.

Database Access and Independence

Almost any service beyond providing access to organized, static data relies on a database. Hence, it is very important to make database access powerful and efficient, both in use and implementation. WebObjects relies on the Enterprise Objects layer, which represents your database using Java objects (enterprise objects) with custom behavior and validation rules.

Other solutions for database access rely on technology such as embedding database access code, like SQL (Structured Query Language), within the Web pages themselves, which makes modifying the application much more difficult.

The Enterprise Object technology handles the database access tasks, like caching, fetching, saving, and relationship modeling, allowing you to concentrate on the implementation of your custom business logic. It even constructs the basic Java code required for your objects—you modify this code to add specialized logic, appropriate to your application. By providing this level of object abstraction, Enterprise Objects allows you to modify your database schema or even move to a totally different storage mechanism without any code modification.

Scalability

WebObjects is scalable at several levels, from development to deployment.

At the development level, individual pages and components can be developed in a modular fashion and reused, because they are each individual Java objects or WebObjects components. Further, a project can be easily broken into frameworks and products to facilitate code sharing and multiple-developer organization.

The WebObjects system itself scales over a broad range of user load, without any developer intervention. When a new request is made to your application, a new session is created. This session encapsulates the activities of a particular user. Caches are maintained by the application as well as by each of its sessions to speed response generation and minimize database access. In addition, WebObjects automatically caches component definitions to minimize the need to read files from disk. For more detail on session and state management see "Managing State" (page 99).

At deployment, WebObjects offers a linear scaling mechanism. The simplest deployment system is one computer running a Web server, WebObjects, and a database server. As your needs increase, the database server and Web server can be moved to other computers. Additional instances of your application can run in parallel and use the same database transparently. If demand increases further, additional application servers can be added using the same database and Web servers. WebObjects even automatically adds new application servers to its load-balancing system to ensure the most efficient access possible. (See *Inside WebObjects: Deploying WebObjects Applications* for more information on application deployment.)

Object Orientation

Experience has demonstrated that object orientation is a very useful paradigm for many development projects. WebObjects is designed on an object-oriented model, with every part of the system, from components to the process of generating pages itself, organized using an object model.

As a developer, you gain many benefits from this model. You can customize the process of page generation by adding your logic to standard methods, which are invoked at determined stages. This is possible because all WebObjects components inherit from the WOComponent class. A **component** is an object that encompasses the look and behavior of a Web page or a portion of one.

Pages or components that share behavior—for example, a component that displays search results for each of the entities you have in your database—can gain the usual benefits of inheritance, saving you from writing duplicate code and all the attendant inefficiencies.

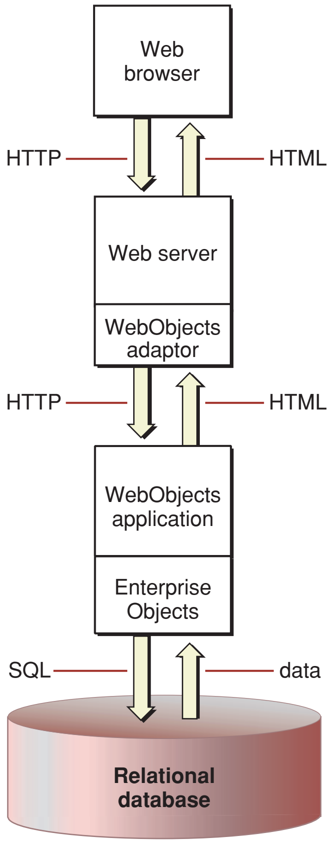
You are also freed from the need to think of your data as anything but first-class objects. Rather than thinking about tables, columns, and rows in a database when retrieving information or manipulating relationships, you manage data by creating objects or arrays of objects, and by invoking their methods. The Enterprise Object technology manipulates the database for you to reflect your changes.

Dynamic Publishing

The simplest websites are composed of static pages in HTML (hypertext markup language) that are served to a user's browser. These pages change infrequently, and the scope of the changes is such that it is practical to manually update the documents on the rare occasion that they change.

Increasingly, though, users need more dynamic or frequently updated content via the Web. Weather reports, news stories, and stock quotes change frequently, and it's impractical or impossible to alter static text documents on a Web server to reflect new information often enough to be useful. Instead, software generates HTML code with the latest information when requested. (The pages generated can include current information obtained from a data source, often a relational database.) This code is then sent to a user's Web browser. Figure 2-1 illustrates how requests by a Web browser are processed to generate a response page.

Figure 2-1 Dynamic page generation in WebObjects



Each page in a WebObjects application is created as Web component. A Web component contains an HTML template that defines the structure of the Web page, a Java class that provides event-handling logic and business logic, and a bindings file that links the page's elements to instance variables or methods defined in the Java class. Web components other components in a recursive structure, or special WebObjects elements. For more on Web components and WebObjects elements, see "The Main Component" (page 45).

When the template is requested, WebObjects fills in the missing data represented by the tags by calling the methods associated with the element's attributes and inserting the result into the returned HTML code on demand. The called methods might access a database, perform calculations, or carry out any other custom logic you have defined.

Several other common idioms exist for dynamic page generation. These range from various third-party solutions to hand-made Perl or Java servlet systems. Few offer the easy database access or close association with Java logic that WebObjects provides. Another problem common to most other solutions is the lack of scalability—page-based logic rapidly becomes impossible to maintain as the size of the site increases.

User Input

There is a qualitative change in the kind of services your application can offer to its users when it no longer relies solely on Web page navigation for control. By allowing the users of your application to provide data, you increase the application's scope dramatically beyond that of solely sharing already extant information. For example, you have to devise logic that validates the data the user enters, so that your database does not become corrupted.

A few mechanisms for user input are in common use. Most involve encoding the data into a string that is attached to the URL (Uniform Resource Locator) of the page the user requests and parsing it on the server end through a custom program written to use the CGI (Common Gateway Interface) protocol.

Instead, WebObjects follows the same paradigm used for dynamic page generation. Standard HTML form elements can be associated with variables and methods in your Java code and, when the user submits a form, the methods indicated are called with the user-entered data as an argument. Your methods can take any action you determine to record this input—and if you associate a form element with a database field via the enterprise object property representing it, user input is recorded in the database automatically.

Client-Server Applications

One of the most pressing issues in Web application development is the need to determine ways of maintaining state—the information about the user's interaction with the application during a given period—between requests. Because HTTP (Hypertext Transfer Protocol) is a stateless protocol, there is no connection maintained between the user's Web browser and the Web server. This leaves the responsibility for maintaining state up to you.

In a desktop application, the notion of state is implicit: there is only one user. In a Web application, however, there may be hundreds of simultaneous users.

There are two ways of maintaining state in a Web application: using cookies and customizing URLs.

When **cookies** are used, information is stored on a Web browser (the client) by the Web server. When the server needs to determine the current state of a client, it retrieves the cookie. The drawback of this approach is that Web browsers can be configured to refuse cookies. In such cases, the application's functionality can be severely limited.

To ensure that state can be maintained, whether cookies are enabled or not on the client, many Web applications use customized URLs, in which they add the kind of information that would otherwise be stored in a cookie.

WebObjects can maintain state using either of these approaches. However, you don't have to worry about which one is actually used. All you do is store the required state information in an instance of the Session class. When a request is processed, WebObjects automatically activates the Session instance associated with the user who initiated the request (the fact that such information was retrieved from a cookie or from the URL is transparent to you). See "Managing State" (page 99) for more information.

Development Tools

For the most part, you interact with the WebObjects development environment using three tools: Project Builder, WebObjects Builder, and EOModeler.

Project Builder

Project Builder is your primary WebObjects development tool. It provides an integrated development environment that allows you to edit code, organize resources, and compile your project, as well as facilitate your work with other programs like WebObjects Builder when you edit your WebObjects components.

Project Builder is installed as part of the developer package. It is located in `/Developer/Applications`.

WebObjects Builder

WebObjects Builder is a specialized application for editing WebObjects components. It handles editing the HTML file as well as the WOD (WebObjects data) file that controls the connection between your HTML components and your Java code.

WebObjects Builder is located in `/Developer/Application`.

EOModeler

EOModeler is a tool for constructing a **model** that relates a data-store structure (a database or a naming and directory service) to Java objects. With EOModeler you can create a model in two ways:

- Reverse-engineer an existing data-source schema.

  EOModeler reads your data source's schema and creates a model from it.

- Create the data model manually.

  You can create a new model by defining the entities, attributes, and relationships that represent your data model. You can then have EOModeler create the underlying database tables. This is the approach used in "Creating the Authors Model" (page 132).

For more information on EOModeler, see *Inside WebObjects: Using EOModeler*.

Constructing a good model is a very important part of developing a database-based WebObjects application. With a properly constructed model, an application practically writes itself.

Your First Project

WebObjects is a large system, built of many complex layers. Fortunately, those layers are largely self-contained, allowing you to ignore some complexity until you find you want more control over specific processes. This allows you to construct a simple WebObjects application that is fully functional, without needing to have complete knowledge of the underlying system. As you add more features to the applications you create, you explore the WebObjects frameworks to a greater depth.

Most of your access to the WebObjects system will be through the tools you use to create applications in it—Project Builder, WebObjects Builder, and for data-driven applications, EOModeler.

Your first WebObjects application will display “Hello, World!” in your Web browser. Though this project is trivial, it does serve as an example with which to examine the interface of Project Builder. Also, a successful build and launch verifies that your development environment is correctly installed and configured.

In this chapter, you

- run Project Builder and perform initial setup
- learn about the components of a project
- use the New Project Assistant
- learn about the parts of the Project Builder editing window
- build and run a simple application

Launch Project Builder

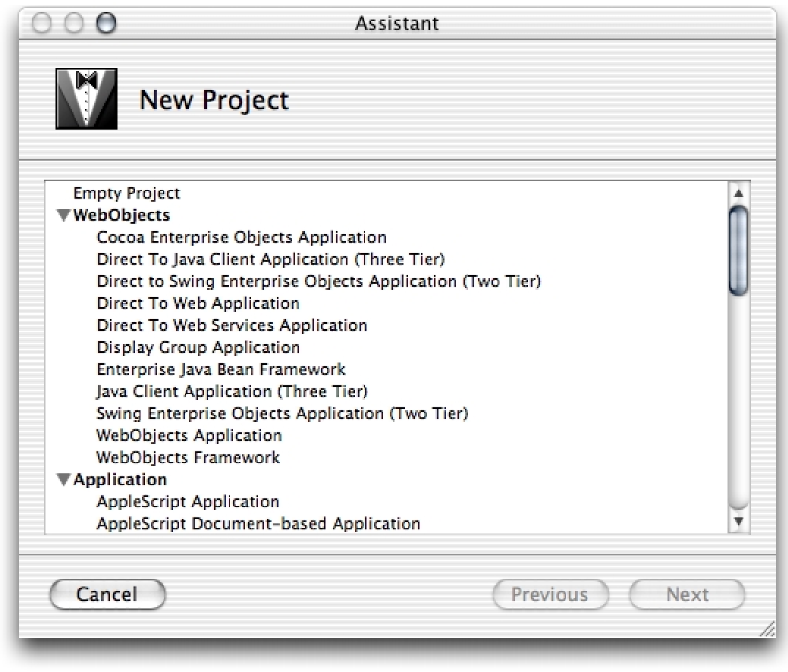
1. Navigate to `/Developer/Applications`.
2. Double-click the Project Builder icon.

   The first time you run Project Builder, you are greeted by a setup assistant that walks you through some of the essential configuration settings of Project Builder. At this point, you could customize the build system used to compile your projects, but for now, accept the default options on each pane.

Using the New Project Assistant

When you first launch Project Builder, you see only its menu bar. To create a project to work in, choose New Project from the File menu. The Project Builder Assistant appears, walking you through a few steps to create a new project.

Figure 3-1 The New Project Assistant



Empty Project
WebObjects
Cocoa Enterprise Objects Application
Direct To Java Client Application (Three Tier)
Direct to Swing Enterprise Objects Application (Two Tier)
Direct To Web Application
Direct To Web Services Application
Display Group Application
Enterprise Java Bean Framework
Java Client Application (Three Tier)
Swing Enterprise Objects Application (Two Tier)
WebObjects Application
WebObjects Framework
Application
AppleScript Application
AppleScript Document-based Application

There are several project types to choose from. Each starts out with a slightly different set of files and configuration to facilitate particular types of applications, from command-line tools to desktop applications. The following are two types of WebObjects project you can develop:

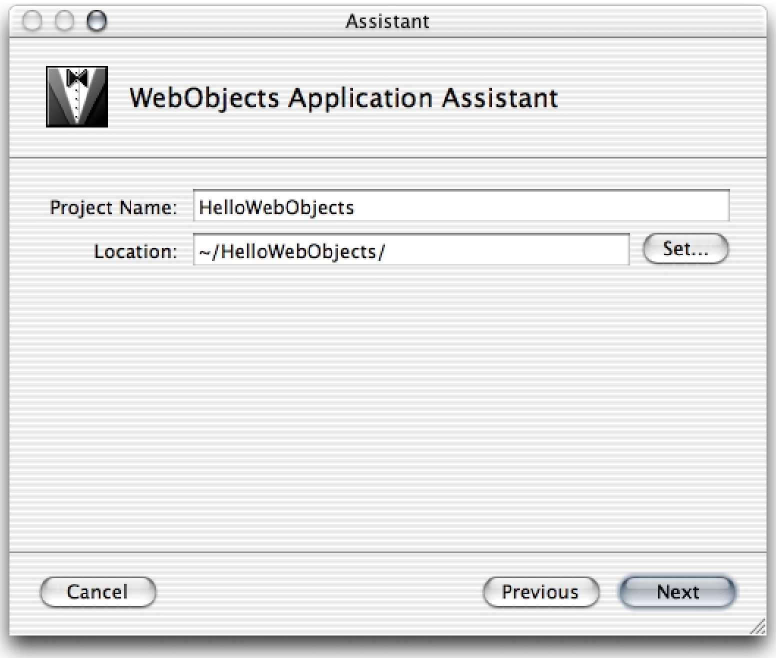
- **WebObjects Application.** This project type is the starting point for WebObjects applications. It provides one Web page, a system for moving resources like images to your Web server's document root during installation, and other basic components like the Session and Application classes.
- **WebObjects Framework.** A framework is a bundle of related code, resources such as sounds and graphics. You can make your applications depend on your frameworks as a means of sharing code between applications. Your WebObjects applications are based on the JavaWebObjects framework, and you can write your own as well.

Follow these steps to build your first WebObjects application project:

1. Select WebObjects Application from the list of templates and click Next.
2. Type `HelloWebObjects` in the Project Name text input field and click Next.

   If you don't want to use the default project location, click Set and navigate to the directory where you want to store your project.

Figure 3-2 Choosing a location for the project



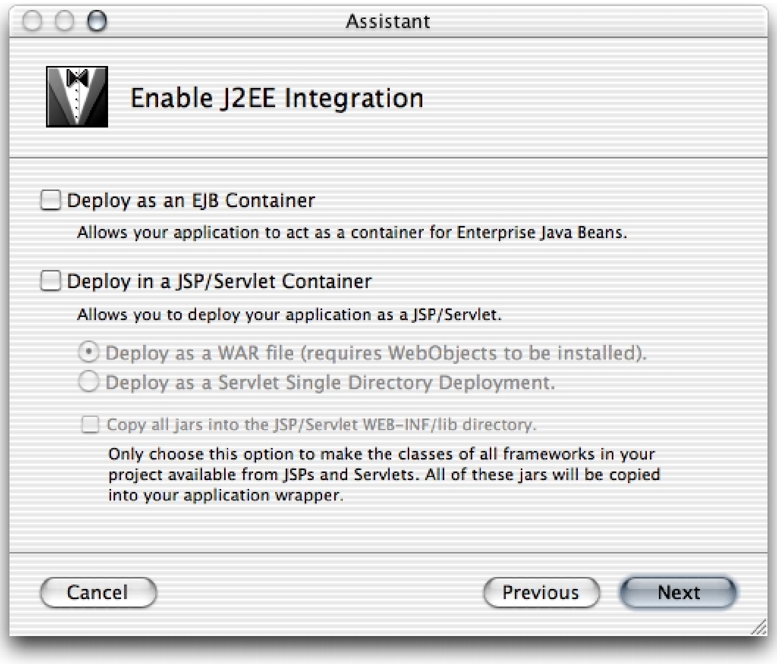
Project Name: HelloWebObjects

Location: ~/HelloWebObjects/

3. The Enable J2EE Integration pane of the Assistant allows you to tell Project Builder whether you want to deploy the application as an Enterprise JavaBeans (EJB) container. It also allows you to specify whether to deploy the application in a JavaServer Pages (JSP) container as a servlet. For the purposes of this example, don't select either option; just click Next.

For more information on EJB, JSP, and servlets in WebObjects, see *Inside WebObjects: Enterprise JavaBeans* and *Inside WebObjects: JavaServer Pages and Servlets*.

Figure 3-3 The Enable J2EE Integration pane of the Assistant


Assistant

Enable J2EE Integration

☐ Deploy as an EJB Container

Allows your application to act as a container for Enterprise Java Beans.

☐ Deploy in a JSP/Servlet Container

Allows you to deploy your application as a JSP/Servlet.

● Deploy as a WAR file (requires WebObjects to be installed).

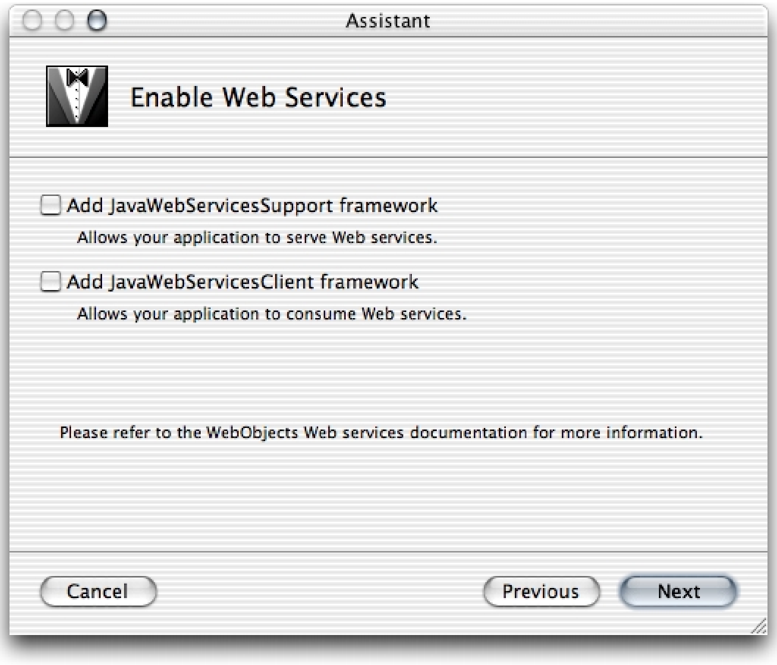
○ Deploy as a Servlet Single Directory Deployment.

☐ Copy all jars into the JSP/Servlet WEB-INF/lib directory.

Only choose this option to make the classes of all frameworks in your project available from JSPs and Servlets. All of these jars will be copied into your application wrapper.

4. The Enable Web Services pane of the Assistant lets you indicate whether you want the application to serve or consume Web services. Again, just click Next.

Figure 3-4 The Enable Web Services pane of the Assistant



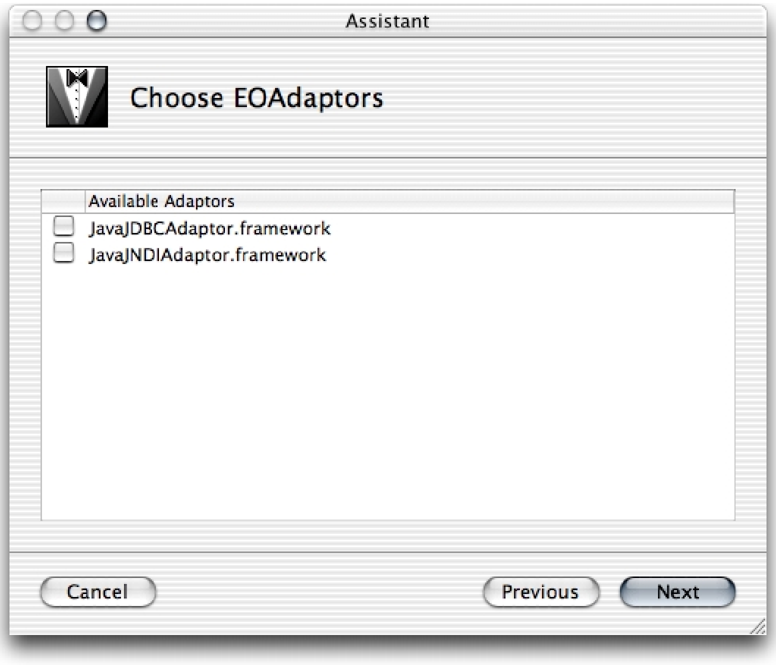
☐ Add JavaWebServicesSupport framework
Allows your application to serve Web services.

☐ Add JavaWebServicesClient framework
Allows your application to consume Web services.

Please refer to the WebObjects Web services documentation for more information.

5. The Choose EOAdaptors pane of the Assistant allows you to specify which Enterprise Object adaptors you plan to use in your application. Enterprise Objects is the WebObjects technology that allows you to perform database operations through the manipulation of Java objects. This application doesn't make use of a database. Make sure no adaptors are selected and click Next.

Figure 3-5 The Choose EOAdaptors pane of the Assistant

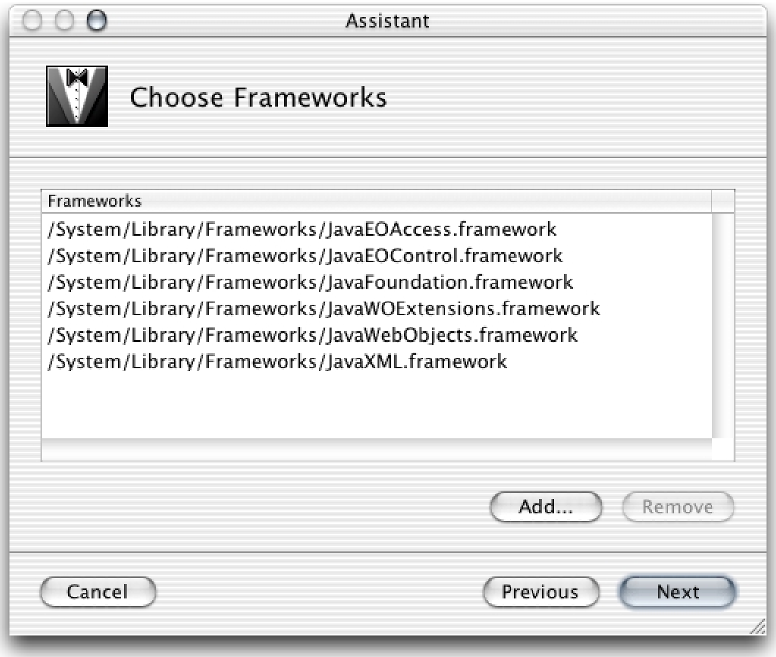


Available Adaptors

- [ ] JavaJDBCAdaptor.framework
- [ ] JavaJNDIAdaptor.framework

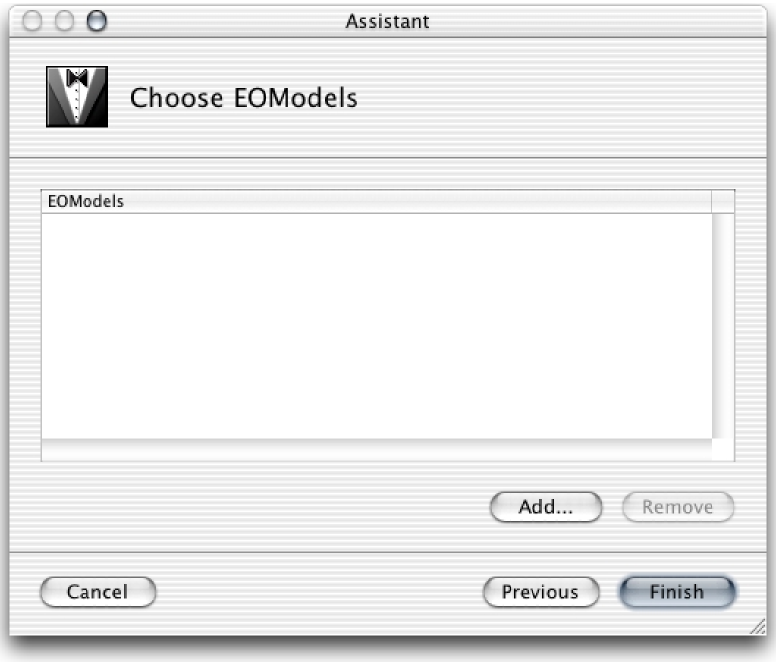
6. The Choose Frameworks pane of the Assistant allows you to add frameworks needed by your application to the project. This simple application needs no additional frameworks. Click Next.

Figure 3-6 The Choose Frameworks pane of the Assistant



/System/Library/Frameworks/JavaEOAccess.framework
/System/Library/Frameworks/JavaEOControl.framework
/System/Library/Frameworks/JavaFoundation.framework
/System/Library/Frameworks/JavaWOExtensions.framework
/System/Library/Frameworks/JavaWebObjects.framework
/System/Library/Frameworks/JavaXML.framework

7. The Choose EOModels pane of the Assistant allows you to add data models to your project. Data models let you manage data stored in a database using an object-oriented approach. For more information on EOModels, see *Inside WebObjects: Using EOModeler*.



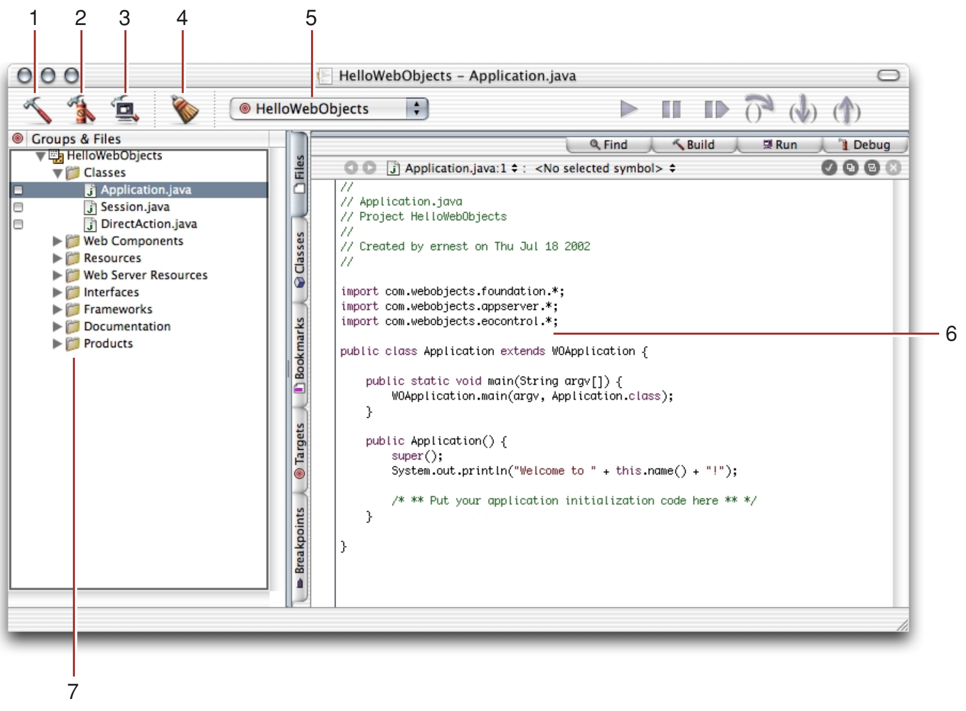
8. Click Finish to create the project.

   The main window of Project Builder appears.

The Project Builder Main Window

When you're finished with the Assistant, you'll see the Project Builder main window, as shown in Figure 3-7. The main window organizes all the files in your project and provides all the tools you need to edit, build, and debug them.

Figure 3-7 Project Builder's main window



```
//
// Application.java
// Project HelloWebObjects
//
// Created by ernest on Thu Jul 18 2002
//

import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;

public class Application extends WOApplication {

    public static void main(String argv[]) {
        WOApplication.main(argv, Application.class);
    }

    public Application() {
        super();
        System.out.println("Welcome to " + this.name() + "!");

        /* ** Put your application initialization code here ** */
    }

}
```

1. Build project.
2. Build and debug.
3. Build project and run application.
4. Clean project.
5. Target menu.
6. Code editor.
7. Groups & Files list

The left pane is a tabbed pane used for organization. In the Groups & Files pane, which is initially visible, there are several groups of files, each with a disclosure triangle.

- Classes

  This group initially contains the `.java` files for the Application, Session, and DirectAction classes that your application uses. You can customize your application by changing these files. In addition, when you add new classes to your project, they are stored here by default.

- Web Components

  Each Web page or component you create is stored within its own subgroup in the Web Components group. Each subgroup contains the files that define the HTML representation and WebObjects behavior for each component. Initially, only the Main subgroup is present.

  Inside the Main subgroup you find three items: `Main.wo`, `Main.java` , and `Main.api`. They define the look and behavior of the Main component.

- Resources

  Graphics, sounds, and movies for your components are stored in this folder. In data-driven applications, the model files (with the extension `.eomodeld`) are stored here.

- Web Server Resources

  Some resources may be referenced not only by your WebObjects applications but also by static pages in other parts of the site. Resources in this folder are moved to a location outside of the application bundle, where they can be accessed by other means as well.

- Frameworks

  Every WebObjects project is dependent on at least the JavaWebObjects framework, which contains the essential code behind WebObjects. You can add additional frameworks to your project by choosing Project > Add Files.

- Documentation

  Documentation for your project can be organized by Project Builder.

- Products

  The files created when you compile your application are listed under this group. It includes the executable, an organized tree of resources for components, and localized versions of the components themselves.

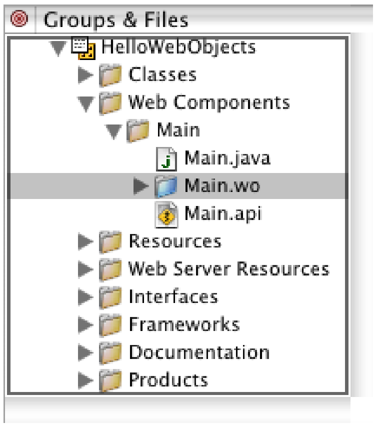
The Classes pane provides a simple class browser that you can use to view the class hierarchy and view the methods provided by each class or interface.

The other three panes, Bookmarks, Targets, and Breakpoints, are explained in greater detail later in the book.

Modifying the Main Component

Now you'll use WebObjects Builder to modify the Main component.

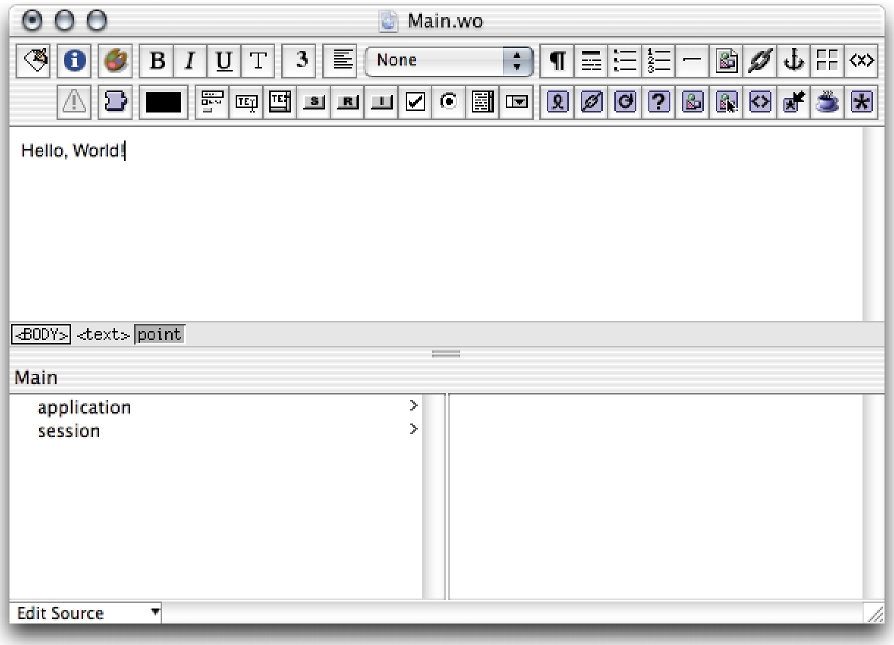
1. Open `Main.wo`.



   In Project Builder, double-click the `Main.wo` component in the Main subgroup in the Web Components group in the Groups & Files list. The WebObjects Builder application opens and displays a window for `Main.wo`.

2. Modify `Main.wo`.

   Enter `Hello, World!` in the content pane.



3. Save `Main.wo`.

   Choose File > Save.

Building the Project

All that remains is to compile the project and run it. When you start the build process, Project Builder does more than compile the Java bytecode from your files. First, only files that have changed since the last build are compiled, to save time. Project Builder also gathers all the resources required for your project, organizes them for your Web server, and compresses your Java class files into a JAR (Java Archive) file.

When you choose Build from the Build menu, the Build pane appears so you can watch the progress of the build. This is also the pane that displays Java compilation errors if your project has any, but its output is frequently very useful even when it doesn't contain error messages.

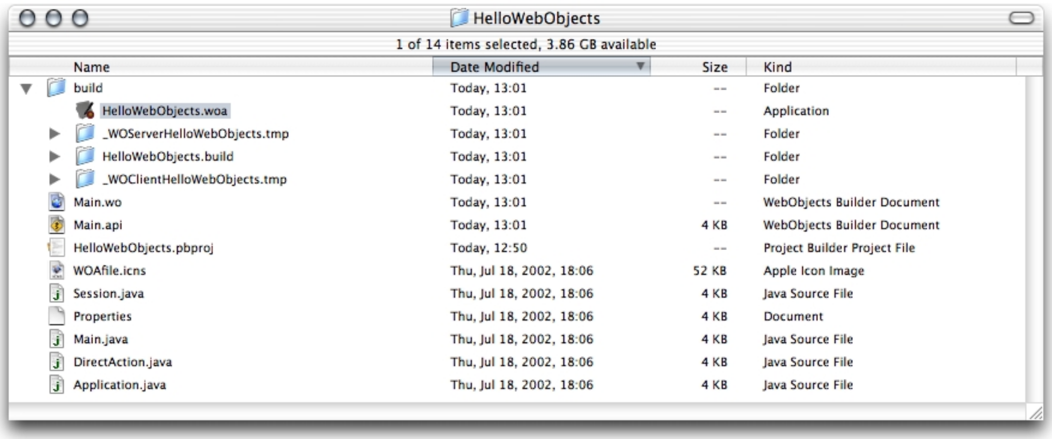
In Project Builder, choose Build > Build or click the Build button (the one with the hammer) in the main window.

Because you didn't modify any Java code, you shouldn't encounter any compilation errors. When the compilation progress bar is complete, you're ready to run your project.

Running the Project

Unless you changed the default location when you first ran Project Builder, you now have a bundle called `HelloWebObjects.woa` in the `build` directory at the top level of your project's directory, as the Finder window in Figure 3-8 shows.

Figure 3-8 HelloWebObjects project folder



HelloWebObjects

1 of 14 items selected, 3.86 GB available

Name	Date Modified	Size	Kind
build	Today, 13:01	--	Folder
HelloWebObjects.woa	Today, 13:01	--	Application
_WOServerHelloWebObjects.tmp	Today, 13:01	--	Folder
HelloWebObjects.build	Today, 13:01	--	Folder
_WOClientHelloWebObjects.tmp	Today, 13:01	--	Folder
Main.wo	Today, 13:01	--	WebObjects Builder Document
Main.api	Today, 13:01	4 KB	WebObjects Builder Document
HelloWebObjects.pbproj	Today, 12:50	--	Project Builder Project File
WOAfile.icns	Thu, Jul 18, 2002, 18:06	52 KB	Apple Icon Image
Session.java	Thu, Jul 18, 2002, 18:06	4 KB	Java Source File
Properties	Thu, Jul 18, 2002, 18:06	4 KB	Document
Main.java	Thu, Jul 18, 2002, 18:06	4 KB	Java Source File
DirectAction.java	Thu, Jul 18, 2002, 18:06	4 KB	Java Source File
Application.java	Thu, Jul 18, 2002, 18:06	4 KB	Java Source File

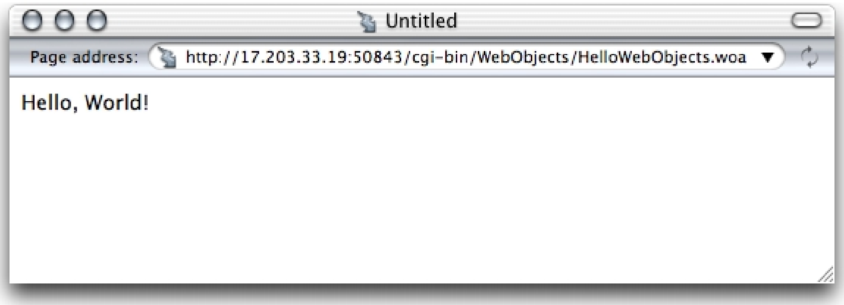
Choose Debug > Run Executable or click the Build and Run button (the one with the hammer and computer) in Project Builder's main window.

After a moment the Run pane appears, displaying the output from your application as it runs:

```
Reading MacOSClassPath.txt ...
Launching HelloWebObjects.woa ...

...

Creating adaptor of class WODefaultAdaptor listening on port -1 with a listen
queue size of 128 and 2 WOWorkerThreads.
Creating LifebeatThread now with: HelloWebObjects -1 1085 30000
Welcome to HelloWebObjects!
Opening application's URL in browser:
http://localhost:49189/cgi-bin/WebObjects/HelloWebObjects
Waiting for requests...
```

After the last line appears, the URL shown opens automatically in your default browser.

Figure 3-9 The HelloWebObjects application in action



http://17.203.33.19:50843/cgi-bin/WebObjects/HelloWebObjects.woa

Hello, World!

C H A P T E R 4

Developing Dynamic Content

Having dynamic content means only that the information your website or application displays varies based on some conditions. Examples of dynamic content include news sites, product catalogs where entries change and users can accumulate a shopping cart of items, and online polls and statistics.

With WebObjects, you can generate your dynamic content several ways. You can use all the programming logic you're familiar with to determine which image to display or what information to present; you can define templates that are filled in from a database; you could also allow the user to enter data to be displayed.

In this chapter, you add some elements to your Web page and learn how to connect them to Java code that you write. You also learn in general how the HTML code, WebObjects components, and their Java classes relate to each other, and you are introduced to the **request-response loop**, the system WebObjects uses to interact with the users of your application.

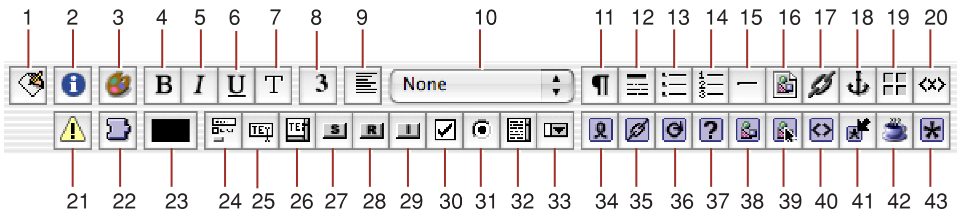
In this chapter, you

- learn about the WebObjects Builder toolbar and the WebObjects Builder Inspector
- use WebObjects Builder to bind WOElements to your Java code
- use methods to provide dynamic data in your Web page
- learn the basics of the request-response cycle
- customize the Main component, the default entry point of your application

The WebObjects Builder Toolbar

Figure 4-1 shows the WebObjects Builder toolbar, and the following list identifies all the controls in the toolbar. The identifying phrases are the help tags that appear for the controls, and these phrases are used to refer to the controls in this document. For example, control 24 is the Add WOForm button.

Figure 4-1 WebObjects Builder toolbar



1. Switch preview, layout and source views.
2. Show Inspector
3. Show Palette
4. Bold
5. Italic
6. Underline
7. Typewriter (fixed-width font)
8. Change Font Size
9. Change Text Alignment
10. Change Font Typeface
11. Add Paragraph
12. Add Heading

13. Add Unordered List
14. Add Ordered List
15. Add Horizontal Rule
16. Add Image
17. Add Link
18. Add Anchor
19. Add Table
20. Add Customizable Marker
21. Show Validation Warnings
22. Show API Editor
23. Change Text Color
24. Add WOForm
25. Add WOTextField
26. Add WOText
27. Add WOSubmitButton
28. Add WOResetButton
29. Add WOImageButton
30. Add WOCheckBox
31. Add WORadioButton
32. Add WOBrowser
33. Add WOPopUpButton
34. Add WOString
35. Add WOHyperlink
36. Add WORepetition
37. Add WOConditional
38. Add WOImage
39. Add WOActiveImage

40. Add WOGenericContainer
41. Add WOComponentContent
42. Add WOApplet
43. Add Custom WebObject

Components and Classes

Each Web page displayed in a user's Web browser is a WebObjects component. A component is made up of several parts:

- **HTML file.** This portion of the component is mostly standard HTML code. A component is either a complete HTML page, with `<HTML>` and `</HTML>` tags, or a shorter segment of HTML code that can be inserted inline into another component.

  In addition to regular HTML tags and text, a component can contain special tags used by WebObjects, the `<WEBOBJECT> and </WEBOBJECT>` tags. Web browsers never see these tags because WebObjects replaces them with regular HTML code before sending them to the browser.
- **WOD file.** This is the glue between your HTML file and your Java code. Every WebObjects element used in a component has an entry in this file specifying its parameters, such as Java methods to call for data. WOD stands for WebObjects data.
- **Java file.** Every component has a Java class file associated with it. These classes inherit from the WOComponent class, which provides the basic functionality a component needs. To customize behavior, you can add your own variables and logic to intercede in the built-in system
- **API file.** If you design your own components for reuse, they may rely on certain information being present in their Java code definitions. The API file lists the parameters for your custom components.
- **WOO file.** Contains information about **display groups**, special components used to display database information. WOO stands for WebObjects object.

The Web Components group—in the Groups & Files list in Project Builder's main window—lists all the components of a project. Each item is itself a subgroup named after the component. Such groups contain the Java and API files for the component. The HTML, WOD, and WOO files are contained in a subgroup of the component subgroup, named using the component's name with the `.wo` extension (`Main.wo`, for example). The contents of the `.wo` group are maintained by WebObjects Builder.

The Main Component

By default, every WebObjects application includes a Main component. This component, initially empty, is the first page displayed to users unless you arrange otherwise. It can be used as the login page for the rest of your application.

The initial Main component is entirely empty. In this section, you add a method that calculates the date to the Java class, add a WOElement to the page, and use the WOD file to bind it all together.

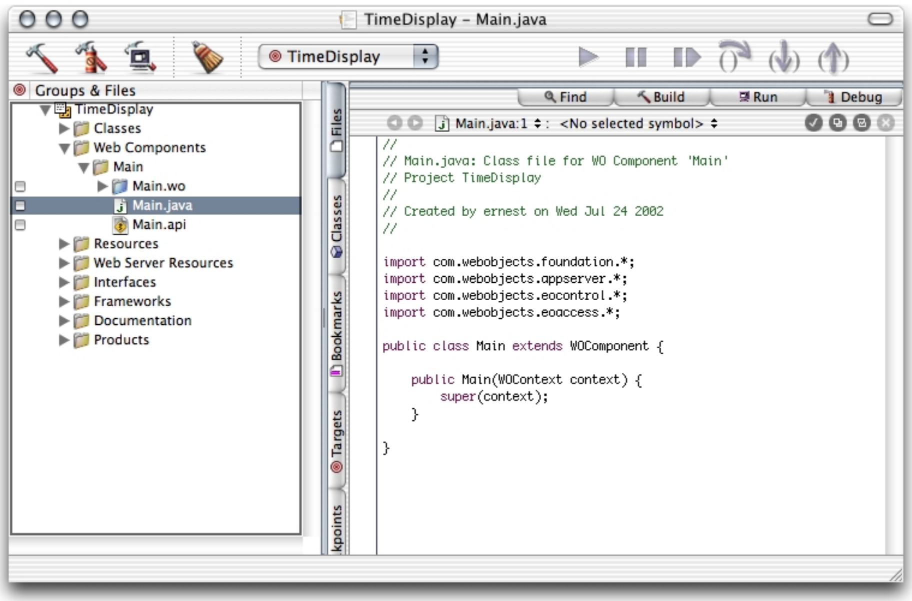
Adding Methods

First, you add a Java method to the `Main.java` file. This method simply returns the current date when it is called.

1. Create a WebObjects application project and name it TimeDisplay.

   For details on how to create a new project see "Your First Project" (page 25).

2. Select `Main.java` in the Groups & Files list in Project Builder's main window.



```
//
// Main.java: Class file for WO Component 'Main'
// Project TimeDisplay
//
// Created by ernest on Wed Jul 24 2002
//

import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;

public class Main extends WOComponent {

    public Main(WOContext context) {
        super(context);
    }

}
```

3. Add the following code to the `Main.java` file. This is a public method that returns the current time using the NSTimestamp class.

```
/**
 * Creates an NSTimestamp object with the current
 * date and time.
 *
 * @return current date and time.
 */
public NSTimestamp currentTime() {
    return new NSTimestamp();
}
```

   Notice that the Main class extends WOComponent.

   The WOComponent class defines dozens of methods needed by WebObjects. Many of these methods are introduced later in this book.

4. Save the `Main.java` file by choosing File > Save.

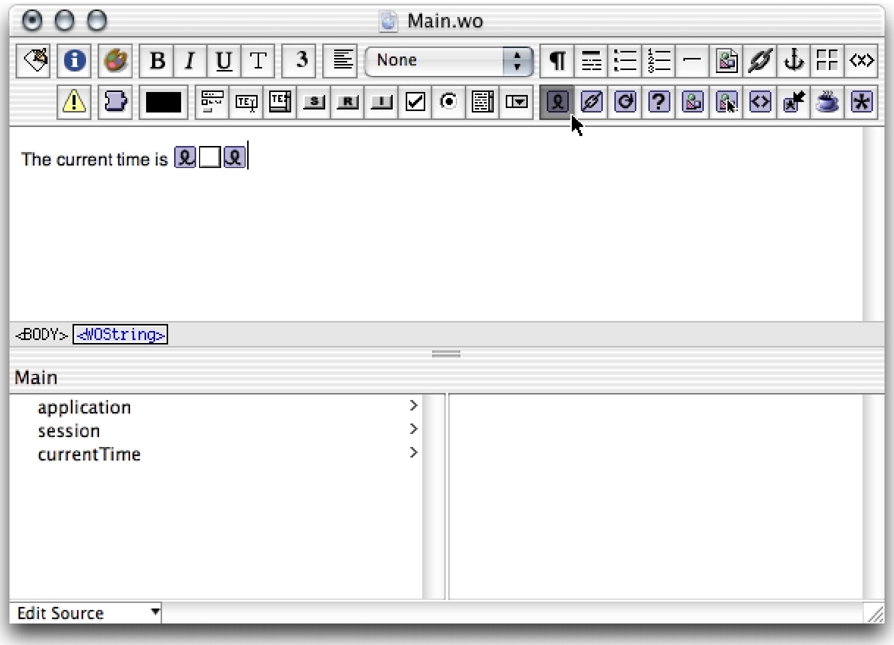
Adding a WOString

To display dynamic text, you add a WebObjects element to the Main component. This element is the WOString, which is used to display dynamic string data in a page. Such strings can be the output of a method that returns a String object or another object that can be converted to a String object.

1. Open the Main component with WebObjects Builder by double-clicking `Main.wo` in Project Builder.
2. Add text and a WOString.

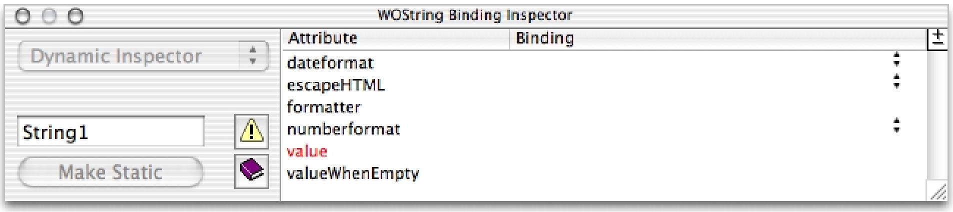
   Enter `The current time is` in the content editor in WebObjects Builder's main window.

   With the cursor at the end of the new text, press the Space bar and click the Add WOString button in the toolbar.



3. Open the WOString Binding Inspector.

   Select the WOString element and click the Show Inspector button in the toolbar. The WOString Binding Inspector appears.



   If the Inspector appears, but doesn't look like the one shown, click the WOString you just inserted. The Inspector displays information about the element that is currently selected.

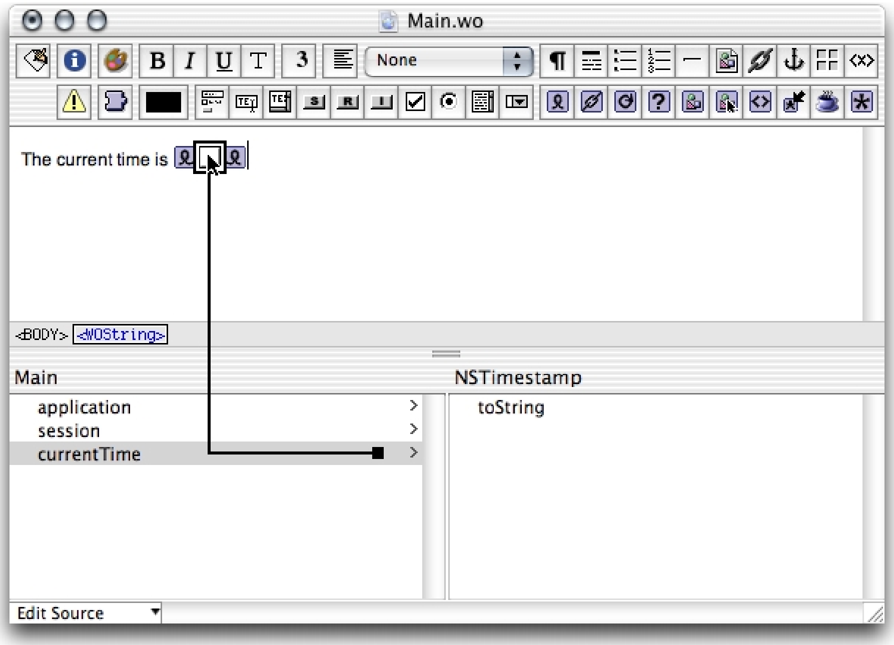
   The Inspector displays the attributes for WOString elements. Each of them can be set either to static values or by binding them to instance variables or methods in your code, which provide a value at runtime.

   Notice that the `value` attribute is displayed in red. This means that this binding is required. In this case, the `value` attribute's binding produces the text that the WOString displays, and the other attributes affect how the string is displayed or what to display when the attribute contains no value. You use this WOString to display the current time.

4. Bind the WOString's `value` attribute to the `currentTime` method.

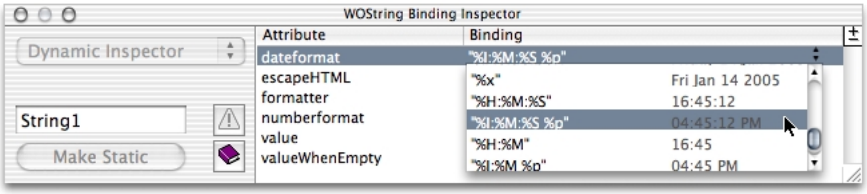
   Notice that the name of the `currentTime` method you entered in `Main.java` is listed in the Main list, in the bottom-left corner of the `Main.wo` window.

   Drag a connection from the `currentTime` method to the WOString element in the content editor.



Although WOString has several attributes, WebObjects Builder assumes you want to bind the `value` attribute because it's the one most commonly used in WOStrings.

5. Choose a time-display format.



Attribute	Binding
dateformat	"%I:%M:%S %p"
escapeHTML	
formatter	
numberformat	
value	
valueWhenEmpty	

"%x"	Fri Jan 14 2005
"%H:%M:%S"	16:45:12
"%I:%M:%S %p"	04:45:12 PM
"%H:%M"	16:45
"%I:%M %p"	04:45 PM

6. Save `Main.wo`.

The `currentTime` method is now bound to the WOString in the component. This connection is recorded in the WOD file.

HTML and WOD Files

The connection you just made for the WOString element is implemented in the WOD file. You can examine the HTML code and WOD files in Project Builder, within the `Main.wo` subgroup.

In the HTML file, the `<WEBOBJECT>` tag after your static text represents the location where the WOString inserts the value returned by the `currentTime` method. This is a basic behavior; other WebObjects elements offer more capabilities, such as the conditional display of content and repetitions.

```
<BODY BGCOLOR=#FFFFFF>
    The current time is <WEBOBJECT NAME=String1></WEBOBJECT>
</BODY>
```

Notice that the tag reads `<WEBOBJECT NAME=String1>`. The corresponding entry in the WOD file has the same name.

```
String1: WOString {
    value = currentTime;
}
```

The entry has only one listed binding: the connection between the `value` attribute and the `currentTime` method. This method is called whenever the WOString needs to determine the value to display.

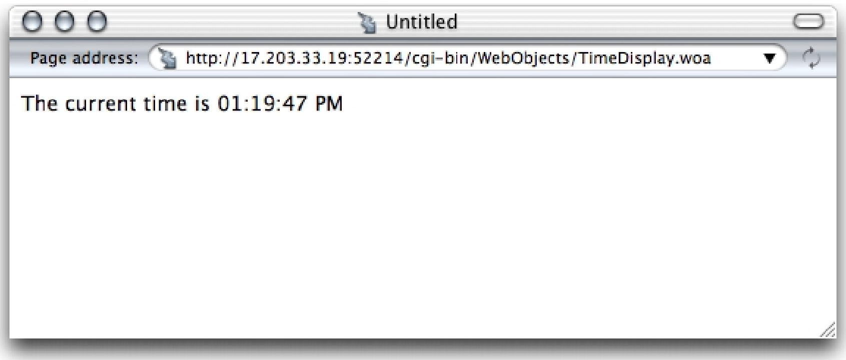
Build and Run the Application

Now that you've customized the Main component, you can run the application and observe the results.

Choose Build > Build and Run or click the Build and Run button in the Project Builder toolbar.

A page similar to the one in Figure 4-2 appears after Project Builder builds and starts your application.

Figure 4-2 Web page displaying the Main component



http://17.203.33.19:52214/cgi-bin/WebObjects/TimeDisplay.woa

The current time is 01:19:47 PM

Response Generation

When you run the TimeDisplay application, the page displayed by your Web browser replaces the WOString you added to the Main component with the current time. If you reload the page, the time changes. WebObjects assembles the page dynamically during the request-response cycle.

When your Web browser requests the URL corresponding to your WebObjects application, your **Web server** hands control to the WebObjects **adaptor** (a process that connects WebObjects application instances to Web servers). This program goes through several steps in generating the response it returns:

1. Reading the HTML file

   Much like a regular Web server, WebObjects first reads an HTML file. Unlike a regular Web server, though, WebObjects parses a `<WEBOBJECT>` tag before returning it to the Web server.

2. Merging the WOD file

   When a `<WEBOBJECT>` tag is encountered, the WOD file for the component is consulted. All the WebObjects tags in an HTML file are named, and each one is listed by its name in the WOD file.

   Each `<WEBOBJECT>` tag represents a WebObjects component. When a `<WEBOBJECT>` tag needs to be evaluated, the entire response generation process is invoked recursively on the new component, continuing as many times as necessary. The Main component is added to your project automatically by the Project Builder Assistant. You can create and use your own components (pages) as you'll see later in "Defining a New Component" (page 89).

3. Invoking methods

   Each type of WebObjects component has special logic for constructing the HTML code to return to the Web server. Customization of this process is done with attributes defined by the component's developer. Each binding in a WOD file can be either static or dynamic. If a binding is static, the value supplied is used directly.

   If a binding is dynamic—that is, an attribute is bound to a method or instance variable—WebObjects invokes the method or accesses the instance variable to obtain the value at runtime. In the example above, when the WOString is evaluated, it invokes the method named in its `value` binding (`currentTime`) to get the value to display. The implementation of WOString turns the NSTimestamp object into a string and displays it in your Web browser.

This process takes place each time your Web browser requests the Main component. If you reload the page, the method is invoked again and a new time value is displayed.

For more information on the request-response loop, see "Request Processing" (page 62).

Maintaining State in the Component

Understanding the connection between a component and its class file is an important part of WebObjects development. Not only do you associate methods with the component to create dynamic content in this fashion, but you can also use the methods provided by the WOComponent class to maintain state for a component.

When you add methods to a component in WebObjects Builder, you are actually editing the component's Java file. When you modify how the component looks or add display elements, you are editing HTML code. A WebObjects component is a high-level view of both the HTML code and the Java class that describe a Web page, or part of one. After using WebObjects Builder to define the major parts of a component, you can add details by editing the HTML code directly and by modifying its Java file.

When your application runs, components are instantiated as needed. That is, each component is also an object in your application. For example, when the TimeDisplay application launches, a Main object is created. As the component's content is determined by WebObjects, methods in `Main.java` are used to provide the data for its dynamic elements, in this case, the WOString that displays the current time. When it's time for WebObjects to add the content for the WOString, it looks up the element's `value` binding. In the example, `value` is bound to the `currentTime` method. WebObjects then invokes the `currentTime` method, which returns the current time.

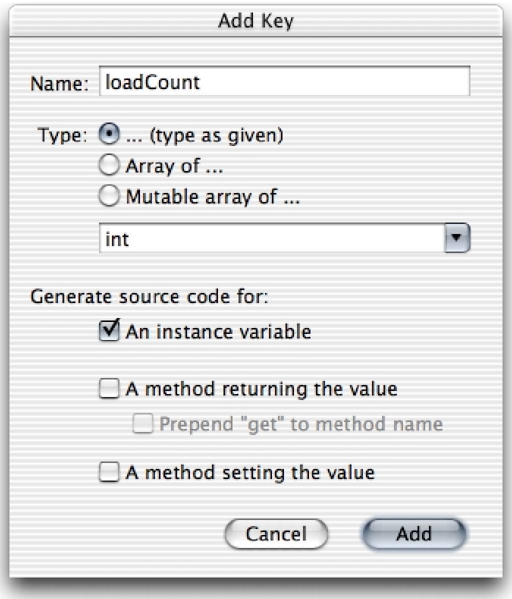
An instance of a component "survives" at least for two cycles of the request-response loop: in the first cycle the page is rendered while in the second cycle the component determines which component to display next. If the component to be displayed is different from the first one, the latter is discarded while an instance of the new component is created. However, if the component to display is the same one, then the instance "lives on." You can use instance variables in your component's class to store information and keep track of the user's behavior as she interacts with your application.

In this case, you'll add a variable to the Main component and add code to increment it each time the page is displayed. You can use this variable to show the number of times the page has been loaded by a specific user in the **session**, which is an object that represents a conversation between your application and a particular client.

To keep track of the number of times the `currentTime` method is invoked, you need to add an integer instance variable to the `Main.java` file, increment it each time the page is loaded, and add a means of telling the page to refresh itself.

Adding the Variable to Count Method Calls

1. Open `Main.wo` in WebObjects Builder (if it's not already open) by double-clicking it in Project Builder's main window.
2. Choose Add Key from the Edit Source menu at the bottom-left corner of the `Main.wo` window.
3. Add a key of type `int` named `loadCount`.


Add Key

Name: loadCount

Type: [✓ ... (type as given)] Array of ... Mutable array of ...

int

Generate source code for:

[✓ An instance variable]

A method returning the value

Prepend "get" to method name

A method setting the value

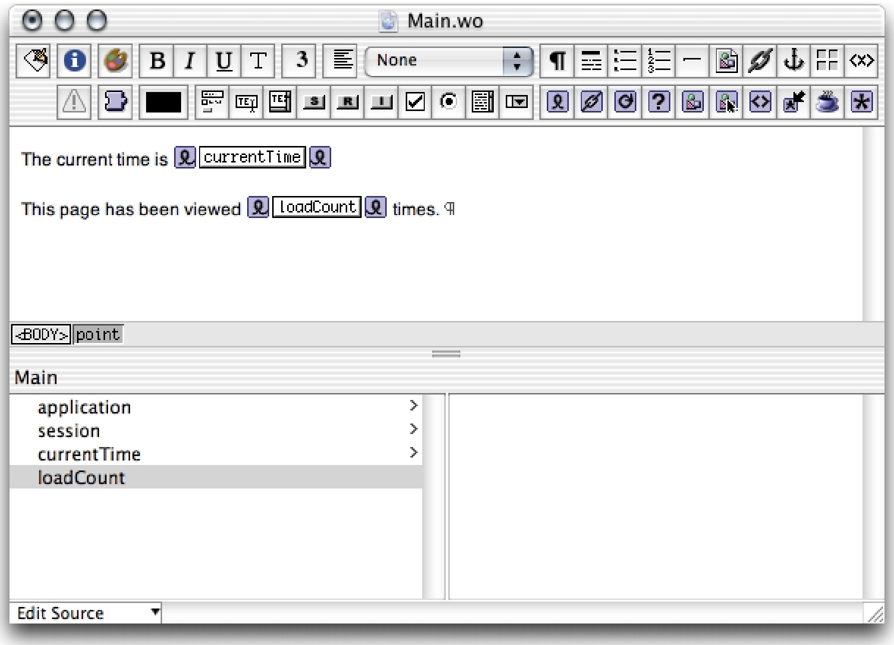
4. Examine the Java file in Project Builder to confirm that the variable was added.

```
public class Main extends WOComponent {
    protected int loadCount;
```

Displaying the Count

To display the load count in the component, you need to add another WOString to the component.

1. Add a label and a WOString to the component.
   a. Enter `This page has been viewed` below the line that displays the current time.
   b. Add a space and a WOString to the right of the label.
   c. Add a space and `times.` to the right of the WOString.
2. Bind the `loadCount` variable to the new WOString's `value` attribute.



The current time is currentTime

This page has been viewed loadCount times.

Increasing the Variable's Value

Modify the `currentTime` method so it increments the `loadCount` variable each time it is called. Since WebObjects calls the method each time the page needs to be displayed, `loadCount` is increased by one each time.

```
/**
 * Increases loadCount and creates an NSTimestamp object
 * with the current date and time.
 *
 * @return current date and time.
 */
public NSTimestamp currentTime() {
    loadCount++;
    return new NSTimestamp();
}
```

Refreshing the Page

Finally, you need to add a way to reload the page. In WebObjects, regular hyperlinks (WOHyperlinks) can call Java methods on your components. Action methods are covered in greater detail in "Request Processing" (page 62). For now, you need only to add a method that simply reloads the current page.

1. Add the action method.

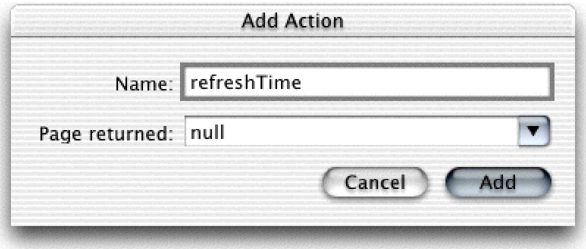
   Open the Main component in WebObjects Builder and choose Add Action from the Edit Source menu.

   a. Name the action `refreshTime`.

   b. Select `null` from the "Page returned" pop-up menu.

      The value returned by an action method represents the next page (component) to be displayed. When you return `null`, the current page is redrawn. In a later task, you learn how to return a new component.

   c. Click Add.



Name: refreshTime

Page returned: null

2. Add a hyperlink.

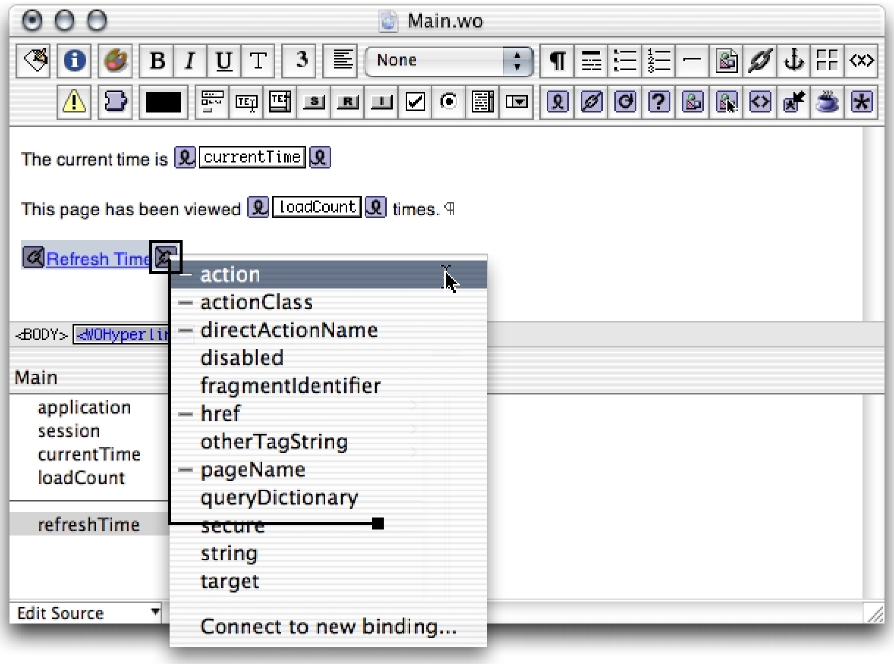
   Position the cursor below the line where the load count is displayed.

   Choose WebObjects > WOHyperlink, or click Add WOHyperlink on the toolbar.

   By default, the text for a new link is `Hyperlink`. You can replace this by selecting the text and typing something more appropriate over it, such as `Refresh Time`.

3. Connect the `refreshTime` method to the WOHyperlink.

   Much like a WOString, a WOHyperlink has several attributes. In this case, you bind the `refreshTime` method to the `action` attribute of the WOHyperlink.

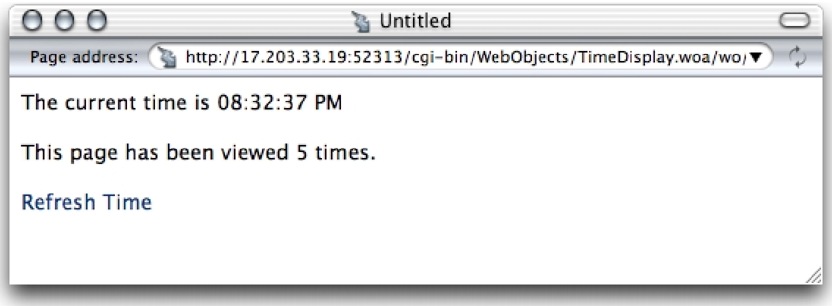


Drag from the `refreshTime` method in the Main list to the WOHyperlink. When you release the mouse button, you will see a pop-up list of attributes. Choose the `action` attribute to indicate that you want the `refreshTime` method called when the link is clicked.

4. Save `Main.wo`.

The Counter in Action

Build and run the TimeDisplay application. When your browser loads the page, you'll see that the counter has been increased to 1. If you click Refresh Time, the time and the load count are updated.



http://17.203.33.19:52313/cgi-bin/WebObjects/TimeDisplay.woa/wo/

The current time is 08:32:37 PM

This page has been viewed 5 times.

Refresh Time

The same counter instance variable is increased by one each time you use the link because WebObjects creates a Main object and associates it with your Web browser window through a Session object. Each time you interact with the application, by clicking Refresh Time, the same object is used. If you open another browser window and connect to the application again using the URL shown in Project Builder's Run pane, a separate instance of Main is created and associated with that window. From then on you can work with both windows individually. As a matter of fact, not only is a new instance of Main created, a new Session object is created as well.

WebObjects determines that a new Session object needs to be created when the incoming URL does not contain a session ID. The first time you connect to the application using a URL like the one in Listing 4-1, WebObjects creates a Session object and assigns it a session ID and other information. That information is added to the URL returned to your browser together with the Web page to be displayed (see Listing 4-2). When you send another request from your browser (by clicking Refresh Time, for example) WebObjects uses the session ID encoded in the URL to locate the Session object that is to process the request. This is the default mechanism WebObjects uses to keep track of the state of each user. For more on state management see "Client-Server Applications" (page 23) and "Managing State" (page 99).

Listing 4-1 URL that generates a new Session object

```
http://foo.com:49361/cgi-bin/WebObjects/DateDisplay
```

Listing 4-2 URL with session ID

```
http://foo.com:49361/cgi-bin/WebObjects/DateDisplay.woa/wo/
whcV5sauLNtG8Tfh6xCuvM/0.1
```

Further Exploration

You've learned how to use some of WebObjects's tools, and how to add elements and bind them to your Java code using WebObjects Builder. You also learned how to display dynamic content based on Java code, and maintain state data from one request to the next. Feel free to explore WebObjects Builder to learn more. Here are a few suggested exercises:

- All the usual attributes of a Web page—title, background color, font size, and the like—can be maintained in WebObjects Builder. Make the TimeDisplay application a bit smoother around the edges by setting the page title and customizing the text displayed. If a WOString is inside another HTML tag, the WOString is affected just like ordinary text.
- What happens if the WOString that displays the value of the `loadCount` instance variable is placed before the WOString that displays the time (and updates `loadCount`)? WebObjects parses the WOStrings in the order in which they appear, so `loadCount` is `0` the first time it is displayed.

CHAPTER 5

Managing User Input

WebObjects's ability to dynamically display information is sufficient for some Web applications, but most require more complex interaction with the user.

WebObjects provides a system for associating display and user input elements on a Web page with your variables and methods. You've seen how easy it is to display your dynamic data in Web pages.

In this chapter, you

- learn the system WebObjects uses to take in user input
- take input from the user via form elements like WOForm and WOTextField
- use WOConditionals for the conditional display of elements
- learn to construct derived properties with custom logic

User input in WebObjects is based on the basic HTML input elements—forms, text input fields, and so on. Connecting these elements to variables and methods is very similar to the process used to bind the `value` attribute of WOStrings (see "Adding a WOString" (page 47) for more information).

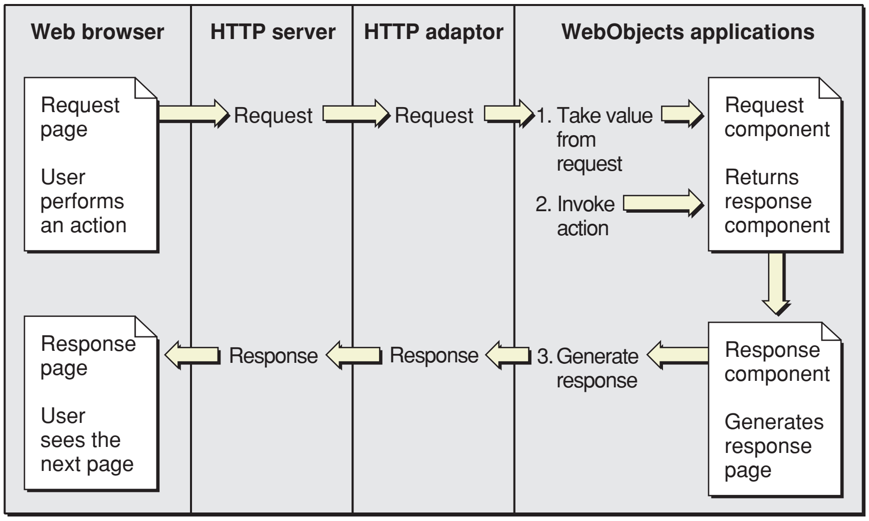
You place components that mirror HTML form elements into your components. These components use your Java code to generate HTML code that Web browsers can interpret and display, and are programmed to translate user-entered data or selections back into variables. The system by which values are taken from these elements and communicated to your code is called request processing.

Request Processing

Each action taken by a user is communicated to your application via the Web server and the WebObjects adaptor. All the pertinent details of the user's action—the contents of text fields, the state of radio buttons and checkboxes, and the selections in pop-up menus—as well as information about the session and button or link activated is encoded in the HTTP (Hypertext Transfer Protocol) request.

The request is decoded by the action of the WebObjects adaptor and default behaviors in the application. This decoding process, which culminates in the generation of a response page to be returned to the Web browser, is called the request-response loop. See Figure 5-1.

Figure 5-1 The request-response loop

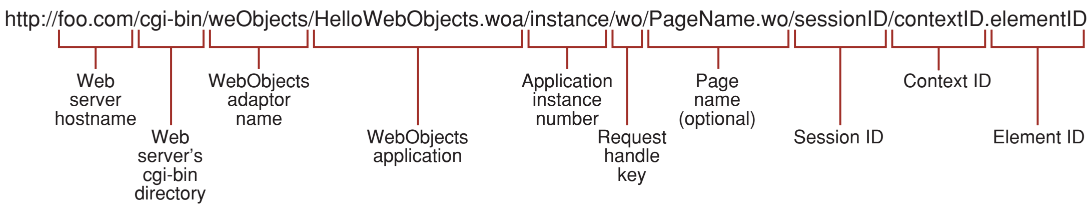


WebObjects has two request-processing models: component actions and direct actions.

- **Component actions.** This model enables you to maintain state in your applications; therefore, it requires and uses sessions. By default, WebObjects applications use this model and it's the one explained in this chapter.
- **Direct actions.** This model is used by applications that don't require state management (such as search engines, product catalogs, document libraries, and dynamic publishing). Applications that use this model don't have sessions by default.

As Figure 5-2 shows, a component action URL contains all the information necessary for WebObjects to reconstruct the state the session and components were in when a page was last generated for a given client. Listing 5-1 shows an example of a component action URL.

Figure 5-2 Structure of a component action URL


http://foo.com/cgi-bin/weObjects/HelloWebObjects.woa/instance/wo/PageName.wo/sessionID/contextID.elementID
Web server hostname
Web server's cgi-bin directory
WebObjects adaptor name
WebObjects application
Application instance number
Request handle key
Page name (optional)
Session ID
Context ID
Element ID

Listing 5-1 Example of a component action URL

```
http://foo.com:49663/cgi-bin/WebObjects/DateDisplay.woa/wo/
NDdW3uF2xRVjvbXUgRCVM/0.5
```

Table 5-1 shows a summary explanation of the phases of the request-response process. Table 5-2 (page 65) shows the order in which the methods involved are invoked. The process is explained in detail in "Processing the Request" (page 65) and "Generating the Response" (page 67).

Table 5-1 Request-response processing phases

Phase	Method	Description
Awake	`public void awake()`	The Application, Session, and Component objects are awakened. (Custom initialization logic can be added.)
Sync	`public void takeValuesFromRequest (WORequest, WOContext)`	Form data is read into the instance variables the WebObjects elements are bound to. (Setter methods are used.)
Action	`public WOActionResults invokeAction (WORequest, WOContext)`	The action the user triggered (with a link or a submit button) is performed. The action could create a new page.
Response	`public void appendToResponse (WOResponse, WOContext)`	The response page is generated. The form elements' contents are set to the values stored in the instance variables the WebObjects elements are bound to. (Getter methods are used.)
Sleep	`public void sleep()`	The Application, Session, and Component objects are put to sleep. (Custom deactivation logic can be added.)

Table 5-2 Request-response processing time line

Application	Session	Component
`awake`		
	`awake`	
		`awake`
`takeValuesFromRequest`		
	`takeValuesFromRequest`	
		`takeValuesFromRequest`
		Setter methods invoked.
`invokeAction`		
	`invokeAction`	
		`invokeAction`
`appendToResponse`		
	`appendToResponse`	
		`appendToResponse`
		Getter methods invoked. Response page generated.
		`sleep`
	`sleep`	
`sleep`		

Processing the Request

Request processing takes place in three stages: awakening, state synchronization, and action invocation.

- **Awake.** This stage is carried out when WebObjects invokes the `awake` method.

  In a multi-user system, limited resources need to be used as efficiently as possible. To this end, applications are active only while they perform a task. A single server can be running several different applications or many instances of the same application. WebObjects keeps applications asleep while they are not participating in the request-response loop. See "Generating the Response" (page 67) for more information.

The application object's `awake` method is invoked first, then the session's, and finally the component's. You can customize this method in each of the classes involved to provide logic that needs to be performed before processing the request. Although the default implementations of those methods do nothing, you should call the superclass's method before executing custom logic, as Listing 5-2 shows.

Listing 5-2 Overriding the `awake` method

```
public void awake() {
    super.awake();

    /* Custom logic goes here. */
}
```

- **Sync.** During this stage, the `takeValuesFromRequest` method is invoked, which causes the values entered in form elements by the user to be copied into the corresponding instance variables. If the component contains no form elements or if the values of the form elements were not changed, this stage is not performed.

  WebObjects invokes the application's `takeValuesFromRequest` method. The application then invokes the session's method, which in turn invokes the component's method. The component invokes each dynamic element's `takeValuesFromRequest` method, which causes form elements to copy the values from the request into the appropriate component bindings. WebObjects uses the NSKeyValueCoding interface to determine how to set the value of the binding.

  To set the value of a key named `key`, WebObjects looks for an available setter method or an instance variable in the following order:

  1. `public void setKey()`
  2. `private _setKey()`
  3. `_key`
  4. `key`

- **Action.** This is where the `invokeAction` method is invoked; the action the user chose is executed.

  Like the `takeValuesFromRequest` method, WebObjects invokes the application's `invokeAction` method. The application then invokes the session's method, which in turn invokes the component's method. The component then invokes the method on each of its dynamic elements.

  When the `invokeAction` method of the dynamic element that triggered the request is invoked (a submit button, for example), the dynamic element invokes the method bound to its `action` attribute.

Generating the Response

After the form values are gathered and the action method is invoked, the application creates a response page. This is the component returned by the action method. The response-generation process has two phases: append to response and sleep.

- **Response.** Here is where the response page is generated. Each WebObjects element's `appendToResponse` method is invoked, so that it can add its content to the page to be displayed.

  WebObjects invokes the application's `appendToResponse` method. Then the application invokes the session's method, which in turn invokes the component's method. The component goes through its HTML code creating the page's content. When it finds a `<WEBOBJECT>` tag, it invokes the corresponding element's `appendToResponse` method, so that it can get the values of its binding and add the resulting content to the page. The process continues recursively until the entire response page has been created.

  When a variable needs to be evaluated, WebObjects uses a system similar to the one it uses when a variable needs to be set. When the value of a key named `key` is requested, WebObjects first looks for a getter method. If one is not found, it accesses the instance variable itself. The order in which WebObjects tries to obtain the value for `key` is as follows:

  1. `public [...] getKey()`
  2. `public [...] key()`
  3. `private [...] _getKey()`
  4. `private [...] key()`

5. `[...] _key`
6. `[...] key`

- **Sleep.** When the response process is completed, the `sleep` methods of the Component, Session, and Application objects are invoked. (The order in which the objects' `sleep` method is called is the opposite of the order in which the `awake` methods are invoked in the awake phase.) When overriding the `sleep` method, you should follow the structure in Listing 5-3.

Listing 5-3 Overriding the `sleep` method

```
public void sleep() {
    /* Custom logic goes here. */

    super.sleep();
}
```

After all the objects involved in the request-response process are put to sleep, the new page is sent to the WebObjects adaptor.

Backtracking Cache

WebObjects supports the use of a Web browser's Back button (backtracking) by keeping a cache of recently viewed pages on the server. The cache is configured to hold 30 pages per session, but you can customize it to meet your needs. To change the default size of the cache, add code to the Application class's constructor. For example, to change the page cache size to 45 pages, you add this code line:

```
setPageCacheSize(45);
```

When a response page is generated, it and its state information are added to the cache. That way, when the user clicks her browser's Back button, WebObjects can retrieve the correct component and its state.

For backtracking to work properly with dynamic data, a Web browser's own cache must be disabled, so that all page requests go to the Web server and, therefore, your application. You can accomplish this by adding this code line to the Application class's constructor:

```
setPageRefreshOnBacktrackEnabled(true);
```

When the cache becomes full, the oldest page in it is discarded to make room to store a new page. When the user backtracks past the oldest page in the cache, WebObjects informs her of the situation with a special page.

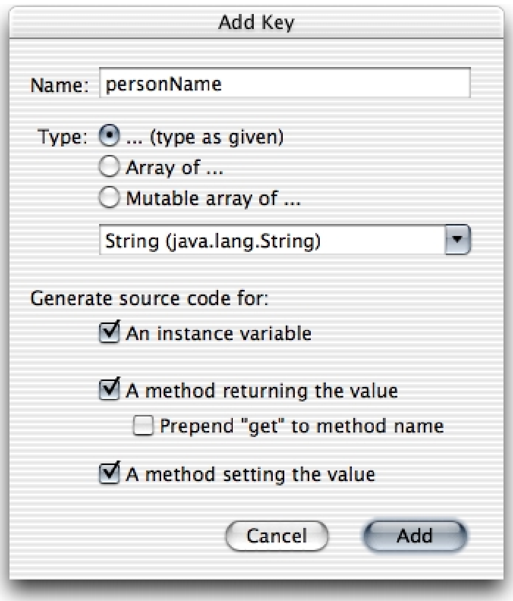
User Interface

Input elements are bound to variables in a way very similar to the way display elements are. In fact, input elements are essentially bidirectional display elements—they get a value from the object when the response is generated and send a value back to the object when a request is received. See "Request Processing" (page 62) for more information.

For this example, you display a bit of information about the user. You'll use text input fields to get data from the user, and once she's entered it, you'll use a WOConditional to hide the text fields and display the data. Then you'll encapsulate the user data into a custom object so you can generate an array of them.

First, create a new project named UserEntry. Edit the Main component with WebObjects Builder. The first step is to add variables for the data the user enters. Then, you add WOTextFields and bind them to the variables.

1. Add two variables named `personName` and `favoriteFood` to the Main component using the Edit Source menu. These variables should be of type `java.lang.String`. Make sure the three options below "Generate source code for" are selected so that an instance variable and accessor methods are generated.


Add Key

Name: personName

Type: [✓ ... (type as given)] Array of ... Mutable array of ...

String (java.lang.String)

Generate source code for:

[✓ An instance variable]

[✓ A method returning the value]

Prepend "get" to method name

[✓ A method setting the value]

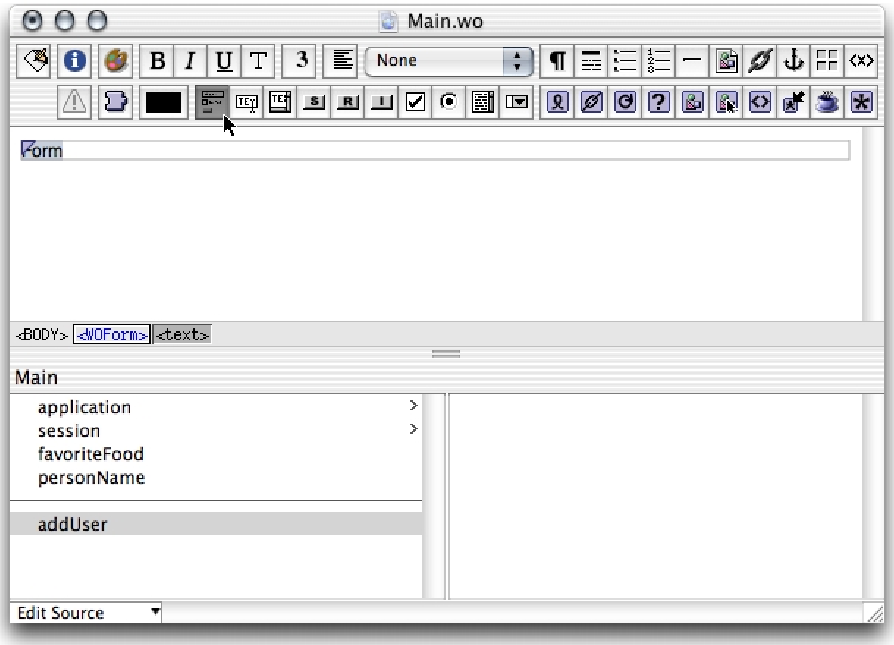
Note: Avoid calling a variable `name`. This name is used by WebObjects and using it for your own purposes will lead to unexpected results.

2. Use the Edit Source menu to add an action method named `addUser`. Accept the default of `null` for the component's return value.

   In a later step, you'll customize this method to set additional variables.

3. Add a WOForm, labels, WOTextFields, and a WOSubmitButton to capture data from the user.

   a. Add the WOForm by choosing Forms > WOForm or clicking the Add WOForm button in the toolbar.



Note: All form elements, including submit buttons, must be within a WOForm to function.

b. Add the WOForm's elements.

   Add two labels `Name:` and `Favorite Food:` in separate lines.

   Add a WOTextField next to the Name label by choosing Forms > WOTextField or clicking the Add WOTextField button in the toolbar.

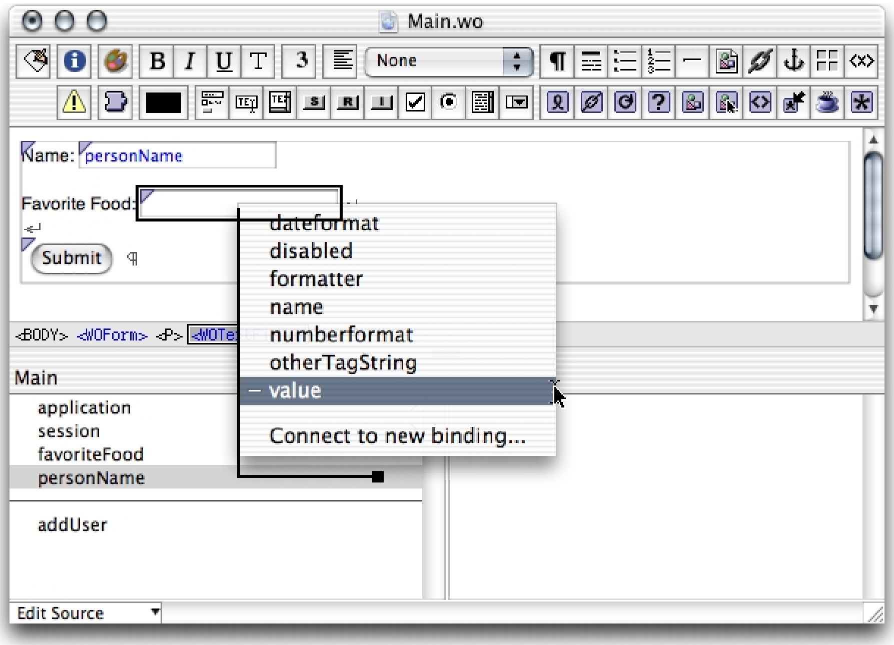
   Add a second WOTextField next to the Favorite Food label.

   Place the cursor at the end of last text field and press Shift-Enter two times.

   Choose Forms > WOSubmitButton to add a button to use to submit the form.

4. Bind the variables to the `value` attribute of the appropriate text fields, just as with the WOString. See Figure 5-3.

Figure 5-3 Binding the Favorite Food text field to personName



5. Bind the `addUser` action to the `action` attribute of the WOSubmitButton.
6. Save `Main.wo`.

All the user interface elements are connected. The WOTextFields set the properties bound to them during request processing. The Java method bound to the WOSubmitButton's `action` attribute is called when the user clicks Submit.

Tracing the Request-Response Loop

Now you'll modify the methods in your Java files to display a message indicating when they're called, so you can watch the phases of the request-response loop in action.

Each time the user submits a request, the contents of the text fields are sent with the request. WebObjects then determines the properties to update and the methods to invoke using the WOD file.

Selecting the options under "Generate source code for" when you added the `favoriteFood` and `personName` keys to the Main component caused WebObjects Builder to insert not just two String variables, but also two methods that are used to update those variables (the accessor methods, a getter method, and a setter method). If you add printing statements to those methods and to the `addPerson` action method, you can watch them being invoked during the request part of the request-response loop. If you add the other methods described in "Request Processing" (page 62), you can watch them being called as you use the application, as well.

Edit the `Session.java`, `Application.java`, and `Main.java` files to add the `awake` method so you can track the processing of the request. You can use the `System.out.println` method to log text to the console of your application; it is then displayed in the Run pane of Project Builder's main window. Add the method in Listing 5-4 to all three files.

Listing 5-4 Tracing the request-response loop—the `awake` method

```
/**
 * Prints a line to the console when invoked.
 */
public void awake() {
    super.awake();
    System.out.println(this.getClass().getName() + "‘s awake invoked.");
}
```

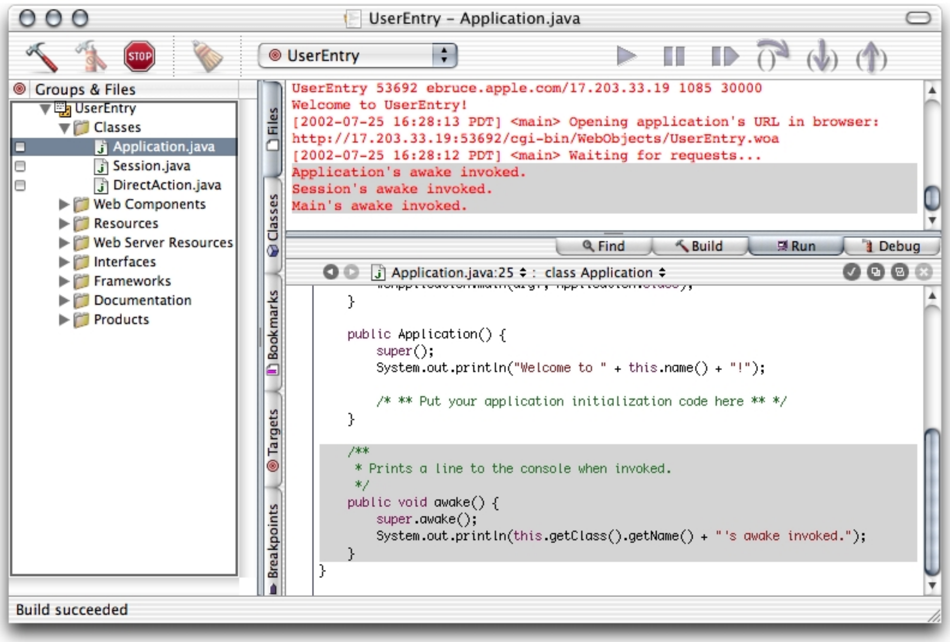
This method prints the name of the class followed by a notification that the `awake` method was called in each class that you put it in. Notice that it calls `super.awake` to ensure that the superclass's `awake` method is called before executing its custom logic.

Edit the `setPersonName`, `setFavoriteFood`, and `addUser` methods in `Main.java` to log strings to the console when they are invoked. Your methods should look like the ones in Listing 5-5.

Listing 5-5 Tracing the request-response loop—the accessor and action methods

```
public void setPersonName(String newPersonName) {
    System.out.println("Setting personName to " + newPersonName);
    personName = newPersonName;
}

public void setFavoriteFood(String newFavoriteFood) {
    System.out.println("Setting favoriteFood to " + newFavoriteFood);
    favoriteFood = newFavoriteFood;
}

public WOComponent addUser() {
    System.out.println("Submit clicked.");
    return null;
}
```

Build and run the new application, correcting any errors revealed during compilation if necessary.



```
UserEntry 53692 ebruce.apple.com/17.203.33.19 1085 30000
Welcome to UserEntry!
[2002-07-25 16:28:13 PDT] <main> Opening application's URL in browser:
http://17.203.33.19:53692/cgi-bin/WebObjects/UserEntry.woa
[2002-07-25 16:28:12 PDT] <main> Waiting for requests...
Application's awake invoked.
Session's awake invoked.
Main's awake invoked.
```

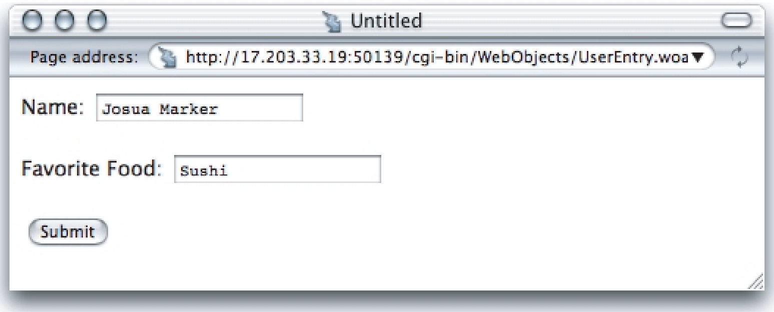
```
    }

    public Application() {
        super();
        System.out.println("Welcome to " + this.name() + "!");

        /* ** Put your application initialization code here ** */
    }

    /**
     * Prints a line to the console when invoked.
     */
    public void awake() {
        super.awake();
        System.out.println(this.getClass().getName() + "'s awake invoked.");
    }
}
```

The Run pane shows the following output:

```
Application's awake invoked.
Session's awake invoked.
Main's awake invoked.
```

Notice that when the page first loads, the `awake` methods are called. This is because the request-response loop is run through the first time the page is generated. Also notice that your `set` methods are not called. This is because at the time of the first request the user has not yet filled in any text fields, so the state synchronization phase does not take place (See "Processing the Request" (page 65)).

Fill in the data fields and click Submit.

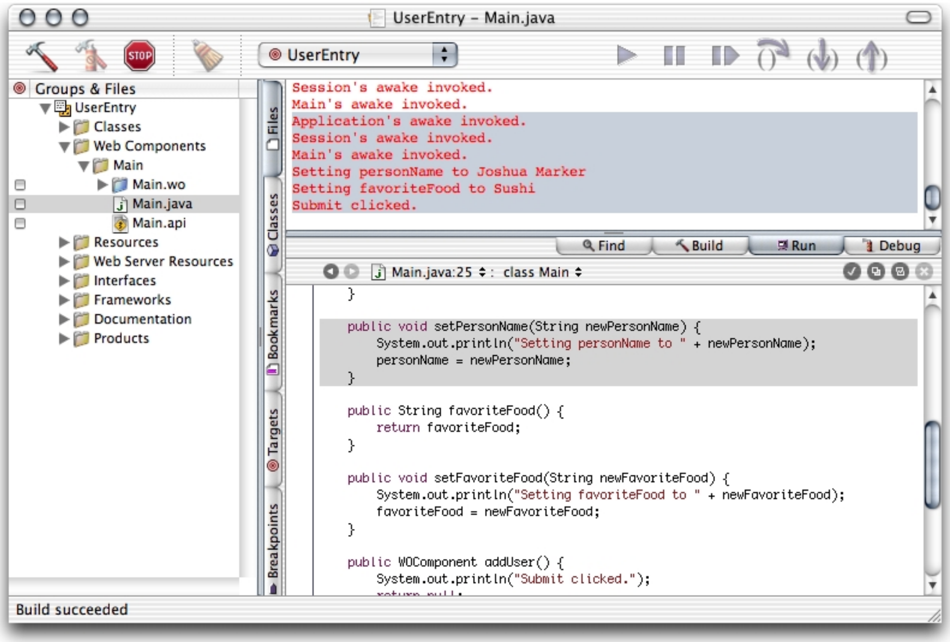


Page address: http://17.203.33.19:50139/cgi-bin/WebObjects/UserEntry.woa

Name: Josua Marker

Favorite Food: Sushi

When you click Submit, you'll be able to watch the request portion of the request-response loop through Project Builder's Run pane as variables are updated.



```
Session's awake invoked.
Main's awake invoked.
Application's awake invoked.
Session's awake invoked.
Main's awake invoked.
Setting personName to Joshua Marker
Setting favoriteFood to Sushi
Submit clicked.
```

```
    }

    public void setPersonName(String newPersonName) {
        System.out.println("Setting personName to " + newPersonName);
        personName = newPersonName;
    }

    public String favoriteFood() {
        return favoriteFood;
    }

    public void setFavoriteFood(String newFavoriteFood) {
        System.out.println("Setting favoriteFood to " + newFavoriteFood);
        favoriteFood = newFavoriteFood;
    }

    public WOComponent addUser() {
        System.out.println("Submit clicked.");
```

This is the output that shows on the Run pane when you click Submit:

```
Application’s awake invoked.
Session’s awake invoked.
Main’s awake invoked.
Setting personName to Joshua Marker
Setting favoriteFood to Sushi
Submit clicked.
```

Because of the `printLn` method invocations added, you can see that WebObjects invokes each `awake` method, performs variable assignment, and then invokes the action method you assigned to the submit button. This means that if code in the `addUser` method referred to the `favoriteFood` or `personName` variables, the values provided by the user would be available, rather than old values, if any. You can take advantage of this to set other variables in your component. For example, currently the form fields remain active even after the user has filled them in. You could make the fields disappear once the necessary data has been entered by checking in the `addUser` method to see if both fields are filled in and setting a Boolean property to indicate whether the entry is still incomplete. You could then use a WOConditional element to hide some elements of the component.

Conditional Display With WOConditional Elements

A WOConditional element provides a means of displaying part of a component only when a particular requirement is met. This part could include text, elements, and other components.

The WOConditional element has two attributes: `condition` and `negate`. The `condition` attribute is required. While it is syntactically correct to use the values `YES` or `NO` for this binding, the element is useful only when `condition` is bound to a method that returns `true` or `false` (you can also bind it to integer objects, in which case nonzero values are interpreted as `true` and zero values as `false`). If the method evaluates to `true`, the contents of the conditional are displayed; otherwise, they are not. If the `negate` attribute it set to `true`, this arrangement is reversed: the contents are displayed only when the `condition` attribute evaluates to `false`.

You can use a pair of WOConditionals to ask the user for input and then display the information she entered. This is the method described in the remainder of this chapter to capture and display user data.

1. Add an instance variable you can use to indicate whether the user has entered the necessary information.

   Add the following variable to `Main.java`:

   ```
   protected boolean entryIncomplete;
   ```

   You can use the WebObjects Builder Edit Source menu or add the variable directly to the class file. (If you use WebObjects Builder, be sure to deselect the options under “Generate source code for” in the Add Key dialog.)

   This variable should be initialized to `true` because the variables are empty when the page is first displayed, so the entry is incomplete. Otherwise, the fields would not be displayed the first time the page is shown. Initialize the variable in the component’s constructor:

   ```
   public Main(WOContext context) {
       super(context);
       entryIncomplete = true;
   }
   ```

   Also modify the `addUser` method to check the form properties and update the value of `entryIncomplete`:

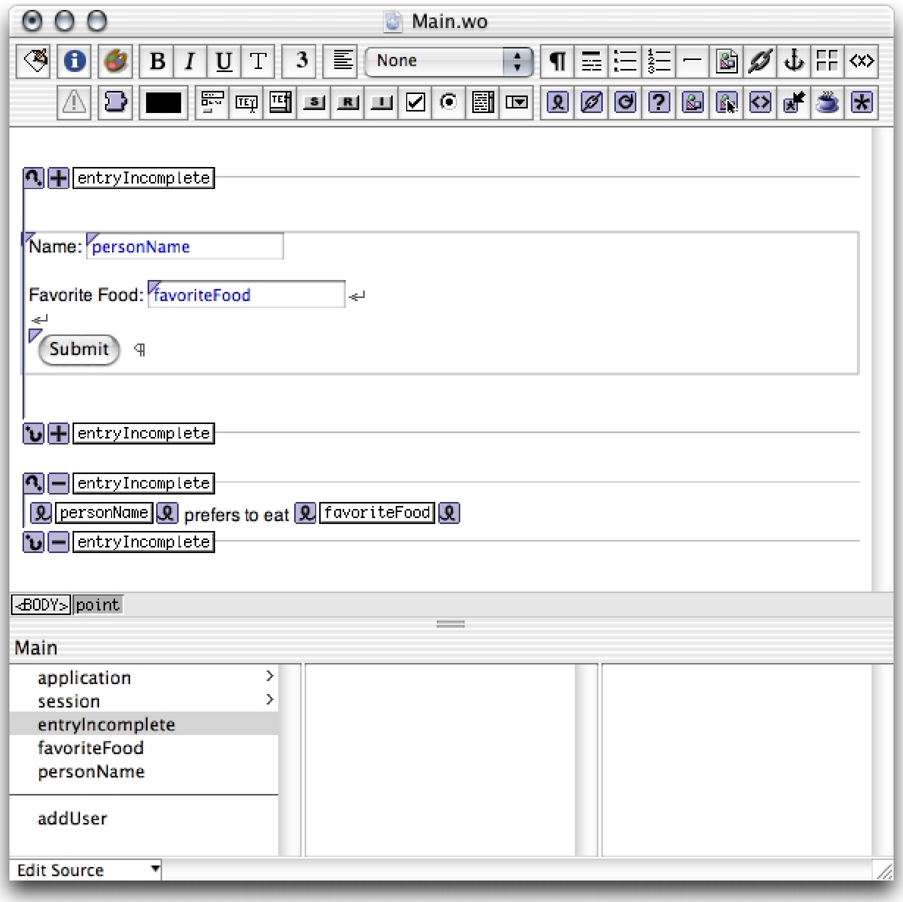
   ```
   public WOComponent addUser() {
       System.out.println("Submit clicked.");
       if (personName.equals("") || favoriteFood.equals("")) {
           entryIncomplete = true;
       }
       else {
           entryIncomplete = false;
       }
       return null;
   }
   ```

2. Save `Main.java`.
3. Open `Main.wo` in WebObjects Builder
4. Make the form element conditional by wrapping it in a WOConditional.

   The fields and the submit button should be displayed only while `entryIncomplete` is `true`. Select the form and choose WOConditional from the WebObjects menu or click the Add WOConditional button in the toolbar. (You can select the form by clicking inside it and then clicking the `<WOForm>` tag in the path pane, located below the content editor.)
5. Bind the `condition` attribute of the WOConditional to the `entryIncomplete` instance variable.

   As long as `entryIncomplete` evaluates to `true`, WebObjects displays the WOConditional's content.
6. Create elements to display the data once it has been entered.
   a. Make a new line below the WOConditional.
   b. Add two WOStrings.
   c. Add the text " `prefers to eat` " between the WOStrings (note the leading and trailing spaces).
   d. Bind the first WOString's `value` attribute to `personName`, and the second's to `favoriteFood`.
7. Select the new items and create a WOConditional around them.
8. Bind the new WOConditional's `condition` to `entryIncomplete`. Click "+" in the WOConditional to invert its meaning. It changes to a "-". When the component is rendered in a Web browser, the contents of the second WOConditional are displayed only when the value of the `entryIncomplete` variable is `false`.

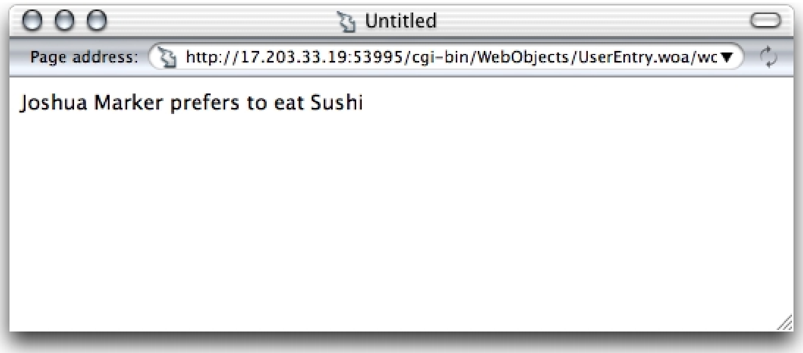
Figure 5-4 WOConditional elements



9. Build and run the application.

The first time the Main component is generated, you see the same page as the last version of the application, because `entryIncomplete` is `true` and the contents of the first WOConditional are displayed.

Once the user enters data and clicks the submit button, the `addUser` method determines if she entered text in both text fields and, if you so, sets `entryIncomplete` to `false`. Since the `addUser` method returns `null`, the page is redrawn with the new variable settings, and this time the contents of the other WOConditional are displayed because the variable changed.


Untitled
Page address: http://17.203.33.19:53995/cgi-bin/WebObjects/UserEntry.woa/wc

Joshua Marker prefers to eat Sushi

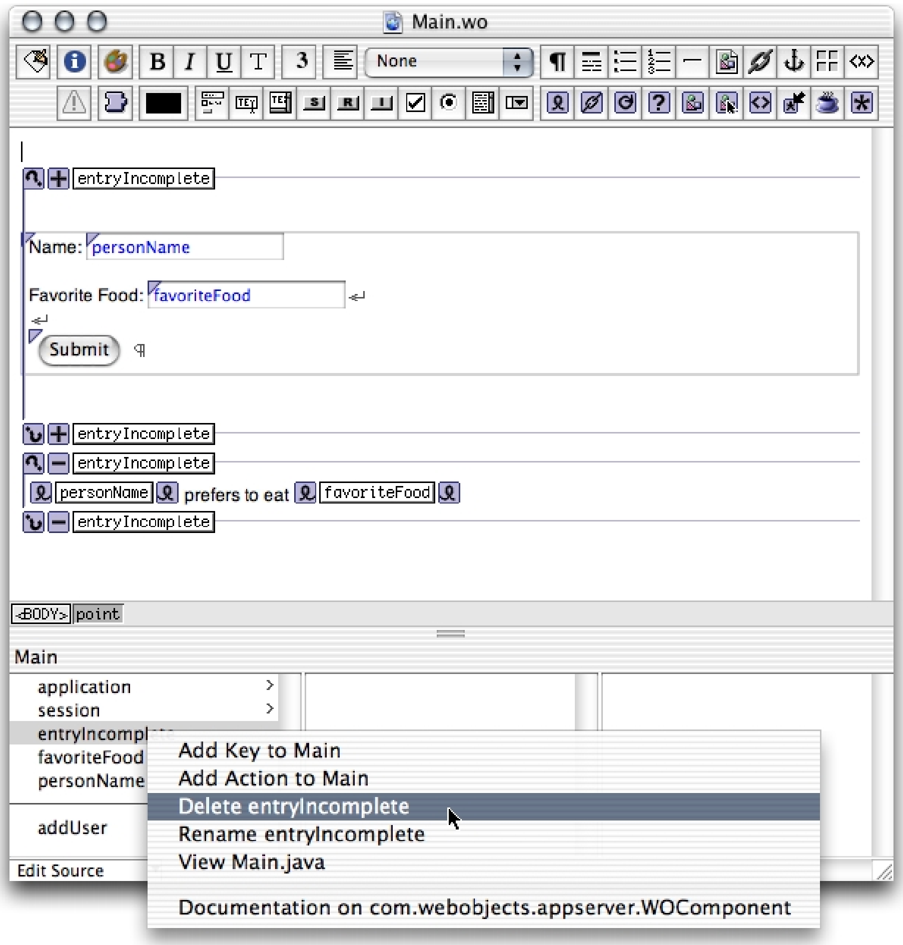
Derived Properties

Instead of using an instance variable to determine when the entry is complete, you could provide logic in a method that evaluates the entry at the moment the method is invoked and decides whether it is complete. By doing this, you can completely remove the variable.

This kind of property is called a *derived property*, because while it is a property of the object, it is not directly stored but is instead derived via logic. You can remove the `entryIncomplete` variable and replace it with an `entryIncomplete` method without changing the WOD file or altering `Main.wo`.

1. Remove the `entryIncomplete` variable from your `Main.java` file.

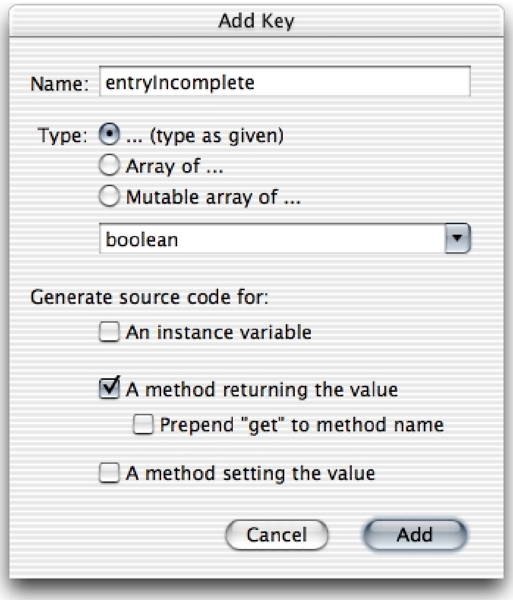
   Remove the variable declaration as well as the assignments in the constructor and in the `addUser` method. You can remove the variable itself using WebObjects Builder by Control-clicking the variable name and choosing "Delete `entryIncomplete`". To remove the other code you must edit the Java file in Project Builder.



2. Add a method called `entryIncomplete` that provides the same information the variable did.

   Add the property by choosing Add Key from the Edit Source menu in WebObjects Builder's main window. In the Add Key dialog, deselect the option to generate an instance variable. WebObjects automatically selects the other two options. Deselect the option to generate a method for setting the value, leaving only the option to generate a method for returning the value, as shown in Figure 5-5.

Figure 5-5 Adding a derived property



Name: entryIncomplete

Type: [✓ ... (type as given)] Array of ... Mutable array of ...

boolean

Generate source code for:

☐ An instance variable

☑ A method returning the value

☐ Prepend "get" to method name

☐ A method setting the value

Note: If you unbound `entryIncomplete` key from the two WOConditionals when you deleted the variable, you must bind the `entryIncomplete` method to the WOConditionals before continuing.

Modify the `entryIncomplete` method so that it looks like the one shown in Listing 5-6.

Listing 5-6 Implementation of `entryIncomplete` as a derived property

```
public boolean entryIncomplete() {
    boolean entryIncomplete;
    if (personName == null || favoriteFood == null ||
personName.equals("") || favoriteFood.equals(""))  {
        System.out.println("The entry is incomplete.");
        entryIncomplete = true;
    }
    else {
        System.out.println("The entry is complete.");
        entryIncomplete = false;
    }
    return entryIncomplete;
}
```

3. Build and run the application.

   Notice that the application runs just as before; you can now see that WebObjects requests the `entryIncomplete` method twice in the course of displaying the page—once while evaluating each WOConditional.

   You can use this approach to derive the value of a property when it is needed rather than storing it in a variable.

CHAPTER 6

Component Communication

One of the aspects of the WebObjects strategy is the ability to define new components and share data between them.

As was said before, WebObjects is a heavily object-oriented system. It provides for easy encapsulation of data into components and custom classes, and facilitates the sharing of data between components when an application is run.

In this chapter, you

- encapsulate data into a custom class
- learn how WebObjects follows keypaths
- add a new WOComponent to your application
- programmatically create new components and return them to the user
- pass information between components

Custom Objects

In the UserEntry project described in "User Interface" (page 69), name and food information is stored in variables of the Main component, forgoing the benefits of an object-oriented system.

If you want to pass the information the user enters to other components, you have to pass both values. If you had more information about a particular person, you would have to pass each datum separately. It would be more convenient to encapsulate all the information about a user into one object and pass that from component to component. Since WebObjects is fully object-oriented, you can define a custom object to contain the user-entered data.

For now, you'll just encapsulate the same data into an object. Later, though, this kind of encapsulation is exactly what will allow you to use a database as your persistent data storage system.

Once you've defined the User class with the appropriate properties, you'll add a variable of type User to the Main component and modify the WOTextFields on `Main.wo` to use that variable's properties instead of the `personName` and `favoriteFood` instance variables.

Adding the Custom Class

In this section you'll create the custom class `User.java` and add it to the UserEntry project.

1. Make sure the UserEntry project is open in Project Builder.
2. Select the Classes group in the Groups & Files list.
3. Add the class file.
    a. Choose File > New File.
    b. In the New File Assistant window, under WebObjects, select Java Class and click Next.
    c. In the New Java Class pane of the Assistant, enter `User.java` in the File Name text field, select Application Server from the Targets list, and click Finish.
4. Move the variables and methods relating to the person name and favorite food properties and the `entryIncomple` method from `Main.java` to the new class by cutting and pasting.
5. Save `Main.java` and `User.java`.
6. Add a variable of type User to the Main component.
    a. Open the Main component in WebObjects Builder.
    b. Choose Add Key from the Edit Source menu.
    c. Name the variable `user` and choose User from the Type pop-up menu. Do not include accessor methods.

7. Instantiate the `user` variable in the Main class.

   You need to create a User object in the Main component. Make the constructor method of the Main class look like Listing 6-1.

Listing 6-1 Instantiating the `user` instance variable in the constructor of the `Main.java` class

```
public Main(WOContext context) {
    super(context);
    user = new User();
}
```

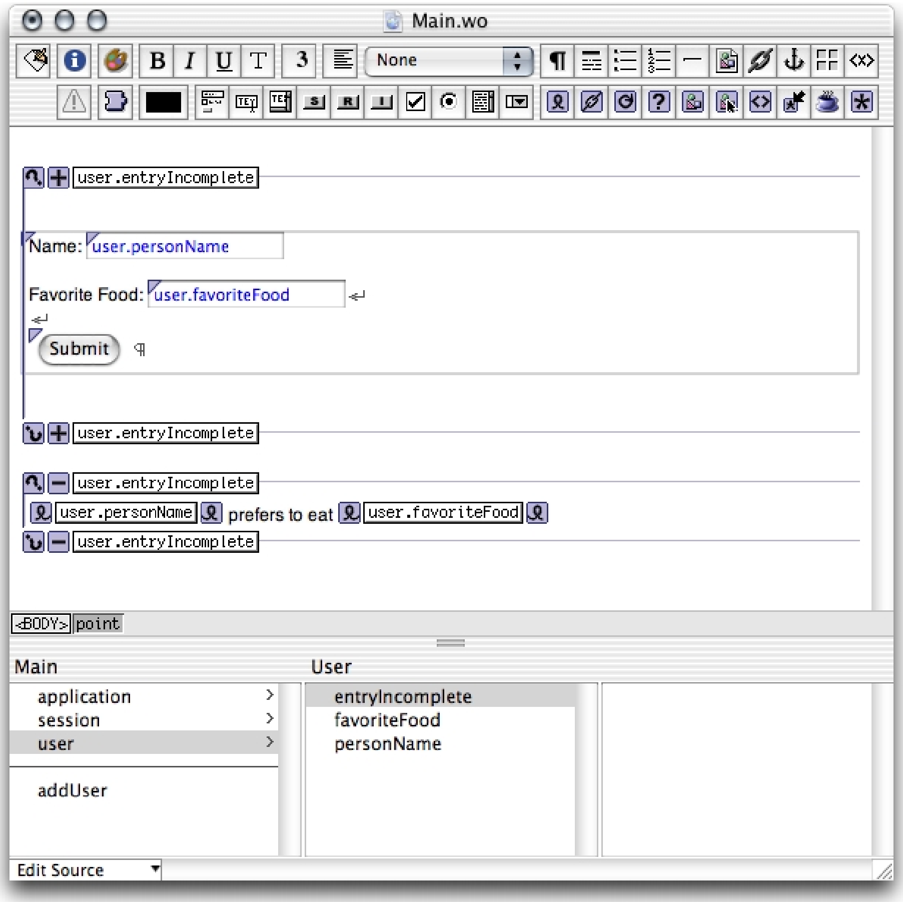
   Save `Main.java`.

8. Change the bindings on the dynamic elements to use the new variable.

   Click `user` in the variable browser of WebObjects Builder's main window. Drag a connection from the variable `favoriteFood` to the WOTextField that is currently bound to `favoriteFood`. When you release the mouse button, a pop-up menu lists the bindings available. Notice that `value` has a checkmark next to it, indicating that it currently has a binding. Selecting `value` replaces `favoriteFood` with `user.favoriteFood`.

   Bind `favoriteFood` to the WOString and replace the `personName` bindings in a similar fashion. Be sure to change both the WOStrings and the WOTextFields.

   Finally, bind `user.entryIncomple` to the WOConditionals.



9. Build and run the application.

   The application behaves in the same manner as before ("Managing User Input" (page 61)), but the data is now being accessed via the new custom object.

Following a Keypath

Notice that the bindings for the dynamic elements in the Main component are in a slightly different format. Rather than simply naming the variable or method to call, they specify a path to the property in question; for example, instead of referring to `personName`, the first WOString inside the WORepetition refers to `user.personName`. This is called a keypath.

Encapsulating data into objects, as in this example, is a very important part of object-oriented development. Access to this data is defined by a keypath that specifies the objects, methods, or variables that can provide the data in question.

A keypath is a set of keys separated by periods. When WebObjects requires access to data specified in a keypath, it follows the keypath by evaluating the first key from the list.

This first key is evaluated within the scope of the instance representing the component—the class file in the component is examined for the method or variable. In this case, the `user` instance variable found in the `Main.java` file.

At this point, if there is another key in the keypath, it is evaluated the same way, but using the result of the first keypath as the source object for the method or variable. Now, the `personName` method is called. Since there are no more keys in the keypath, the value from the `personName` method is returned as the value for the binding.

This way, you can access the data you need, as long as it can be reached by some method from the current component. In the simplest case, you store variables in the component itself. As your data becomes more complex, you may need to store it in custom objects and pass them between components.

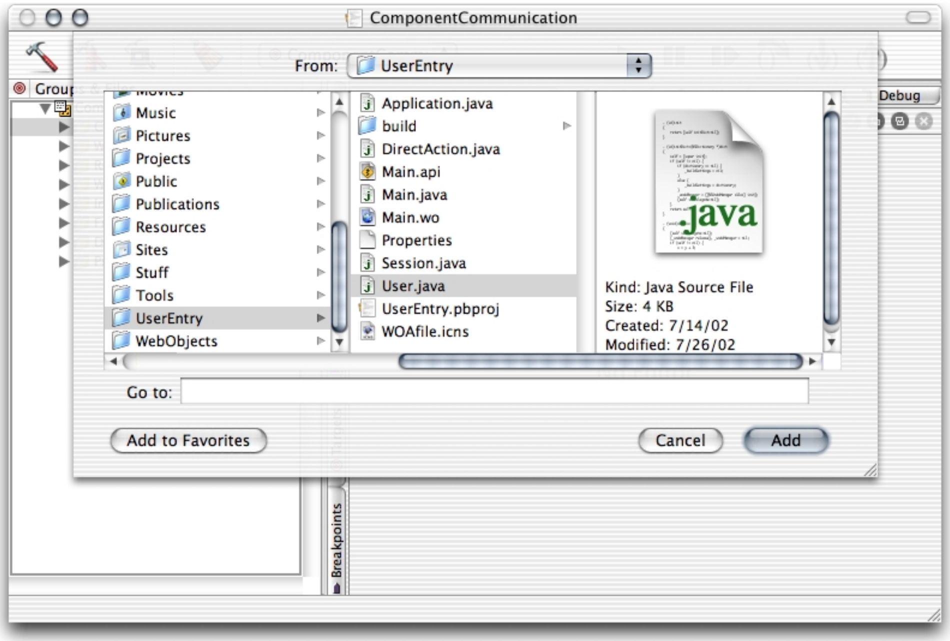
Defining a New Component

This section shows you how to create a component for displaying and editing a user's information.

Remember that each component has its own Java class. It is convenient to think of components, as well as the objects representing them, as self-contained units with specific tasks. The task of the component described here is to allow the user of your application to edit a User object. By encapsulating behavior this way, you ensure that if you add, remove, or alter the properties of users, you need to modify only this component to allow editing the new attributes.

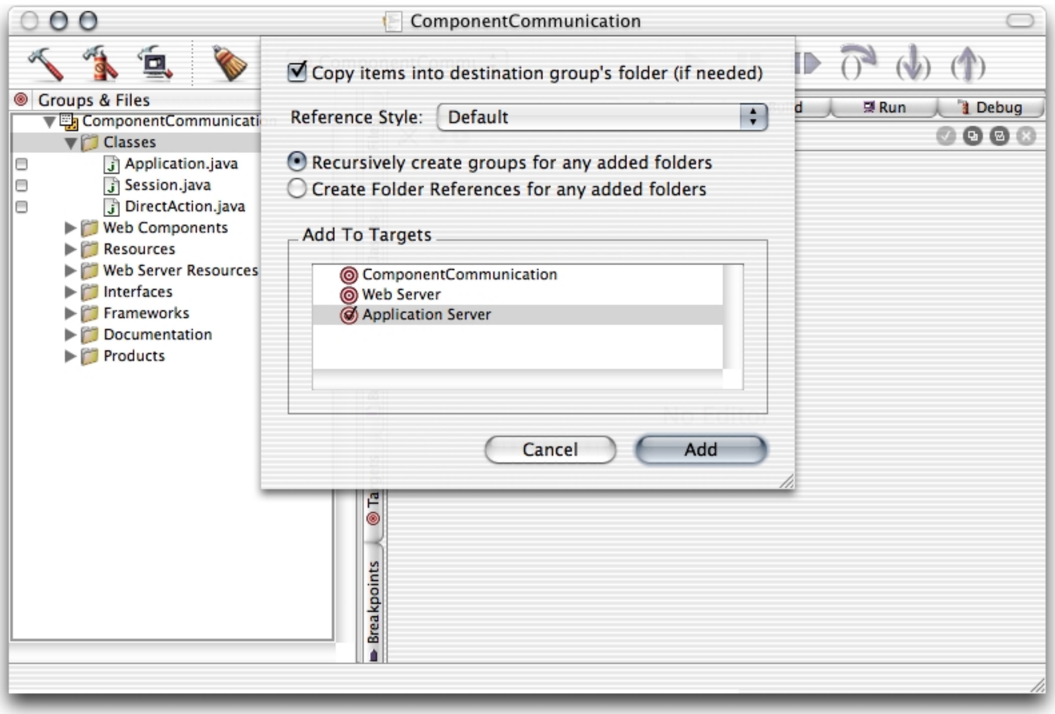
You begin by creating a new project. Then you copy the `User.java` file contained within the UserEntry project and create a component for editing a User object. Finally, you alter the Main component to maintain a list of users rather than a single user, and add methods to use the new component to edit any one of them.

1. Create a new WebObjects application project and name it `ComponentCommunication`.
2. Copy the `User.java` file from the UserEntry project.
    a. Select the Classes group from the Groups & Files list.
    b. Choose Project > Add Files.
    c. Navigate to the UserEntry project folder, select `User.java`, and click Add.



Kind: Java Source File
Size: 4 KB
Created: 7/14/02
Modified: 7/26/02

d. In the sheet that appears, select "Copy items into destination group's folder," select the Application Server target, and click Add.



☑ Copy items into destination group's folder (if needed)

Reference Style: Default

(●) Recursively create groups for any added folders
( ) Create Folder References for any added folders

Add To Targets
- ComponentCommunication
- Web Server
- ✓ Application Server

3. Add a component to the project.

   a. Select Web Components from the Groups & Files list.

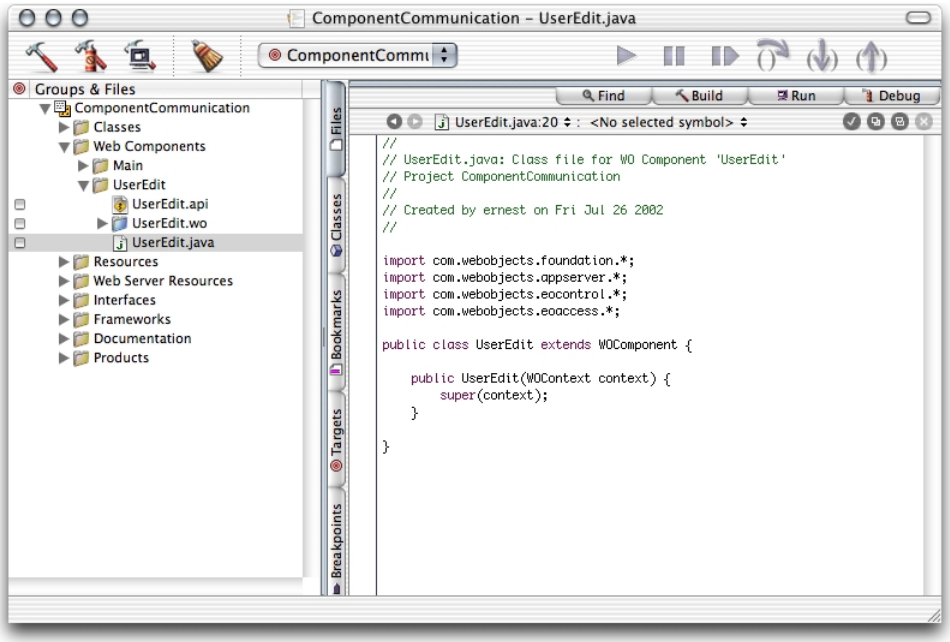
   b. Choose File > New File.

   c. In the New File pane of the Project Builder Assistant, select Component (under WebObjects) and click Next.

   d. Enter `UserEdit` in the File Name text field.

   e. Make sure Application Server is selected in the Targets list and click Finish.

Notice that the new component is added to the project's Web Components group.

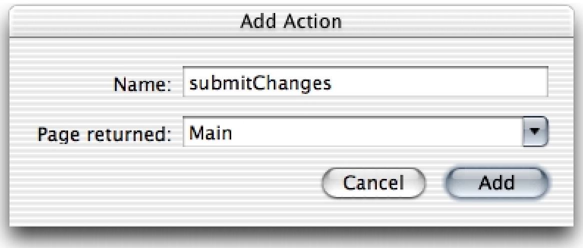


```
//
// UserEdit.java: Class file for WO Component 'UserEdit'
// Project ComponentCommunication
//
// Created by ernest on Fri Jul 26 2002
//

import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;

public class UserEdit extends WOComponent {

    public UserEdit(WOContext context) {
        super(context);
    }

}
```

You're now ready to customize the component used for editing a User object, UserEdit. The user edits one User object at a time, so `UserEdit.java` needs to have one instance variable of type User. The UserEdit component will have fields similar to those defined in the Main component of the UserEntry project (see "User Interface" (page 69)) and buttons to submit and cancel the changes.

1. Open `UserEdit.wo` in WebObjects Builder.
2. Add a User instance variable named `user` to the component.

   Select the options that create an instance variable and provide accessor methods. This variable holds the User object being edited.
3. Add a WOForm element to the UserEdit component.
4. Add the labels and WOTextFields shown in Figure 6-1 (page 94) and bind the WOTextFields to the `user.personName` and `user.favoriteFood`.
5. Add an action method called `submitChanges` to the component. Choose Main as the page returned by the method. This means that when the user is done editing, she's returned to the Main component rather than the UserEdit component.



Name: submitChanges

Page returned: Main

6. Use the Forms menu to add a WOSubmitButton and a WOResetButton, and bind the `submitChanges` method to the WOSubmitButton's `action` attribute.

   The WOResetButton resets the form fields.

7. Save `UserEdit.wo`.

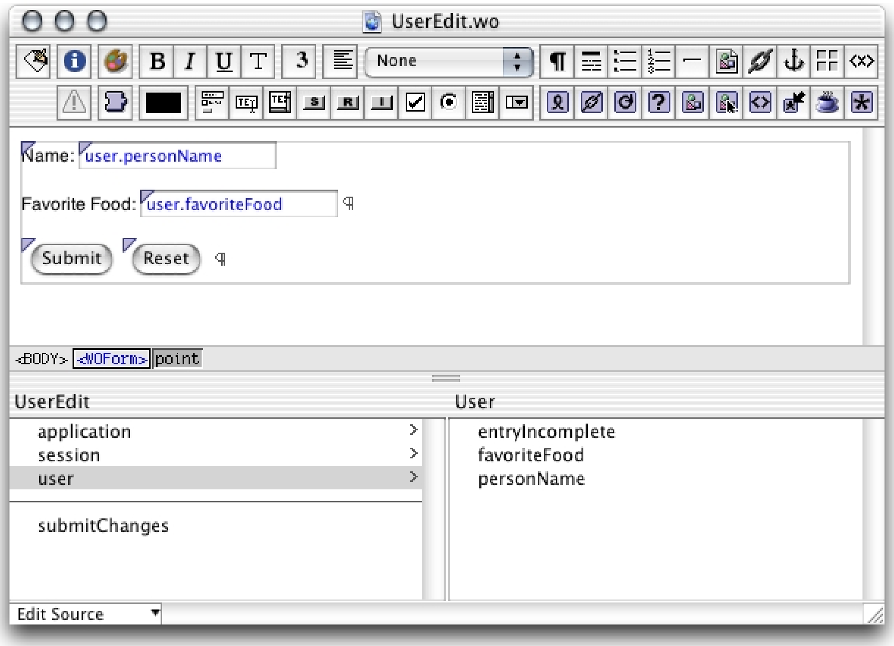
8. Edit the `submitChanges` method in `UserEdit.java` by adding the numbered lines in Listing 6-2.

Listing 6-2 The `submitChanges` method of `EditUser.java`

```
public Main submitChanges() {
    Main nextPage = (Main)pageWithName("Main");

    // Initialize your component here.

    // Send <code>user</code> to the Main page.                  //1
    nextPage.setUser(user);                                      //2

    return nextPage;
}
```

9. Save `UserEdit.java`.

Figure 6-1 The `UserEdit.wo` component in WebObjects Builder



Modifying the Main component

This section shows you how to add elements to the Main component so that it displays the user information after it has been edited. The component needs a WOConditional element so that user information is displayed only if the application's user entered data in the UserEdit page. After the modifications are made, `Main.wo` should look similar to Figure 6-2 (page 97).

1. Add a method called `noDataEntered` to `Main.java`, as shown in Listing 6-3.

Listing 6-3 The `noDataEntered` method of the `Main.java` class

```
/**
 * Determines whether the user has entered data into <code>user</code>.
 *
 * @return <code>true</code> when the user has entered data into
 *         <code>user</code>.
public boolean noDataEntered() {
    boolean noDataEntered;
    if (user == null || user.entryIncomplete()) {
        noDataEntered = true;
    }
    else {
        noDataEntered = false;
    }
    return noDataEntered;
}
```

2. Open `Main.wo` in WebObjects Builder.
3. Add a `user` instance variable of type User, including accessor methods.
4. Add informational text displayed when no data has been entered.
   a. Add a WOConditional element.
   b. Enter the following text inside the WOConditional: `User data has not been entered`.
   c. Bind the WOConditional's `condition` attribute to `noDataEntered`.
5. Add display fields and a caption displayed when data has been entered.
   a. Add another WOConditional element below the first one.
   b. Inside the second WOConditional, add a WOString, enter the text " `likes to eat` " after it, and add another WOString.
   c. Bind the first WOString's `value` attribute to `user.personName` and the second's to `user.favoriteFood`.
   d. Bind the WOConditional's `condition` attribute to `noDataEntered`.

      Click "+" in the WOConditional so that it changes to "-". This makes it so that only one of the WOConditional elements's content is displayed at a time. See "Conditional Display With WOConditional Elements" (page 77) for more information.

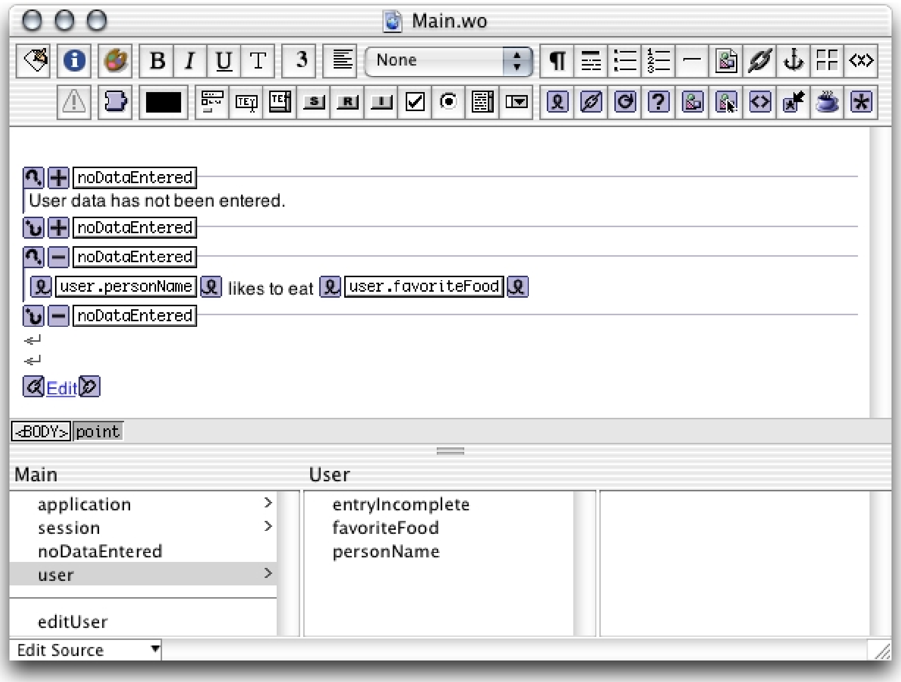
6. Add an action called `editUser` that returns a UserEdit component.
7. Add two line-feed characters below the second WOConditional by pressing Shift-Enter.
8. Add a link that displays the UserEdit page.
   a. Add a WOHyperlink below the second WOConditional, and enter `Edit` as its caption.
   b. Bind the WOHyperlink's `action` attribute to the `editUser` action.
9. Save `Main.wo`.
10. Edit the `editUser` method in `Main.java` by adding the numbered lines in Listing 6-4.

Listing 6-4 Main component's `editUser` action method

```
public UserEdit editUser() {
    UserEdit nextPage = (UserEdit)pageWithName("UserEdit");

    if (user == null) {                                                    //1
        user = new User();                                                 //2
    }

    // Send user to the UserEdit page.                                     //3
    nextPage.setUser(user);                                                //4

    return nextPage;
}
```

11. Save `Main.java`.

Figure 6-2 The `Main.wo` component in WebObjects Builder



Running the Application

Make sure the ComponentCommunication target is selected. Build and run the application. When the Main page is displayed, there is no user data to show (the `user` instance variable is `null`), therefore the message "User information has not been entered" appears instead. When the user clicks Edit, the Main component invokes its `userEdit` action, which returns a UserEdit page. If the user enters data in the Name and Favorite Food text fields in the UserEdit page and clicks Submit, UserEdit's `submitChanges` action, which returns a new Main page, is invoked.

There is only one instance of User during the application's execution. The User object is instantiated in Main's `editUser` method if it does not already exist (see Listing 6-4 (page 96). Main then sends this object to the newly created UserEdit page. Similarly, UserEdit sends the User object to a new Main instance in its `submitChanges` method.

C H A P T E R 7

Managing State

The Web—by its nature—is a stateless medium. A Web server receives a request, produces a response, and returns it to the requesting browser—without any knowledge of previous requests from the same user.

A Web application, however, needs to maintain state between one request from a particular user and the next. WebObjects encodes a unique identifier with each incoming request. This identifier is used to maintain state over a stateless medium. See "Request Processing" (page 62) for more information.

Part of this state is the session. While you can pass information back and forth between components, you frequently need to maintain state that is shared between components. Rather than pass this information from component to component (as described in "Component Communication" (page 85)), you can store it at a higher level—in the Session object. Each component has access to the Session object, so data stored in it is globally available.

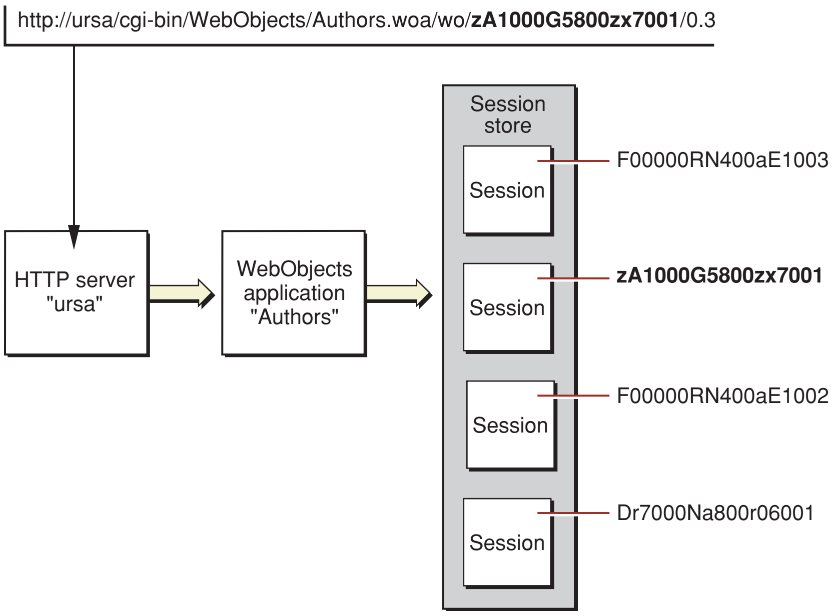
In this chapter, you

- store persistent information in the Session object
- access the session from multiple components
- see the benefit of making your components reusable

The Session

A **session** is a period of time in which one user interacts with your application. Since each application can have multiple users simultaneously, it may have multiple Session objects. Each session has its own data and its own cached copies of the components that the user has requested, as shown in Figure 7-1.

Figure 7-1 Relationship between application and session



The session is represented as an instance of the Session class (`Session.java` in a WebObjects application project). Initially, the session has only WebObjects-provided behavior, but you can add your own methods and variables.

For example, if you were building an online shopping application, the session would be an appropriate place to store a user's shopping cart, because the session is tied to one particular user and persists as long as the user is using the application.

When an incoming request is processed, WebObjects automatically activates the Session instance associated with the user who originated the request, as described in "Request Processing" (page 62).

The WOComponent class includes a method for accessing the currently active session. The Java files of all your components extend this class and WebObjects automatically activates the correct session when a request is processed. Calling the `session` method from your component (or in a keypath) provides you with the session for the current user.

Displaying and Editing Lists of Objects

Start by creating the SessionState project, using the Main and UserEdit components of the ComponentCommunication project created in "Component Communication" (page 85). Alternatively you can copy the SessionState directory (`projects/Chapter_7-Managing_State/starter/SessionState`) to your working directory and skip to "The NSArray and NSMutableArray Classes" (page 103).

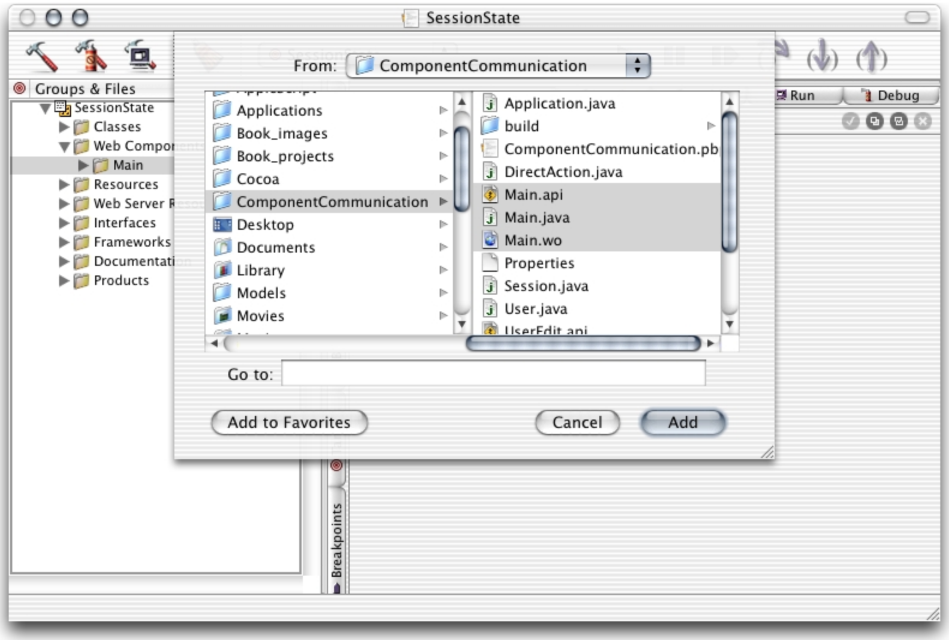
To create the SessionState project:

1. Create a new WebObjects application project and name it SessionState.
2. Remove the Main component from the new project.
   a. Select the Main group under Web Components.
   b. Press Command-Delete.
   c. Click Delete References & Files in the sheet.
   d. Delete the `Main.wo` bundle from the project's directory.
3. Add the Main component from the ComponentCommunication project directory.

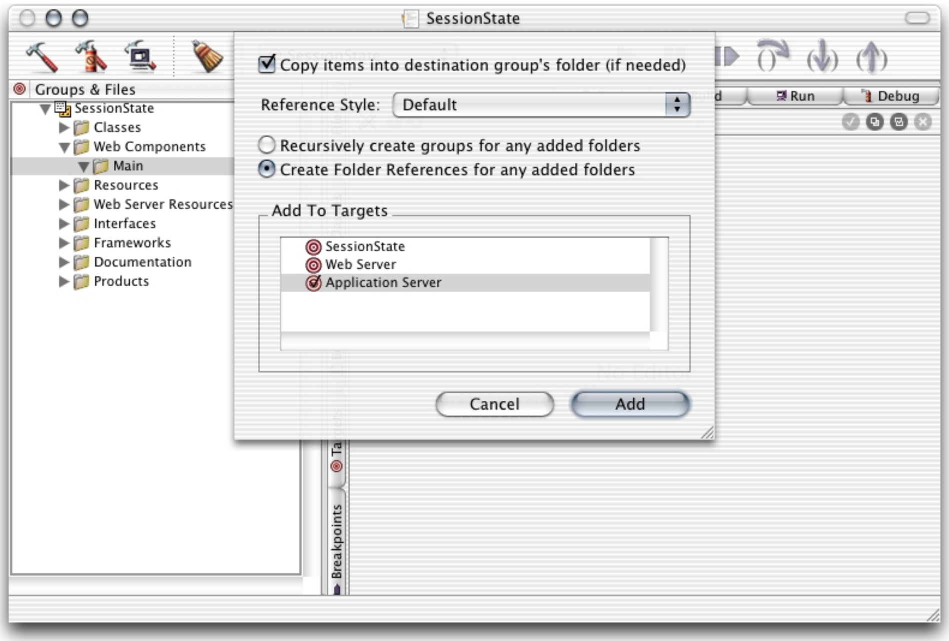
a. Select the Web Components group from the Groups & Files list.

b. Choose Project > New Group and name the group Main. Make sure the newly added group stays selected.

c. Choose Project > Add Files, select the `Main.api`, `Main.java`, and `Main.wo`, files, and click Add.



d. In the sheet that appears, select "Copy items into destination group's folder," "Create Folder References for any added folders," and the Application Server target, and click Add.



☑ Copy items into destination group's folder (if needed)

Reference Style: Default

○ Recursively create groups for any added folders

● Create Folder References for any added folders

Add To Targets

SessionState

Web Server

✓ Application Server

Now, add the UserEdit component of the ComponentCommunication project to the Web Components group and the User class to the Classes group.

The NSArray and NSMutableArray Classes

You'll now edit the Main component to show a list of users instead of just one. For that, you'll need to use the NSArray and NSMutableArray classes.

The NSArray class represents an ordered collection of objects, much like a Java array (`java.lang.Array`). NSArray objects are not changeable after being instantiated. (The array itself is not changeable, but the items it contains can be changed if their types are mutable.) The NSMutableArray class (a subclass of NSArray) is intended for arrays that need to grow and shrink dynamically.

The following sections list the NSArray and NSMutableArray methods that you may find useful when manipulating arrays.

NSArray

`objectAtIndex(int index)`

Returns the object at the given integer index. The first object in an NSArray is at index 0. Objects are returned as generic Objects. Your code may need to cast objects to a specific class to use them.

`count`

Returns an integer indicating the number of objects in the NSArray.

NSMutableArray

`addObject(Object anObject)`

Adds *anObject* to the end of the array, increasing its size by 1.

`removeObjectAtIndex(int index)`

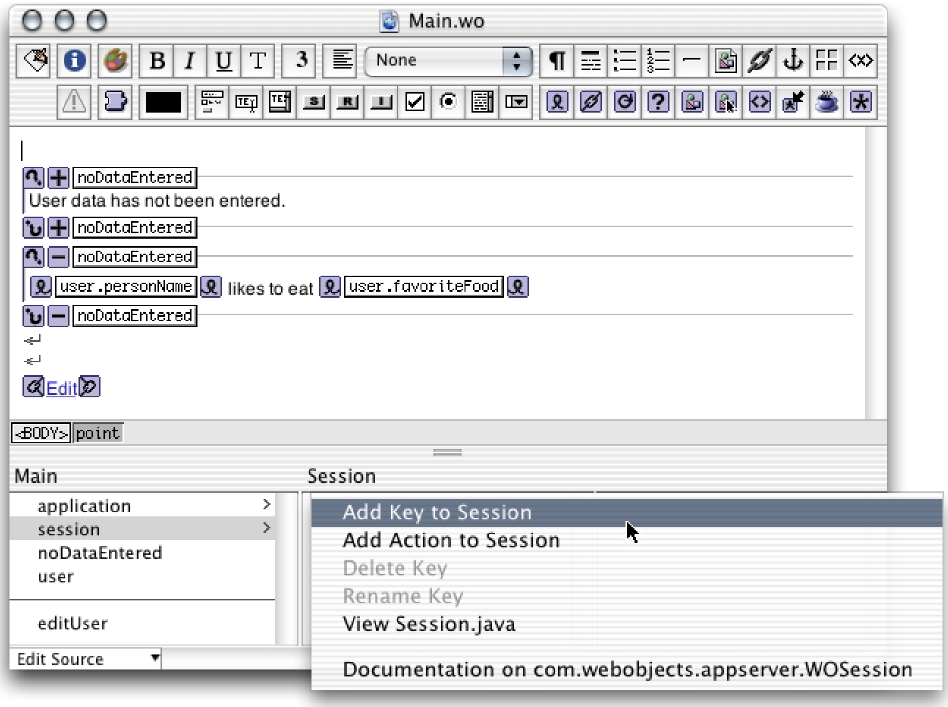
Removes the indicated object from the array, causing it to shrink in size.

In addition to these methods, the NSArray and NSMutableArray classes have other methods you may find useful. You can examine them using Java Browser or by consulting the WebObjects API documentation at http://developer.apple.com/webobjects.

Adding the NSMutableArray to the Session

You can use WebObjects Builder to add an array to the Session class.

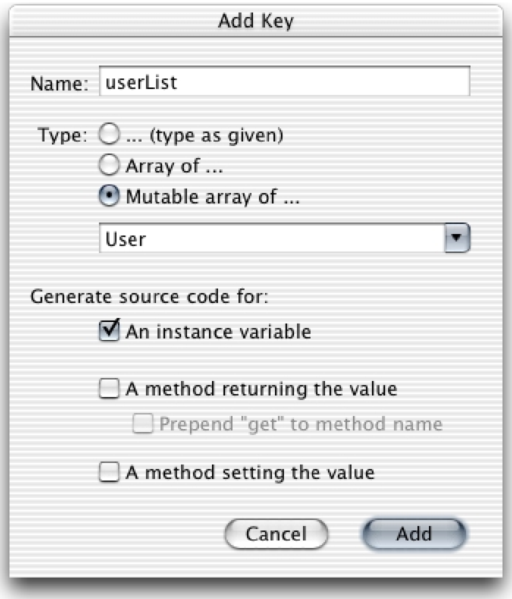
1. Open `Main.wo` in WebObjects Builder.
2. Add the `userList` instance variable to `session`.
   a. Select `session` in the Main list.
   b. Control-click the Session list and choose Add Key to Session from the pop-up menu that appears.



c. Name the variable `userList`.

d. Select the “Mutable array of” option and then User from the pop-up menu.

e. Under “Generate source code for,” make sure only “An instance variable” is selected and click Add.



Name: userList

Type: ( ) ... (type as given)
( ) Array of ...
(•) Mutable array of ...

User

Generate source code for:
[✓] An instance variable
[ ] A method returning the value
[ ] Prepend "get" to method name
[ ] A method setting the value

3. Initialize the new array in `Session.java`.

   The NSMutableArray needs to be instantiated when the Session object is created. It's empty when the application starts; here you add methods to add objects to it.

   Edit the constructor of the `Session.java` class by adding the `addToUserList` and `removeFromUserList` methods. Add the numbered lines in Listing 7-1.

Listing 7-1 The `Session.java` class of the SessionState project

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;

public class Session extends WOSession {
    /** @TypeInfo User */
    protected NSMutableArray userList;
    public Session() {
        super();

        userList = new NSMutableArray();                                //1
    }

    /**                                                                 //2
     * Adds a User object to <code>userList</code>.
     *
     * @param newUser   the User object to add
     */
    public void addToUserList(User newUser) {                           //3
        userList.addObject(newUser);                                    //4
    }                                                                   //5

    /**
     * Removes a User object from <code>userList</code>.
     *
     * @param aUser   the User object to remove
     */
    public void removeFromUserList(User aUser) {                        //6
        userList.removeObject(aUser);                                   //7
    }                                                                   //8
}
```

Adding the WORepetition to Main

A WORepetition element iterates over each item in an NSArray, repeating a set of HTML code (possibly including WebObjects elements) once for each item.

A WORepetition has bindings for a list to iterate over (the `list` attribute) and for a variable to use to hold each item temporarily as it iterates over the list (the `item` attribute). As the contents of a WORepetition are displayed, the current item in the list is stored in `item`. WebObjects elements within the WORepetition can refer to this placeholder variable, and the value of each item is substituted in turn.

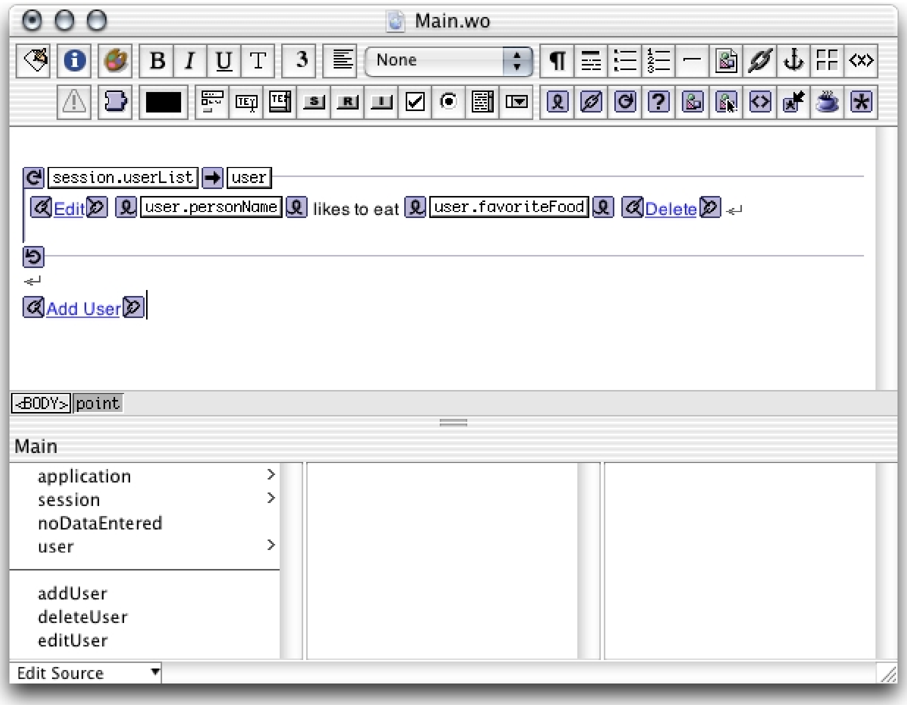
Now, you wrap the dynamic elements in `Main.wo` in a WORepetition. You can use the `user` instance variable as the WORepetition's placeholder. After performing the following steps, `Main.wo` should look similar to Figure 7-2 (page 109).

1. In `Main.wo`, delete the first WOConditional element (the one that contains the text "User information has not been entered."
2. Cut the internal contents of the remaining WOConditional.
3. Select the WOConditional and delete it.
4. Paste the content after the Edit WOHyperlink.
5. Add a `deleteUser` action that returns `null`.
6. Add a WOHyperlink to delete users.
    a. Add a WOHyperlink element after the second WOString.
    b. Enter `Delete` as the WOHyperlink's caption.
    c. Add a carriage return after the WOHyperlink by pressing Shift-Enter.
    d. Bind the Delete WOHyperlink's `action` attribute to the `deleteUser` method.
7. Wrap the dynamic elements in `Main.wo` with a WORepetition.
    a. Select all the elements in the page.
    b. Choose WebObjects > WORepetition or click Add the WORepetition button in the toolbar.
8. Bind the WORepetition's `list` attribute to `session.userList`.

    Drag from `session.userList` to the first square of the WORepetition.
9. Bind the WORepetition's `item` attribute to `user`.

    Drag from `user` to the second square of the WORepetition.
10. Add an `addUser` action that returns a UserEdit page.
11. Add a WOHyperlink to add new users.
    a. Add a WOHyperlink below the WORepetition.
    b. Enter `Add User` as the WOHyperlink's caption.
    c. Bind the WOHyperlink's `action` attribute to the `addUser` method.

Figure 7-2 The `Main.wo` component with a WORepetition



Editing the Users

You can use the UserEdit component to edit an arbitrary user. To do so, you'll use the `editUser` method in `Main.java`. The method has additional logic that is not needed in this application. Edit the `editUser` method so that it looks like Listing 7-2.

Listing 7-2 The `editUser` method of the `Main.java` class

```
public UserEdit editUser() {
    UserEdit nextPage = (UserEdit)pageWithName("UserEdit");

    nextPage.setUser(user);
    return nextPage;
}
```

The `editUser` method creates an instance of UserEdit and calls its `setUser` method with `user` as the argument. The `user` variable contains the appropriate object because, when the user clicks Edit, WebObjects stores the `session.userList` item corresponding to the row on which the Edit link is located in the `user` instance variable. Remember that the WORepetition's `item` attribute is bound to `user`.

The UserEdit component requires a minor change. The `submitChanges` method in `UserEdit.java` no longer needs to invoke the `setUser` method of the `Main.java` class (user information is stored at the session level, which Main can access through the `session` object). Edit the `submitChanges` method so that it looks like Listing 7-3.

Listing 7-3 The `submitChanges` method of the `UserEdit.java` class

```
public Main submitChanges() {
    Main nextPage = (Main)pageWithName("Main");

    return nextPage;
}
```

Adding Users

This is where it all ties together. Right now, you have a means of editing a specific user (the UserEdit component); a list of users, which starts out empty (`session.userList`); and a WORepetition that displays your list (in the Main component). All you need to add is a way to build the list.

You need to edit the `addUser` method in `Main.java` so that it creates a new User object, adds it to the session's list of users, and also passes it to the UserEdit page before it is sent to the Web browser to be edited. Edit `addUser` so that it matches Listing 7-4 by adding the numbered lines. Notice in particular the code that retrieves the Session object. The `addToUserList` method of that object is then invoked with the newly created User object as the argument.

Listing 7-4 The `addUser` method of the `Main.java` class

```
public UserEdit addUser() {
    UserEdit nextPage = (UserEdit)pageWithName("UserEdit");
    // Get the session for the current user.                              //1
    Session session = (Session)session();                                 //2

    // Create a new User object.                                          //3
    User newUser = new User();                                            //4

    // Add <code>newUser</code> to the session's user list.               //5
    session.addToUserList(newUser);                                       //6

    // Send <code>newUser</code> to the UserEdit page.                    //7
    nextPage.setUser(newUser);                                            //8

    return nextPage;
}
```

Deleting Users

The last step is to edit the `deleteUser` method in `Main.java` so that it removes a user from the list. The method is very similar to the `addUser` method described in "Adding Users" (page 110). The only difference is that, instead of creating a new user object and invoking the `Session.addToUserList` method, it invokes only the `Session.removeFromUserList` method with the User object in the `user` instance variable (updated by WebObjects when the user clicks Delete).

Add the numbered lines of Listing 7-5 to the `deleteUser` method in `Main.java`.

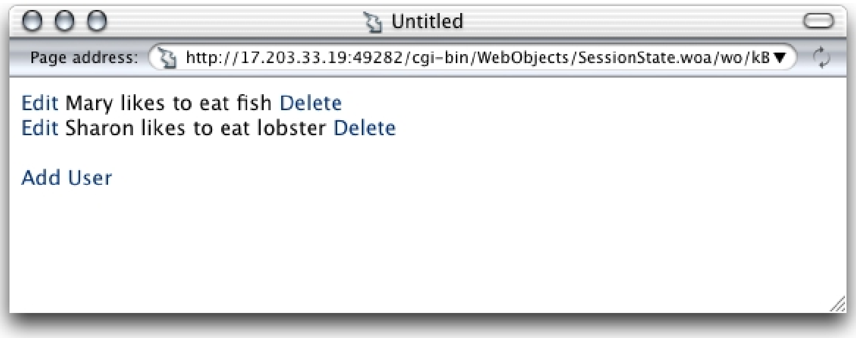
Listing 7-5 The `deleteUser` method of the `Main.java` class

```
public WOComponent deleteUser() {
    // Get the session for the current user.                              //1
    Session session = (Session)session();                                 //2

    // Remove <code>user</code> from the session's user list.             //3
    session.removeFromUserList(user);                                     //4

    return null;
}
```

Running the Application

Build and run the application. Verify that new users you create are added to the list and that you can edit and delete existing users.

Figure 7-3 The SessionState application in action


Untitled
Page address:
http://17.203.33.19:49282/cgi-bin/WebObjects/SessionState.woa/wo/kB

Edit Mary likes to eat fish Delete
Edit Sharon likes to eat lobster Delete

Add User

Benefits of Encapsulation

The benefits of moving the list of users to the session are all organizational. If you add other components that interact with the list of users to the application, they can all share the list present in the session without any additional code changes.

Notice also that your UserEdit page required only a minimal change, because you wrote it to work on any given User object without tightly bound relationships to other parts of the application. Because every WebObjects component is an object, you can use encapsulation, a traditional benefit of object-oriented programming.

Further Exploration

This chapter introduced several new concepts. These can be combined with the techniques you've already learned to add many features. Here are some ideas to get you started:

- **Improve the Edit link.** Give the user the ability to edit the users in the array by clicking their names rather than Edit. You can do this by replacing the static text within the WOHyperlink with the WOString that displays the user's name.
- **Indicate whether the entry is complete.** Add a textual indication of whether the user entry is complete using a WOConditional and the `entryIncomplete` property of the `user` variable.
- **Show the number of items in the list.** Use a WOString and the `count` method of `userList` to display the number of entries in the list.

C H A P T E R 8

Database Basics

WebObjects uses a technology called Enterprise Objects to access data from databases. While Enterprise Objects lets you treat your data as instances of Java classes rather than requiring you to concern yourself with the database level, it is useful to have some knowledge of basic relational-database structure.

This chapter is only a basic introduction to the structure and use of a database. It is sufficient for you to begin using Enterprise Objects, but you should consult the documentation that accompanies your database or your local database administrator for further details. You can also learn about how Enterprise Objects serves as an interface between your business logic and your database by consulting *Inside WebObjects: Using EOModeler*.

In this chapter, you learn

- how a relational-database is structured
- how relationships are implemented in a database

Database Structure

A relational database loosely follows the object-oriented paradigm. The basic unit of organization is the **table**, which defines the attributes of a data entity. A table can have zero or more rows.

Tables

The table is the equivalent of a class definition. It defines the attributes or properties that each **row** in it has. Each property is called a **column**.

Much like in Java, most databases use primitive data types and each column on a table is defined to be of a given one of these types. Enterprise Objects handles the conversion from database internal types to Java types. Further, for each column, you can declare whether a given row is required to provide a value or whether it is allowed to be `null`.

You can think of a table as a class, columns as instance variables, and rows as individual object instances. Like a class definition file in Java, a table does not have variables. You must add a row to it to use any of its properties.

Rows

A row is the equivalent of an instance of a Java class. Where the table is like a class, a row is like an instance of a class.

Uniquing

Each row in a given table has to be different in some way from the others. This is so the database system knows which row to update or delete when you make changes to your data.

The database uses a **primary key** that defines a property (or set of properties) whose value uniquely identifies each row. For each table you define, you provide a key (or list of keys) that defines how the database system can be sure two given rows are different. For example, if you are defining a table to contain data about people, you might decide to use a person's last name to differentiate each row from the others.

The database system ensures that each row has a unique value in the column (or combination of values in the set of columns) you specify as the primary key. Enterprise Objects hides many of the complexities of database interaction, including the conversion from database internal types to Java types. However, if you try to override it and set a value in a row that conflicts with a value in another row, the database refuses to make the change and reports an error that your application should handle.

It's important to choose the primary keys for your tables carefully. If you define your unique key as the column containing a person's last name, you could run into difficulty as soon as you try to add two people with the same last name to your table. In general, it's best to make your primary key one you don't plan to use for anything else and let Enterprise Objects handle it for you.

In the example Authors database (see "Creating the Authors Database" (page 129)), each table has a column that servers as its primary key called *ENTITYNAME*_ID of type integer.

Not Null

You may wish to declare that without certain data, a row isn't valid. By declaring a given column *not-null*, you tell the database system to reject any new rows that don't provide the required data.

If you're gathering information for an email mailing list, for example, a row without a value for the EMAIL_ADDRESS column isn't useful.

A table's primary key column must always be declared not-null.

Relationships

Part of a database's scheme is each table's relationships with other tables. Each relationship has source and destination keys that define it.

Rows and tables can relate in many ways. Just as with object-oriented programming, the way you design your database tables is depends on how you intend to use them. Relationships can model ownership, where one row is a subordinate part of another row, but placed in a separate table for organization—for example, each row in a PERSON table can *own* a row in an ADDRESS table because every person must have a mailing address. Relationships can be optional or required. In the Authors database, every book must have an author, but it is conceivable that some authors are as yet unpublished (and hence would have no rows in the BOOK table).

Since a relationship leads from a source table to a destination table, we speak of *following* a relationship. While Enterprise Objects removes most database considerations from using your object's relationships, it's useful to understand how a relationship is maintained at the database level.

Relationships are followed not from table to table but from a specific row into another row. It doesn't make any sense to ask what the author of the BOOK table is, but each row of the BOOK table has a corresponding row in the AUTHOR table.

To link table rows, **foreign keys** are used. Foreign keys are columns in the source table whose rows point to the primary-key column in the target table. For example, the BOOK table has a foreign-key column (AUTHOR_ID) that is used to find the corresponding row in the AUTHOR table (the author of the book). AUTHOR_ID in the BOOK table does not provide any information on a book. Its only purpose is to link the rows of the BOOK table with rows in the AUTHOR table.

One important attribute of a relationship is *ordinality*. Ordinality is a measure of whether a relationship necessarily relates a row to only one other row, or to multiple rows in the destination table. This breaks relationships into two types: **to-one** or **to-many**.

To-One Relationships

A to-one relationship, as the name suggests, is one where the source row is connected to only one row. Each row in the BOOK table has only one author in the AUTHOR table.

The destination column of a to-one relationship must be the same as the primary-key columns of the destination table. This guarantees that there is only one destination row for any given source row.

To find the row corresponding to a specific book's author, you would perform the following steps:

1. Get the source and destination columns.

   According to the definition of the relationship, the source column is AUTHOR_ID in the BOOK table, and the destination column is AUTHOR_ID in the AUTHOR table.

2. Get the value in the source row's AUTHOR_ID column.
3. Find the target row in the AUTHOR table (the row whose AUTHOR_ID column is equal to the value of the AUTHOR_ID column in the source row).

   Since AUTHOR_ID is the primary key for the AUTHOR table, there is only one matching row. This row contains data about the book's author.

To-Many Relationships

Alternatively, a relationship can connect the source row with multiple destination rows. For example, in the Authors database, each author can have multiple books. Remember that the AUTHOR_ID column in the BOOK table is a foreign key, not a primary key. Therefore, that column doesn't have to have unique values throughout the rows of the BOOK table.

To find the BOOK rows corresponding to an AUTHOR row, you would perform the following steps:

1. Get the source and destination columns.

   According to the definition of the relationship, the source column is AUTHOR_ID in the AUTHOR table, and the destination attribute is AUTHOR_ID in the BOOK table.
2. Get the value in the row's AUTHOR_ID column for the source row.
3. Find the rows in the BOOK table whose AUTHOR_ID column's value is equal to the value from the AUTHOR_ID column in the source row.

   Notice that AUTHOR_ID is not the primary key for the BOOK table. This means that the relationship could lead to more than one row in the BOOK table. Each of these would have the same value in their AUTHOR_ID column, meaning that the books that they represent have been written by the same author.

   Further, there's no guarantee that there would be any rows in the BOOK table whose AUTHOR_ID column's value match the value of the AUTHOR_ID column in the source row at all. So the relationship could lead to no rows in the BOOK table.

C H A P T E R 9

Introduction to Enterprise Objects

Enterprise Objects is a powerful system that makes using a database as a persistent storage for your data almost transparent to you as a developer. It removes the need for you to work with SQL or other database-querying languages by providing an object-oriented API that you use to manage data just as you manage simple objects, regardless of the type of data source the data resides in.

This abstraction layer allows you to concentrate on the business logic of your **enterprise objects** (the Java classes that EOModeler generates from your data model) rather than spending your time implementing dozens of procedures to handle your data on the database. It also allows you to benefit from the advantages that object-oriented programming provides. EOModeler is an application that lets you create a data model from a database or database tables from a data model. For more information on EOModeler, see *Inside WebObjects: Using EOModeler*.

This chapter introduces the theory behind the Enterprise Object technology. "Creating an Enterprise Objects Application" (page 129), puts the theory to use.

This chapter explains

- the layers that make up an Enterprise Objects application
- the role the model plays in an Enterprise Objects application

System Architecture

Enterprise Objects is a suite of tools and code that allow you to create database-based applications. It is divided into several layers concerned with connecting to the database, converting result sets to enterprise objects, and ensuring that the state of the enterprise objects and the database are always synchronized. WebObjects adds another layer on top of Enterprise Objects; it is used to manipulate enterprise objects and display their data.

The following components, listed from the WebObjects layer down to the database, make up the Enterprise Objects architecture:

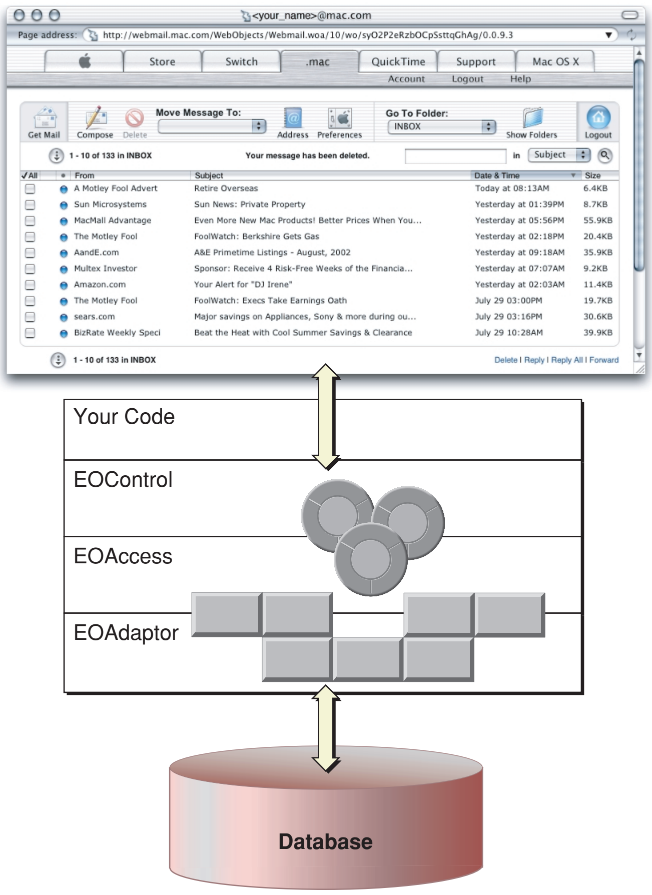
- WebObjects **components** display and manipulate enterprise objects.
- **Enterprise objects** are the instances of EOGenericRecord or Java classes you create to represent your database rows. EOGenericRecord (`com.webobjects.eocontrol.EOGenericRecord`) provides the default behavior of propagating changes to the database, but does not allow the addition of custom logic.
- An **editing context** manages a graph of objects and keeps track of changes that need to be transmitted to the database.
- The **model** (maintained with EOModeler) provides a high-level view of your data entities. It defines the mapping between the data entities your application requires and the tables in your database. It also defines relationships between entities, which are reflected in the database tables with primary-key and foreign-key definitions.

  The model, specifies which columns in your database are associated with a particular property for each entity of your data model. It represents the entire data store, from table structure to delete rules. It specifies to Enterprise Objects how to translate data between the enterprise objects and database rows. You use EOModeler to create models.

  The model also specifies the information needed to connect to the database, including network and password information.
- The **database level** translates from the dictionaries used by the adaptor level to enterprise objects, and vice versa.
- The **adaptor level** understands the preferred protocol of the database, and uses it to translate between simple objects called dictionaries and the raw data in the database.
- The **database** is external to Enterprise Objects and provided by a third party.

Figure 9-1 illustrates the approach that Enterprise Objects takes when interacting with a database.

Figure 9-1 The Enterprise Objects approach



1 - 10 of 133 in INBOX   Your message has been deleted.

✓All	●	From	Subject	Date & Time	Size
☐	●	A Motley Fool Advert	Retire Overseas	Today at 08:13AM	6.4KB
☐	●	Sun Microsystems	Sun News: Private Property	Yesterday at 01:39PM	8.7KB
☐	●	MacMall Advantage	Even More New Mac Products! Better Prices When You...	Yesterday at 05:56PM	55.9KB
☐	●	The Motley Fool	FoolWatch: Berkshire Gets Gas	Yesterday at 02:18PM	20.4KB
☐	●	AandE.com	A&E Primetime Listings - August, 2002	Yesterday at 09:18AM	35.9KB
☐	●	Multex Investor	Sponsor: Receive 4 Risk-Free Weeks of the Financia...	Yesterday at 07:07AM	9.2KB
☐	●	Amazon.com	Your Alert for "DJ Irene"	Yesterday at 02:03AM	11.4KB
☐	●	The Motley Fool	FoolWatch: Execs Take Earnings Oath	July 29 03:00PM	19.7KB
☐	●	sears.com	Major savings on Appliances, Sony & more during ou...	July 29 03:16PM	30.6KB
☐	●	BizRate Weekly Speci	Beat the Heat with Cool Summer Savings & Clearance	July 29 10:28AM	39.9KB

1 - 10 of 133 in INBOX

Previous chapters taught you how to manipulate objects using WebObjects components. Now, we start at the top with those objects and follow the interactions in the Enterprise Objects layer down to the database.

Enterprise Objects

An enterprise object is first and foremost a Java object. It has instance variables and methods that act on them. However, it has the additional characteristic of being linked to a database structure by Enterprise Objects. Enterprise objects differ from other objects in that they are a representation of data that is stored in a database.

Each enterprise object typically represents one row from a database. When the properties (instance variables) of an enterprise object are changed and you instruct Enterprise Objects to save those changes, they are propagated through the layers down to the database.

An enterprise object can be an instance of the default EOGenericRecord class or of a custom Java class. You use an EOGenericRecord when you don't need special behavior beyond that of basic representation of database values. You define a custom class when you wish to have more control over the properties and behavior of your data. Custom classes are defined as subclasses of EOGenericRecord so they inherit the default enterprise-object behavior.

Enterprise objects, whether represented by EOGenericRecord or a custom class, are defined in a model created with EOModeler.

EOGenericRecords use the **key-value coding** mechanism defined in the EOKeyValueCoding interface to store their data. Each key is named for the database column it represents. When an enterprise object is instantiated from a row in the database, the value of its keys are obtained from their corresponding columns in the row. WebObjects dynamic elements use key-value coding to get and set the values of enterprise-object attributes.

EOControl

The control layer is the principal domain of enterprise objects. It provides an insulated layer dedicated to maintaining the state of enterprise objects. Data flows out and upward to WebObjects components and can be propagated downward toward the database. The EOControl layer is responsible for

- tracking changes to enterprise objects
- updating the database when changes are saved
- managing undo operations in the object graph
- managing uniquing in the object graph

Uniquing is used by Enterprise Objects to ensure that an enterprise object is not duplicated in the control layer. This mechanism uses an entity's primary key to determine the identity and uniqueness of each enterprise object in the object graph. It is important that enterprise objects not be duplicated in the object graph to maintain data integrity and use memory efficiently. For example, if two books have the same author, the control layer ensures that they both refer to the same Author object in memory. Uniquing is one of the responsibilities of the object graph.

The Object Graph

An object graph is a collection of all the currently active enterprise objects for a particular external store. You can think of it as a snapshot of the current state of the database reflected in Java objects.

An object graph can also represent a potential state of the database. If your components make changes to some enterprise objects, those changes are stored in an object graph until they are committed to the database. Keeping track of these changes is the responsibility of the EOEditingContext class (`com.webobjects.eocontrol.EOEditingContext`).

The Editing Context

Each editing context object manages one object graph, keeping track of any changed properties of each of its enterprise objects. It also preserves their original values so changes can be undone.

Typically, a set of changes reflecting user input is accumulated in the object graph of an editing context. At some point, the changes are either committed to the database for permanent storage, or they are undone, reverting the object graph to its original state. If the changes are committed, the editing context notifies the EOAccess layer of the changes made to enterprise objects so that it can make the necessary changes to the database.

You can create editing contexts in your application. However, by default, each session has an editing context associated with it. This default editing context, accessible by all components, is usually sufficient.

EOAccess

The access layer provides access to the database through a standardized protocol. Every piece of data crossing between the access layer is in the form of an enterprise object. This level of abstraction makes the job of the control layer much simpler since it can rely on the format of the data.

The access layer is divided into two parts: the adaptor level and the database level.

The Adaptor Level

The adaptor level is where Enterprise Objects translates data from a database and packages it as key-value dictionaries. Currently, the JDBC (Java Database Connectivity) standard is used for database access, but the adaptor level makes it possible to allow access to other database systems, such as legacy databases, simply by adding an adaptor. This allows a developer to remain unconcerned with the specific database to be used while writing code.

The Database Level

The database level manages details about the database that developers don't need to be concerned with. Enterprise objects are created from raw data from the database, and when data is needed by the control layer, the database layer performs the needed fetches from the database. Similarly, the database layer handles the actual updates to the database when an editing context is saved.

C H A P T E R 1 0

Creating an Enterprise Objects Application

Now that you have read the theory behind Enterprise Objects, you can put some of it into practice. This section describes a project similar to the ones explained in previous chapters, but with the added functionality of managing data stored in a database. As you follow the examples, refer to the previous chapters as necessary to make sure you understand the concepts on which each step of the process relies.

In this chapter, you

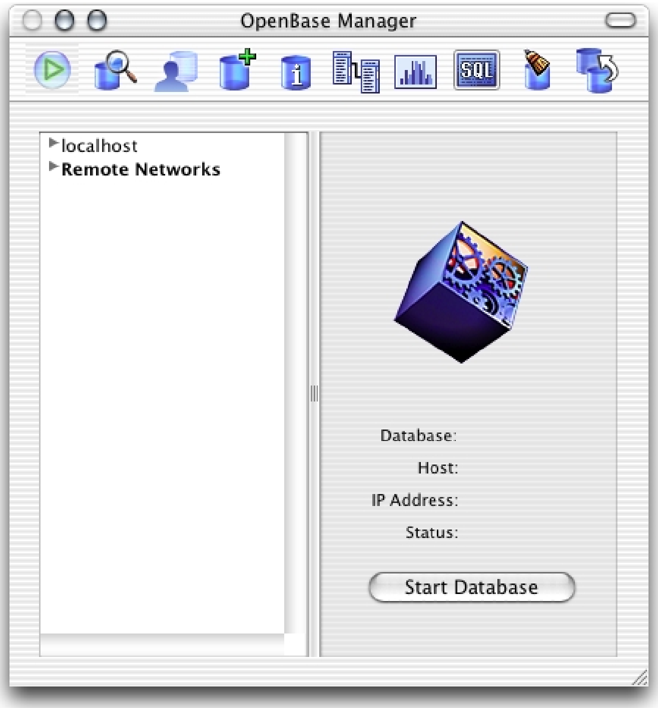
- create a database using OpenBase Manager
- create a data model containing one entity using EOModeler
- create a database table from an entity definition using EOModeler
- perform fetch, insert, update, and delete operations on an object store
- save editing context changes to a data store

Creating the Authors Database

The first step is to create the Authors database. This section describes how to use OpenBase Manager to create a database.

1. Launch OpenBase Manager.

   The OpenBase Manager application is located in `/Applications/OpenBase`.



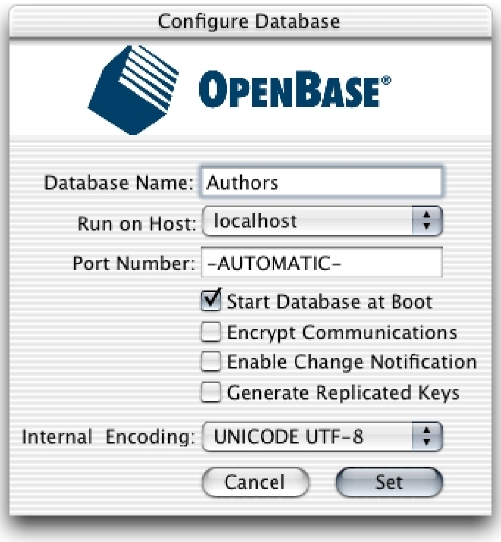
2. Create a new database.

   a. Choose Database > New.

   b. Enter `Authors` in the Database Name text field.

   c. Select the Start Database at Boot option.

   d. Choose UNICODE UTF-8 from the Internal Encoding pop-up menu and click Set.



Database Name: Authors
Run on Host: localhost
Port Number: -AUTOMATIC-
☑ Start Database at Boot
☐ Encrypt Communications
☐ Enable Change Notification
☐ Generate Replicated Keys
Internal Encoding: UNICODE UTF-8

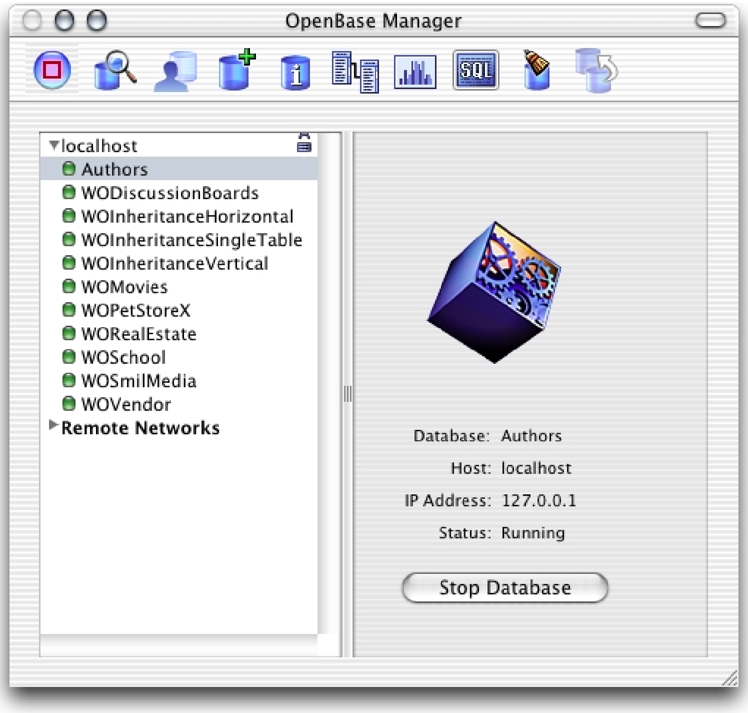
3. Start the database.

   a. From the database list, select Authors under localhost.

   b. Click Start Database.

      When finished, the OpenBase Manager window should look like Figure 10-1.

Figure 10-1 The OpenBase Manager window with the Authors database



Database: Authors
Host: localhost
IP Address: 127.0.0.1
Status: Running

Creating the Authors Model

This section shows you how to create a data model named Authors with one entity named Author and a corresponding database table named AUTHOR. The table stores the first and last names of book authors.

EOModeler is the tool you use to model your data. In it you define the entities that serve as the interface between your code and the database.

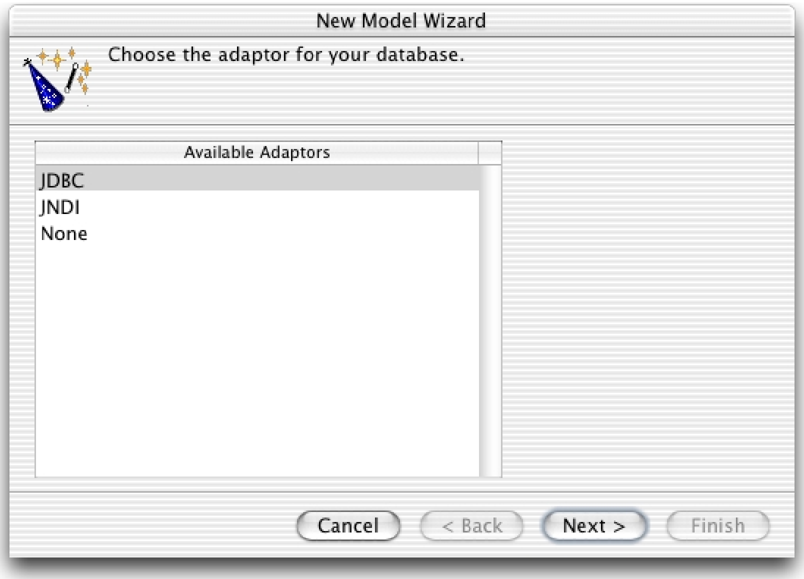
1. Launch EOModeler.

   The EOModeler application is located in `/Developer/Applications`.

2. Choose Model > New.

3. Select the adaptor to use.

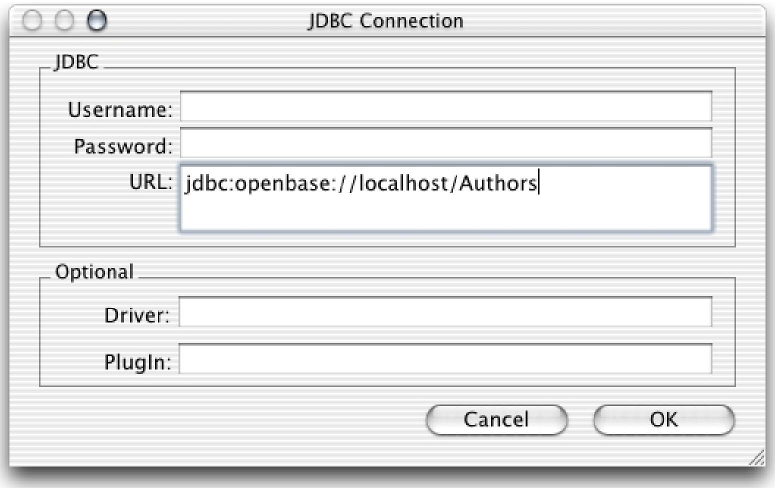
   With the JDBC adaptor provided with your WebObjects installation, you can communicate with any database that includes a JDBC driver.



   Select JDBC from the list and click Next.

4. Provide JDBC connection information.

   Model files include the information necessary to connect to databases. The JDBC Connection dialog is where you enter that information. For this exercise, you only need to specify the URL used to connect to the Authors database.


JDBC Connection

JDBC

Username:

Password:

URL: jdbc:openbase://localhost/Authors

Optional

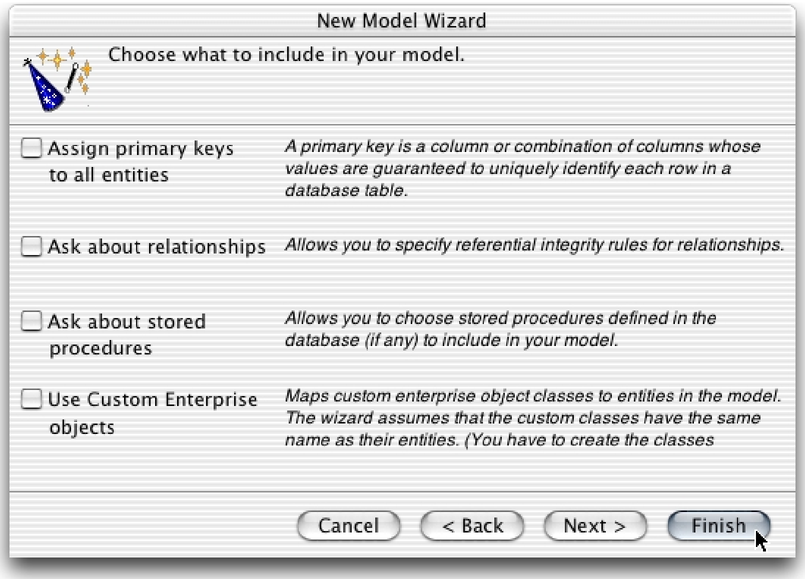
Driver:

PlugIn:

Enter `jdbc:openbase://localhost/Authors` in the URL text field and click OK.

5. Select what to include in your model.

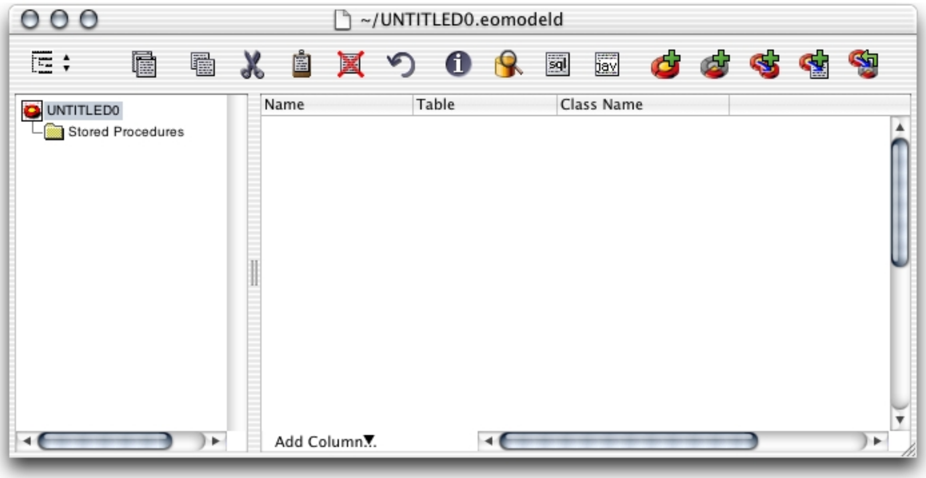
   This pane is where you tell EOModeler how to configure the model entities from an existing database. Because you are creating a new database, none of these options need to be selected. For information on what these options mean, see *Inside WebObjects: Using EOModeler*.


New Model Wizard

Choose what to include in your model.

☐ Assign primary keys to all entities	*A primary key is a column or combination of columns whose values are guaranteed to uniquely identify each row in a database table.*
☐ Ask about relationships	*Allows you to specify referential integrity rules for relationships.*
☐ Ask about stored procedures	*Allows you to choose stored procedures defined in the database (if any) to include in your model.*
☐ Use Custom Enterprise objects	*Maps custom enterprise object classes to entities in the model. The wizard assumes that the custom classes have the same name as their entities. (You have to create the classes*

Deselect all the options and click Finish.


~/UNTITLED0.eomodeld

UNTITLED0
Stored Procedures

Name	Table	Class Name

Adding the Author Entity to the Model

This section shows you how to add an entity called Author to the new model. This entity represents the AUTHOR table in the Authors database.

The Author entity contains three attributes: `firstName`, `lastName`, and `authorID`. The `authorID` attribute serves as the entity's primary key, and its value is not shown in the application's user interface. You don't even need to worry about updating the value of that attribute as you create Author objects; Enterprise Objects does it for you. As you define the Author entity, you also give EOModeler the information it needs to create the AUTHOR table.

1. Add the entity.

   Choose Property > Add Entity.

2. Configure the entity.

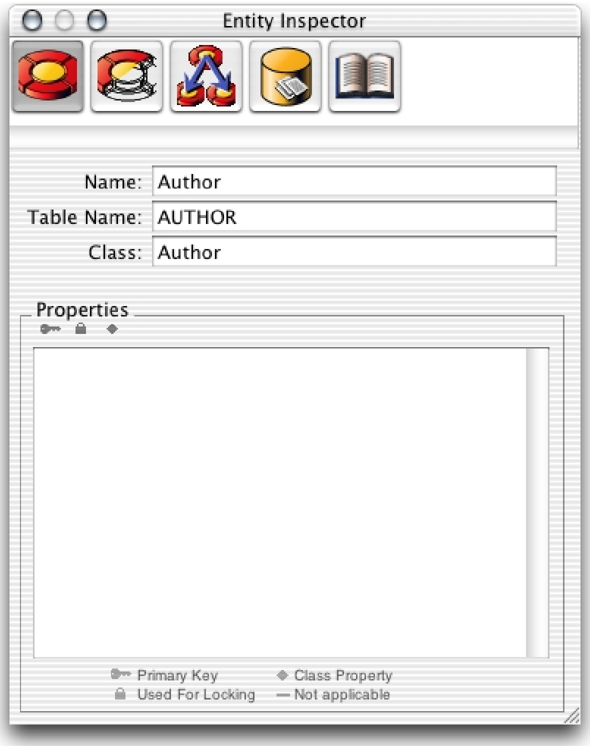
   Choose Tools > Inspector.

   The Entity Inspector allows you to enter a variety of information pertaining to the new entity.

   a. Name the entity `Author`.

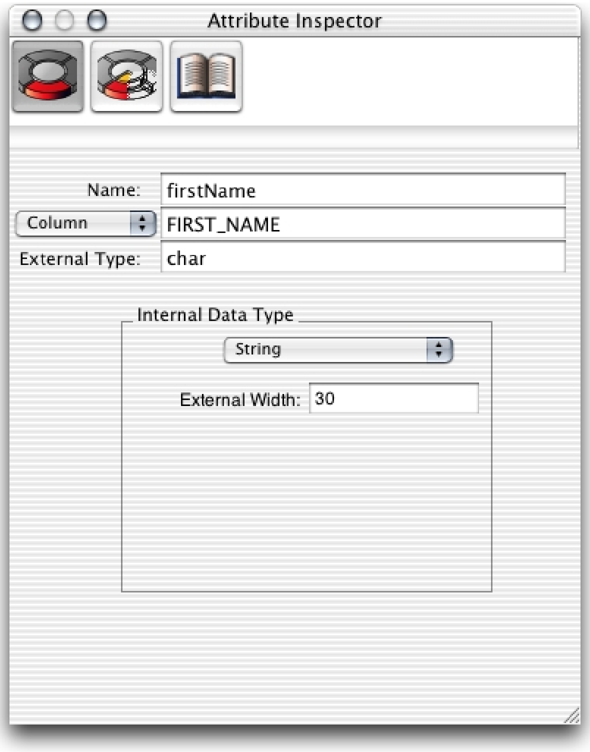
   b. Enter `AUTHOR` in the Table Name text field.

   c. Enter `Author` in the Class text field.



Name: Author
Table Name: AUTHOR
Class: Author

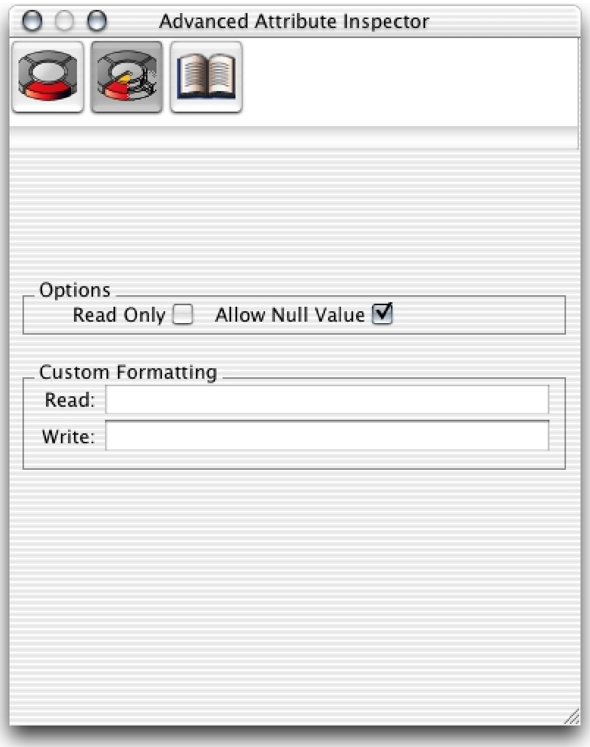
3. Add and configure Author's attributes.
   a. Make sure the Author entity is selected in the entity list.
   b. Choose Property > Add Attribute.
   c. Name the attribute `firstName`.
   d. Enter `FIRST_NAME` as the column name.
   e. Enter `char` in the External Type text field.
   f. Choose String as the internal data type.
   g. Enter `30` in the External Width text field.



Name: firstName
Column: FIRST_NAME
External Type: char
Internal Data Type: String
External Width: 30

h. Click the second button on the Attribute Inspector's toolbar to display the Advanced Attribute Inspector.

i. Select the Allow Null Value option.



Options
Read Only ☐ Allow Null Value ✓

Custom Formatting
Read:
Write:

j. Repeat steps a through i to add the `lastName` attribute.

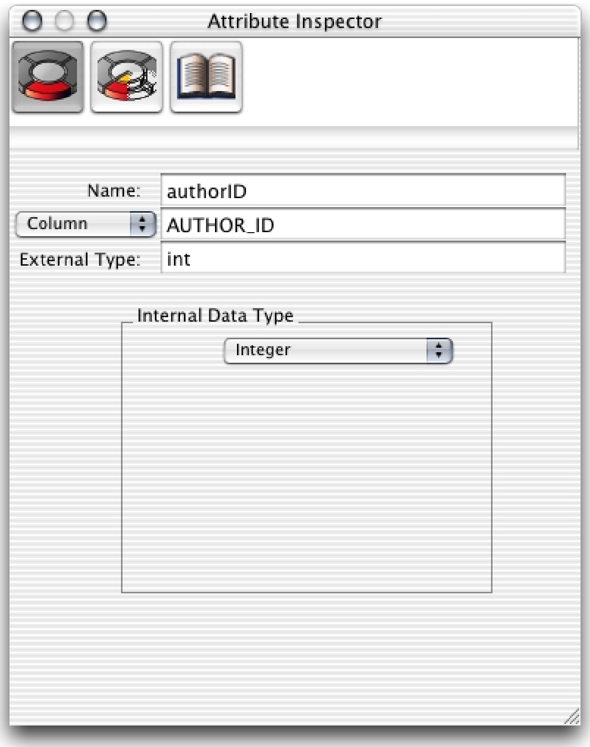
Now, add the attribute that serves as the primary key.

a. Add a new attribute and name it `authorId`.

b. Enter `AUTHOR_ID` as the column name.

c. Enter `int` in the External Type text field.

d. Choose Integer as the internal data type.



Name: authorID
Column: AUTHOR_ID
External Type: int
Internal Data Type: Integer

4. Select a primary key for the Author entity.
    a. In the `authorId` row of the Author Attributes list, click the column with a key as its heading so that a key appears in the row.
    b. Click in the diamond column of the `authorId` row so that the diamond disappears.

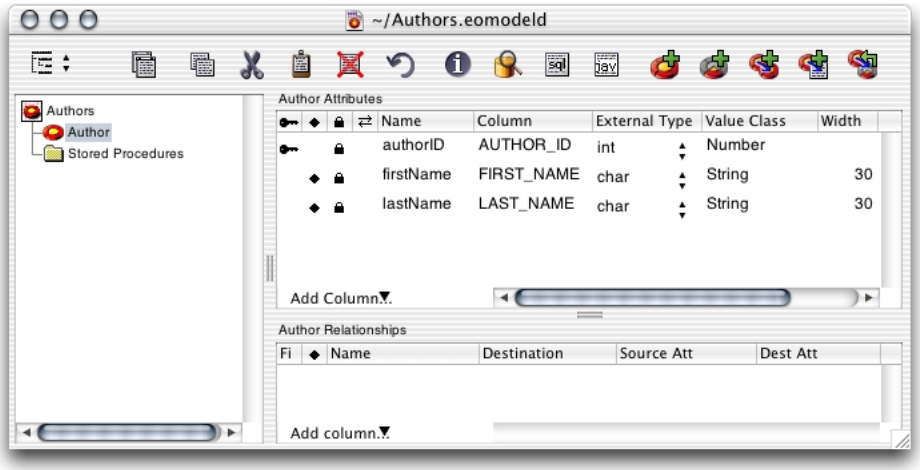
        The `authorID` attribute is nothing more than a relational-database artifact, required to make sure that rows in the AUTHOR table are unique; it has no meaning to you or the application's users. The diamond icon indicates that an attribute is a property that is made available to an application's custom logic and, if necessary, the application's user. Because `authorID` provides no additional information about an author, it is not required for the application's normal operation.
5. Save the model and name it Authors.

The EOModeler Window

The left pane of EOModeler’s main window lists the entities present in the model. If you click an entity, details about its attributes are displayed in the right pane.

Figure 10-2 shows the Authors model window with the Author entity and the definitions of each of its attributes. The values of the columns indicate the properties of each attribute.

Figure 10-2 Authors model with Authors entity



🔑	◆	🔒	⇄	Name	Column	External Type	Value Class	Width
🔑		🔒		authorID	AUTHOR_ID	int	Number	
	◆	🔒		firstName	FIRST_NAME	char	String	30
	◆	🔒		lastName	LAST_NAME	char	String	30

Fi	◆	Name	Destination	Source Att	Dest Att

By default, the most commonly used columns are displayed in this view. To display other columns, use the Add Column menu in the bottom frame of the window. These are the available columns and their meanings:

Primary Key
: The primary-key icon in the first column indicates that the attribute is used to uniquely identify a row. In the Author entity, only `authorID` is a primary key.

Class Property
: The presence of a diamond in the second column indicates that the attribute is a class property. A class property is one for which EOModeler generates Java access methods. Generally, any attributes that are actually a property of the entity are made class properties, and attributes that are used for database level functionality (such as the `authorId` attribute) are not.

Locking
: Indicates whether an attribute should be used for locking when an update is performed. That is, whether Enterprise Objects uses this attribute to determine whether changes have been made.

Client-side attribute
: In Java Client applications, the column with the opposing arrows determines whether the attribute's value is sent from the server to the client. For more information on Java Client technology, see *Inside WebObjects: Java Client Desktop Applications*.

Allows Null
: Indicates whether the database column can have a `null` value.

Name
: The name of the attribute, which determines the Java method names that EOModeler generates when it creates the class definition.

Value Class (Java)
: The class used to represent this attribute.

External Type
: The data type used by the database to represent this attribute.

Width
: The maximum width of an attribute, usually used for String attributes.

Column
: The name of the database column that corresponds to this attribute.

Definition
: The definition for a derived column. A derived attribute doesn't actually exist in the database and hence an attribute can't have values for both Definition and Column. Setting one clears the other.

Precision
: The number of significant digits to include. Used for some numerical types.

Prototype
: The prototype from which this attribute inherits its characteristics. You can use prototypes to set up default attribute types.

Read Format
: Defines formatting that is applied to data read from the database before populating the enterprise-object attribute.

Scale
: The number of characters to the right of the decimal point in a number attribute.

Value Type
: This type is used in decoding values for enterprise objects represented by Objective-C classes rather than Java classes and is not used in this document.

Write Format
: Used in tandem with Read Format to write data to the database in a custom format.

The External Type attribute must be one of the types defined by the JDBC adaptor. These are the most common ones:

`blob`
: A Binary Large Object. Used to store images and large data files. Usually represented as an NSData object.

`char`
: Used to store character information and represented with a Java String. An attribute declared to be a `char` value must have its width set.

`date`, `datetime`
: Used to store date information and usually represented by an NSTimestamp object.

`double`
: Used to store floating-point numbers and generically represented by a Java Number.

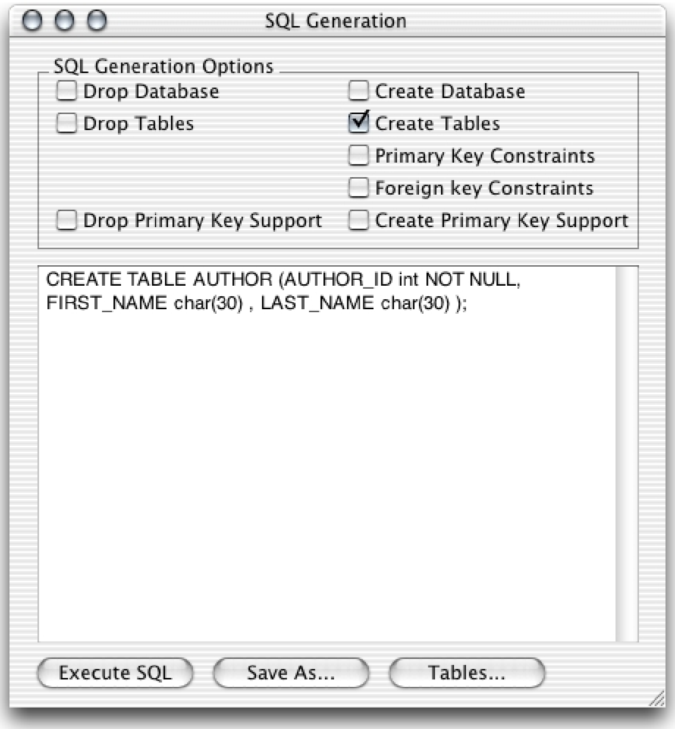
`int`
: Used to store integer numbers and usually represented by a Java Number. Foreign and primary keys are usually best modeled as integers.

`long`
: Used to store very large integers.

Creating the AUTHOR Table

After defining the Author entity, creating the AUTHOR table is easy.

1. Select the Author entity from the entity list.
2. Choose Property > Generate SQL.
3. Make sure that only the Create Tables option is selected.



```
CREATE TABLE AUTHOR (AUTHOR_ID int NOT NULL,
FIRST_NAME char(30) , LAST_NAME char(30) );
```

4. Click Execute SQL.
5. Quit EOModeler.

Creating the Authors Project

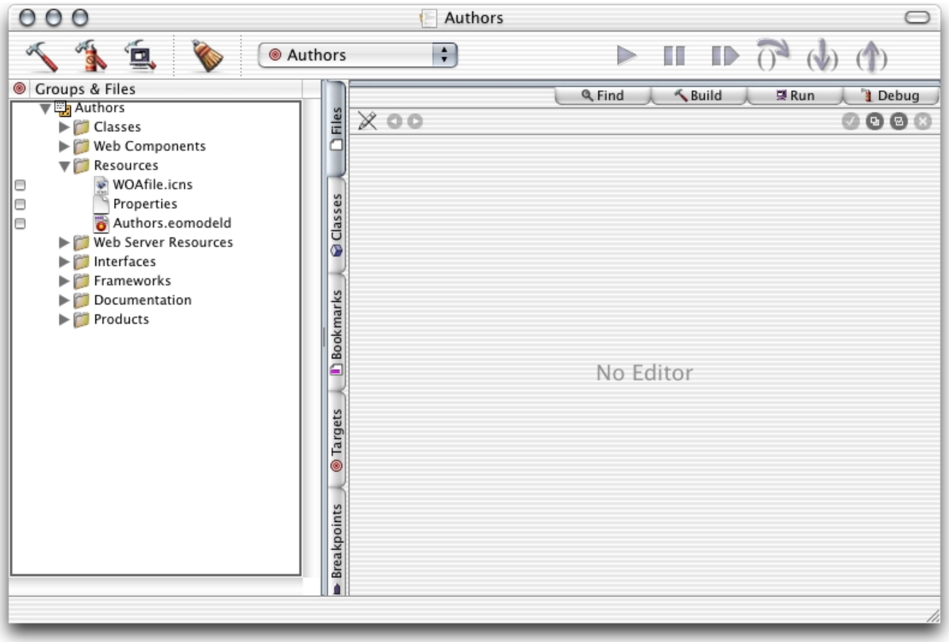
In this section you create the Authors application. The application allows its users to add, edit, and remove authors from the Authors database.

This section introduces the use of enterprise-object classes (custom Java classes derived from entities defined in a model to access database information) and the methods used to add objects into the data store (adding rows to the AUTHOR table in the Authors database). You use EOModeler to create the `Author.java` class. After adding it to your project, you'll be able to create Author objects in your code. You then add those objects to the data store.

1. Create a new WebObjects application project and name it `Authors`.
2. Choose the Java JDBC Adaptor framework in the Choose EOAdaptors pane of the Project Builder Assistant.
3. In the Choose EOModels pane of the Assistant, click Add, select the Authors model file you created in "Creating the Authors Model" (page 132), and click Choose. Then click Finish.

When done, Project Builder's main window should look like Figure 10-3. Notice that Project Builder placed a copy of the Authors model file in the project's folder and added it to the Resources group of the Groups & Files list.

Figure 10-3 The Authors project in Project Builder



Customizing the Main Component

The entire application's functionality is provided by the Main component. It includes an `authorList` array where the authors are maintained while the application runs. When the user clicks Save, the changes made to `authorList` are saved to the database. `Main.wo` includes elements to edit an author's information and actions to add, edit, update, and delete authors. A WORepetition shows the contents of `authorList`.

Customizing `Main.wo`

After following these steps, `Main.wo` should look like Figure 10-4 (page 149).

1. Open `Main.wo` in WebObjects Builder.
2. Add three keys.
   a. Name the first key `author`, set its type as EOGenericRecord, and do not include accessor methods.
   b. Name the second key `authorItem`, set its type as EOGenericRecord, and do not include accessor methods.
   c. Name the third key `authorList`, choose "Mutable array of" and EOGenericRecord for its type, and do not include accessor methods.
3. Add six actions, all of them returning `null`, which tells WebObjects to return the current page, Main, instead of a new one (the same instance of Main persists throughout the application's operation):

```
addAuthor
deleteAuthor
editAuthor
revertChanges
saveChanges
updateAuthor
```

4. Add a WOForm element to edit author information.
   a. Choose Forms > WOForm.
   b. In the WOForm Binding Inspector, choose `true` for the `multipleSubmit` attribute.
   c. Inside the WOForm, enter the text "`Last Name: `", follow it with a WOTextField, and press Shift-Enter.
   d. Bind the Last Name WOTextField's `value` attribute to `author.lastName`.

Note: Since `author`, `authorItem`, and `authorList` are EOGenericRecords, WebObjects Builder does not know what their properties are. You must type the keypaths for them manually.

   e. Enter the text "`First Name: `", follow it with a WOTextField, and press Shift-Enter.

f. Bind the First Name WOTextField's `value` attribute to `author.firstName`.

g. Add two WOSubmitButtons to the WOForm.

   Enter “`Update`” for the first WOSubmitButton's `value` attribute (include the quotation marks), and bind its `action` attribute to the `updateAuthor` action.

   Enter “`Add`” for the second WOSubmitButton's `value` attribute, and bind its `action` attribute to the `addAuthor` action.

5. Add a second WOForm below the first one for the Save and Revert WOSubmitButtons.

   a. Set the WOForm's `multipleSubmit` attribute to `true`.

   b. Add a WOSubmitButton inside the WOForm, enter “`Revert`” for its value attribute, and bind its action attribute to `revertChanges`.

   c. Add another WOSubmitButton to the right of the Revert WOSubmitButton, enter “`Save`” for its `value` attribute, and bind its `action` attribute to `saveChanges`.

6. Add a WORepetition to display the list of authors.

   a. Add the WORepetition below the second WOForm.

   b. Add two WOHyperlinks, separated by a space character, inside the WORepetition.

      Enter `Edit` as the first WOHyperlink's caption and bind its `action` attribute to `editAuthor`.

      Enter `Delete` as the second WOHyperlink's caption and bind its `action` attribute to `deleteAuthor`.

   c. Add two WOStrings, separated by “`, `“ (a comma and a space) to the right of the Delete WOHyperlink.

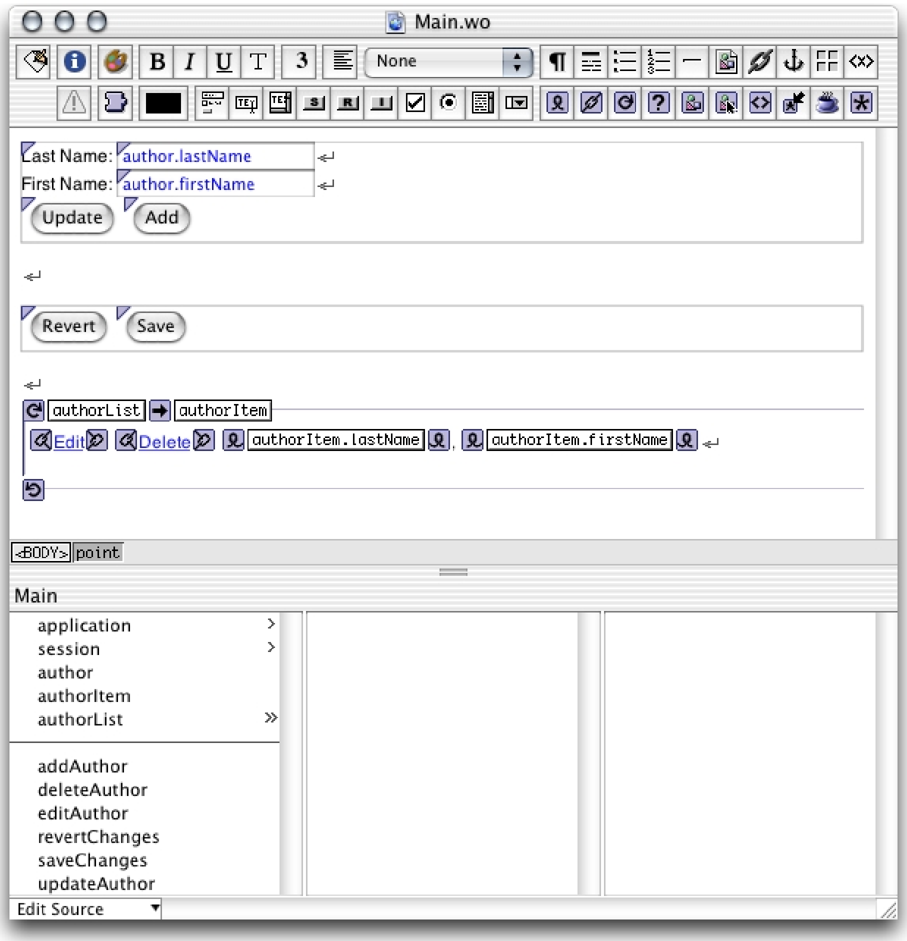
      Bind the first WOString to `authorItem.lastName` and the second to `authorItem.firstName`.

      Put the cursor on the right of the second WOString and press Shift-Enter.

   d. Bind WORepetition's `list` attribute to `authorList`, and its `item` attribute to `authorItem`.

7. Save `Main.wo`.

Figure 10-4 The `Main.wo` component with elements to maintain author information



Customizing `Main.java`

Now, edit `Main.java` to add the application's custom logic.

1. Add the following instance variables:

```
/**
 * References the default editing context.
```

```
     */
    private EOEditingContext editingContext;

    /**
     * Stores the class description of the Author entity.
     */
    private EOClassDescription authorClassDescription;

    /**
     * Stores the fetch specification used to retrieve Authors
     * from the data store.
     */
    private EOFetchSpecification fetchSpec;
```

Several methods in the Main class require the use of the editing context, class description, and fetch specification. Having the class's constructor store these objects in instance variables reduces the lines of code required to implement those methods.

2. Edit the constructor to perform custom initialization.

   When the Main component is created, it needs to request and store the editing context and retrieve the authors stored in the database (the first time you run the application, there's nothing to retrieve).

   Edit the constructor so that it looks like Listing 10-1.

Listing 10-1 The constructor in `Main.java`

```
public Main(WOContext context) {
    super(context);

    // Build the fetch specification.
    fetchSpec = new EOFetchSpecification("Author", null, null);

    // Get the editing context.
    editingContext = session().defaultEditingContext();

    // Fetch authors.
    authorList = new
NSMutableArray(editingContext.objectsWithFetchSpecification(fetchSpec));
```

```
        // Get the Author class description from the Author entity.
        authorClassDescription =
    EOClassDescription.classDescriptionForEntityName("Author");

        // Create an Author object (where form data is stored).
        author = new EOGenericRecord(authorClassDescription);
    }
```

There are three parts to retrieving data from a database with Enterprise Objects: the fetch specification, the editing context, and the fetch.

- **EOFetchSpecification.** An EOFetchSpecification is a representation of a request for objects from the object store (database). It describes the objects that you want to retrieve. You can create fetch specifications programmatically or define them in the model file. For details on entering fetch specification in a model file, see *Inside WebObjects: Using EOModeler*.

  A fetch specification is defined in three parts—the entity to fetch, restrictions used to filter the fetched objects, and the order of the result. The last two are optional, but the first one must be provided when the fetch specification is created.

- **Editing Context.** Fetches are performed through an editing context, which is responsible for maintaining the object graph of the fetched objects.

- **Fetch.** After defining a fetch specification, you can use it to fetch data from the object store. Enterprise Objects translates the fetch specification into SQL statements that your database system can understand. The database returns a list of rows that the Enterprise Object technology translates into enterprise objects (instances of EOGenericRecord) before returning them in an NSArray.

3. Edit the `addAuthor` method so that it looks like Listing 10-2.

Listing 10-2 The `addAuthor` method in `Main.java`

```
public WOComponent addAuthor() {
    // Add the author only when it isn't in the list.
    if (!authorList.containsObject(author)) {
        // Add the author to the list.
        authorList.addObject(author);
```

```
            // Insert author into editing context.
            editingContext.insertObject(author);

            // Create a new author.
            author = new EOGenericRecord(authorClassDescription);
        }

        return null;
    }
```

The Enterprise Object technology makes it easy to insert rows into your database tables; all you have to do is add items to an array. After creating an enterprise object (a subclass of EOGenericRecord), insert the object into an editing context; it is then maintained in the object graph like other objects fetched from the object store. When your application invokes `saveChanges` method, Enterprise Objects creates a row in the appropriate tables for each object added to the editing context.

The `addAuthor` method is invoked when the user clicks Add. If the user isn't editing an existing author, it inserts the Author object that the user edited (through the first form's text fields) into `authorList`, and inserts it in the object graph maintained by the editing context as well. It then creates an Author object, where another author's data can be stored. (Note that the new instance is added to the object graph only if the user clicks Add again.)

4. Edit the `deleteAuthor` method so that it looks like Listing 10-3.

Listing 10-3 The `deleteAuthor` method in `Main.java`

```
public WOComponent deleteAuthor() {
    // Remove the Author object in authorItem from authorList.
    authorList.removeObject(authorItem);

    // Get authorItem's editing context.
    EOEditingContext ec = authorItem.editingContext();

    // Remove the Author object from the editing context.
    ec.deleteObject(authorItem);

    return null;
}
```

In multiuser applications, an object can be in a different editing context than the default one. When you need to delete an enterprise object from a data store, you should ask the object itself for its editing context. Then you invoke that editing context's `deleteObject` method.

5. Edit the `editAuthor` method so that it looks like Listing 10-4.

Listing 10-4 The `editAuthor` method in `Main.java`

```
public WOComponent editAuthor() {
    // Set the author to edit to the one the user selected.
    author = authorItem;

    return null;
}
```

When the user clicks Edit, `authorItem` contains the Author object to be edited. The next time the page is drawn, the text fields are populated with the information for the selected author.

6. Edit the `updateAuthor` method so that it looks like Listing 10-5.

Listing 10-5 The `updateAuthor` method in `Main.java`

```
public WOComponent updateAuthor() {
    // Create an Author object.
    author = new EOGenericRecord(authorClassDescription);

    return null;
}
```

When the user clicks Update, the Author object she edited gets updated with the values entered in the form's text fields (the object is already in the list). Therefore, the only thing this method needs to do is create a new Author object. The next time the page is drawn, the text fields are populated with nothing (because they get their data from the new, empty Author object), allowing the user to enter the information for a new author.

7. Edit the `saveChanges` method so that it looks like Listing 10-6.

Listing 10-6 The `saveChanges` method in `Main.java`

```
public WOComponent saveChanges() {
    // Save the changes made in the editing context to the object store.
    editingContext.saveChanges();

    return null;
}
```

8. Edit the `revertChanges` method so that it looks like Listing 10-7.

Listing 10-7 The `revertChanges` method in `Main.java`

```
public WOComponent revertChanges() {
    // Discard the changes made in the editing context.
    editingContext.revert();

    // Refetch.
    authorList = new
NSMutableArray(editingContext.objectsWithFetchSpecification(fetchSpec));

    return null;
}
```

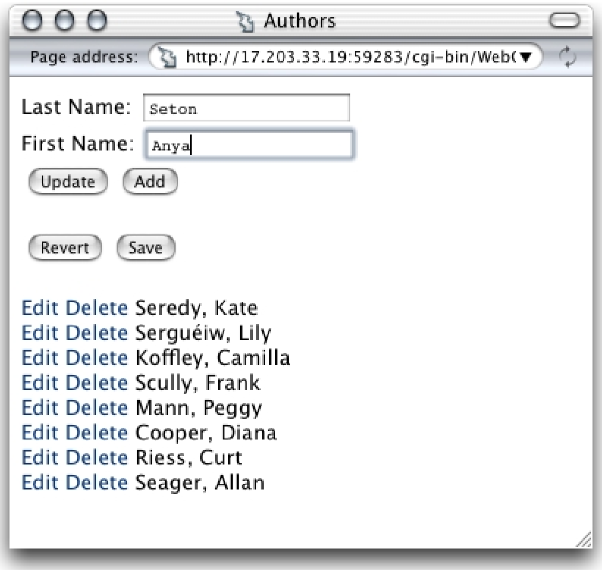
When the user clicks Revert, the `revertChanges` method tells the editing context to discard any changes made since the enterprise objects in it were last fetched or saved. However, the `authorList` array isn't tied to the editing context in any way. Therefore, you must retrieve a new list of authors from the object store and assign it to `authorList`, so that the user sees up-to-date information. (The previous list is garbage-collected by the Java runtime after it is no longer referenced by variables in your application.)

9. Save `Main.java`.

Running the Authors Application

Figure 10-5 shows the Authors application after the names of some authors have been entered.

Figure 10-5 The Authors application



Last Name: Seton
First Name: Anya

Edit Delete Seredy, Kate
Edit Delete Serguéiw, Lily
Edit Delete Koffley, Camilla
Edit Delete Scully, Frank
Edit Delete Mann, Peggy
Edit Delete Cooper, Diana
Edit Delete Riess, Curt
Edit Delete Seager, Allan

There is only one instance of Main throughout the application's execution (all the actions return `null`). When Main is created, it reads the authors from the database and stores them in the `authorList` instance variable. As the user makes changes, `authorList` (and its editing context) is updated. The WORepetition element displays the contents of `authorList` and links so that the user can edit or delete a particular author. Changes are saved when the user clicks Save.

Notice that all the complexities normally required when dealing with databases have been replaced with the straightforward use of enterprise objects.

Browsing the Database

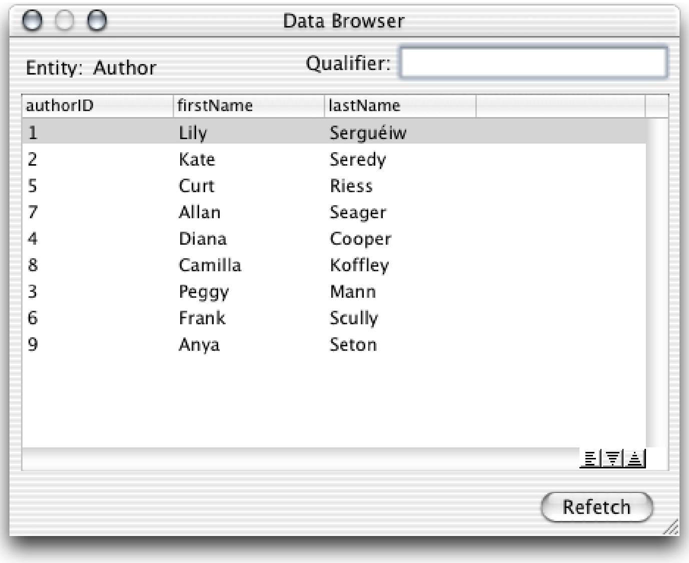
Browsing the raw data in a database can help you to see what Enterprise Objects does for you and can serve as a debugging tool. EOModeler has the ability to browse tables and perform basic filtering, which is useful during application development. This simple facility lets you get a "behind the scenes" look at your data.

1. Open the Authors model in EOModeler.

   Double-click `Authors.eomodeld` under the Resources group in the Groups & Files list of Project Builder's main window.

2. Select the Author entity.
3. Choose Tools > Data Browser.

Figure 10-6 The Data Browser window in EOModeler

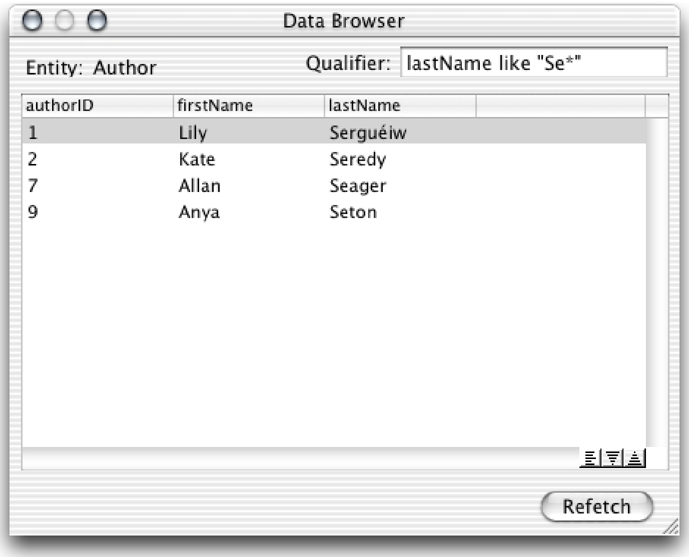


authorID	firstName	lastName
1	Lily	Serguéiw
2	Kate	Seredy
5	Curt	Riess
7	Allan	Seager
4	Diana	Cooper
8	Camilla	Koffley
3	Peggy	Mann
6	Frank	Scully
9	Anya	Seton

The Data Browser window shows the data stored in the AUTHOR table. The Refetch button allows you to refresh this data on demand.

EOModeler also lets you perform simple filtering to limit the number of rows displayed, as Figure 10-7 shows.

Figure 10-7 Data Browser using filtering



Entity: Author

Qualifier: lastName like "Se*"

authorID	firstName	lastName
1	Lily	Serguéiw
2	Kate	Seredy
7	Allan	Seager
9	Anya	Seton

Further Exploration

The third parameter of the constructor for an EOFetchSpecification can be an NSArray of EOSortOrderings, which allows you to alter the order in which objects are returned from a fetch. You can examine the EOSortOrdering class using Java Browser.

To create an EOSortOrdering, you specify the attribute to sort on and the selector to be used for sorting. Four selectors are defined in the EOSortOrdering class:

- `CompareAscending`
- `CompareDescending`
- `CompareCaseInsensitiveAscending`
- `CompareCaseInsensitiveDescending`

The case-insensitive versions of the ascending and descending selectors are for use with strings; they ignore the case of characters when sorting.

A fetch specification created with a sort ordering in place might look like Listing 10-8.

Listing 10-8 Fetch specification that uses sort orderings

```
// Sort ascending by lastName ignoring case.
EOSortOrdering lastNameSort = new EOSortOrdering("lastName",
EOSortOrdering.CompareCaseInsensitiveAscending);

// Sort ascending by firstName ignoring case.
EOSortOrdering firstNameSort = new EOSortOrdering("firstName",
EOSortOrdering.CompareCaseInsensitiveAscending);

// Create an array with the two sort orderings.
NSMutableArray sortOrderings = new NSMutableArray();
sortOrderings.addObject(lastNameSort);
sortOrderings.addObject(firstNameSort);

// Create the fetch specification.
EOFetchSpecification authorFetch = new EOFetchSpecification("Author", null,
sortOrderings);
```

C H A P T E R 1 1

Using Custom Enterprise Objects

The previous chapter taught you how easy it is to manipulate database rows by representing them as EOGenericRecords. Now, instead of using generic enterprise objects, you create an enterprise-object class and customize its behavior.

In this chapter, you

- generate a custom Java class
- add custom logic to an enterprise object class
- learn how to set default values for enterprise objects's properties

Generating a Custom Class

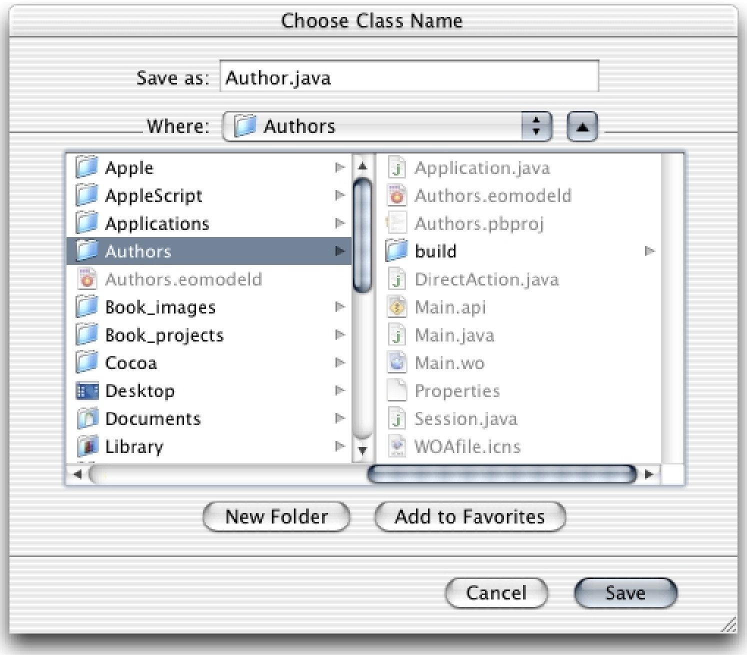
The first step in customizing the behavior of enterprise objects is the generation of custom classes from entity definitions in your model. You continue working on the Authors project created in "Creating an Enterprise Objects Application" (page 129), including the Authors model.

EOModeler generates a Java class based on the attributes of entities as they're defined in the model.

Generating a Java Class From a Model Entity

To generate a Java class representing the Author entity, perform the following steps:

1. Double-click the `Authors.eomodeld` model file in the Resources group to open it in EOModeler.
2. In EOModeler, select the Author entity.
3. Choose Property > Generate Java Files.
4. Navigate to your project's directory and click Save.

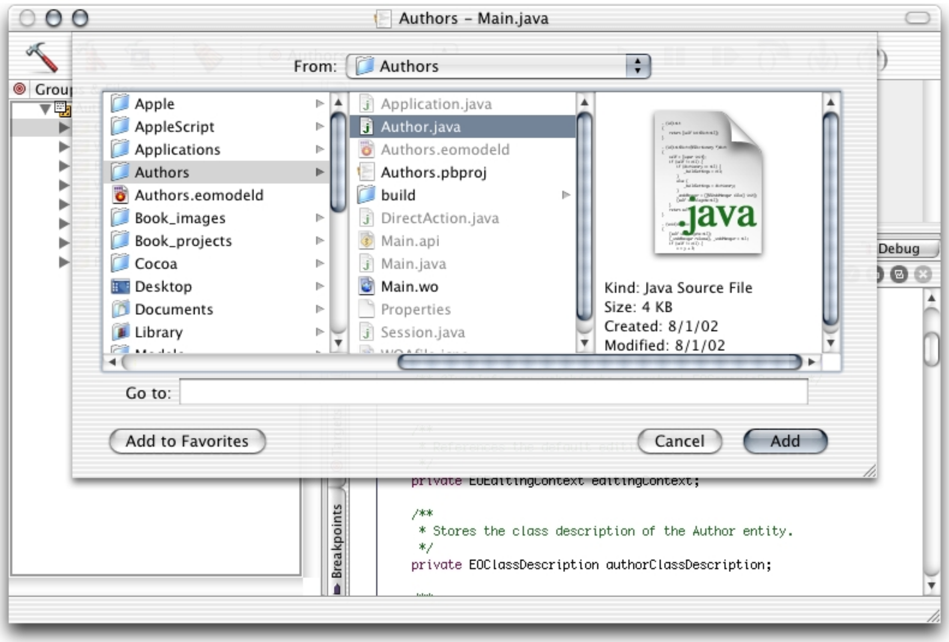


5. Close `Authors.eomodeld`.

Adding a Java Class to the Project

To add `Author.java` to the project, follow these steps:

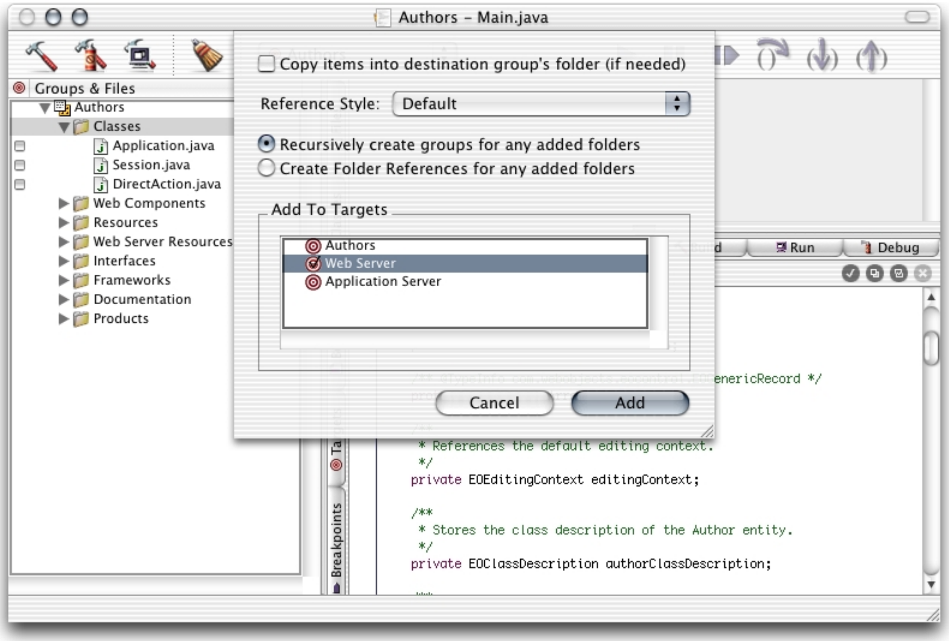
1. In Project Builder, select Classes in the Groups & Files list.
2. Choose Project > Add Files.
3. Select `Author.java` in your project's directory and click Open.



Kind: Java Source File
Size: 4 KB
Created: 8/1/02
Modified: 8/1/02

```
private EOEditingContext editingContext;

/**
 * Stores the class description of the Author entity.
 */
private EOClassDescription authorClassDescription;
```

4. Select the Application Server target and click Add.



[ ] Copy items into destination group's folder (if needed)

Reference Style: Default

[✓ Recursively create groups for any added folders]
Create Folder References for any added folders

Add To Targets
- Authors
- [✓ Web Server]
- Application Server

```
 * References the default editing context.
 */
private EOEditingContext editingContext;

/**
 * Stores the class description of the Author entity.
 */
private EOClassDescription authorClassDescription;
```

EOModeler generates the code in Listing 11-1.

Listing 11-1 `Author.java` generated by EOModeler

```
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import java.math.BigDecimal;
import java.util.*;

public class Author extends EOGenericRecord {
    public Author() {
        super();
    }

    public String firstName() {
        return (String)storedValueForKey("firstName");
    }
    public void setFirstName(String value) {
        takeStoredValueForKey(value, "firstName");
    }

    public String lastName() {
        return (String)storedValueForKey("lastName");
    }

    public void setLastName(String value) {
        takeStoredValueForKey(value, "lastName");
    }
}
```

The Java file generated by EOModeler contains accessor methods for the attributes declared to be class properties.

There are a few things in particular to notice about this file:

- The Author class extends the EOGenericRecord class. This way, all the default behavior of EOGenericRecord is present in your class.
- There are no instance variables in the Author class. Instead, attribute values are accessed via accessor methods, which use key-value coding to get and set the attributes' values.

  If you add code that modifies data to a custom Java class and you want the changes to be stored in the database, you should use these accessor methods to set the values of the appropriate attributes.

Modifying the Authors Project

The code in the Author project is written to work with EOGenericRecords. To take advantage of the Author class, you need to alter the definitions of variables and methods that interact with Author objects in your program, making them Author objects rather than EOGenericRecords.

Make the following changes to the `Main.java` class:

1. Change the `@TypeInfo` line above the `authorList`'s definition so that it reads

   ```
   /** @TypeInfo Author */
   ```

2. Change the class of the `author` and `authorItem` instance variables from EOGenericRecord to Author.
3. Delete the `authorClassDescription` instance-variable definition and its assignment in the constructor.

   You needed the class description only to create new instances of EOGenericRecord with the correct type information obtained from the model file. Now that you are using the Author class, the class description is not needed.
4. Change the constructor, `addAuthor`, and `updateAuthor` methods to create a new Author object instead of a new EOGenericRecord object.

   ```
   author = new Author();
   ```

5. Save `Main.java`.

After making those changes, `Main.java` should look similar to Listing 11-2.

Listing 11-2 The `Main.java` file modified to use Author class instead of EOGenericRecord

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;

public class Main extends WOComponent {
    protected Author author;
    protected Author authorItem;

    /** @TypeInfo Author */
    protected NSMutableArray authorList;

    /**
     * References the default editing context.
     */
    private EOEditingContext editingContext;
```

```
    /**
     * Stores the fetch specification used to retrieve Authors
     * from the data store.
     */
    private EOFetchSpecification fetchSpec;

    public Main(WOContext context) {
        super(context);

        // Build the fetch specification.
        fetchSpec = new EOFetchSpecification("Author", null, null);

        // Get the editing context.
        editingContext = session().defaultEditingContext();

        // Fetch authors.
        authorList = new
NSMutableArray(editingContext.objectsWithFetchSpecification(fetchSpec));

        // Create an Author object (where form data is stored).
        author = new Author();
    }

    public WOComponent addAuthor() {
        // Add the author only when it isn't in the list.
        if (!authorList.containsObject(author)) {
            // Add the author to the list.
            authorList.addObject(author);

            // Insert author into editing context.
            editingContext.insertObject(author);

            // Create a new author.
            author = new Author();
        }

        return null;
    }

    public WOComponent deleteAuthor() {
        // Remove the Author object in authorItem from authorList.
        authorList.removeObject(authorItem);

        // Get authorItem's editing context.
        EOEditingContext ec = authorItem.editingContext();

        // Remove the author from the editing context.
        ec.deleteObject(authorItem);

        return null;
    }

    public WOComponent editAuthor() {
        // Set the author to edit to the one the user selected.
        author = authorItem;

        return null;
    }

    public WOComponent revertChanges() {
        // Discard the changes made in the editing context.
        editingContext.revert();

        // Refetch.
        authorList = new
NSMutableArray(editingContext.objectsWithFetchSpecification(fetchSpec));

        return null;
    }

    public WOComponent saveChanges() {
        // Save the changes made in the editing context to the object store.
        editingContext.saveChanges();

        return null;
    }

    public WOComponent updateAuthor() {
        // Create an Author object.
        author = new Author();
        return null;
    }

}
```

No further changes need to be made for the application to run just as before. Because Author is a subclass of EOGenericRecord, it still responds to the keypaths in the WOD file of the Main component. Build and run the application to confirm it.

Adding Custom Logic

Now that the Author entity is represented by the Author class, you can add custom methods to it.

Frequently, you'll want to display data in a form different from that used to record it in the database. For example, it would be convenient to have a single method in the Author class that returns an author's full name, last name first with a comma separating the last and first names. A similar technique is used in the Authors application (two WOStrings separated by a comma), but putting the logic into a single method allows you to easily suppress the comma if the first name is not present.

Add the `fullName` method shown in Listing 11-3 to the `Author.java`.

Listing 11-3 The `fullName` method in `Author.java`

```
/**
 * Creates this author's full name from <code>lastName</code>
 * and <code>firstName</code>.
 *
 * @return this author's full name.
 */
public String fullName() {
    String firstName = firstName();
    String lastName = lastName();
    String fullName;
    if ((firstName != null) && (!(firstName.equals(""))) {
        fullName = lastName + ", " + firstName;
    }
    else {
        fullName = lastName;
    }

    return fullName;
}
```

Save `Author.java`.

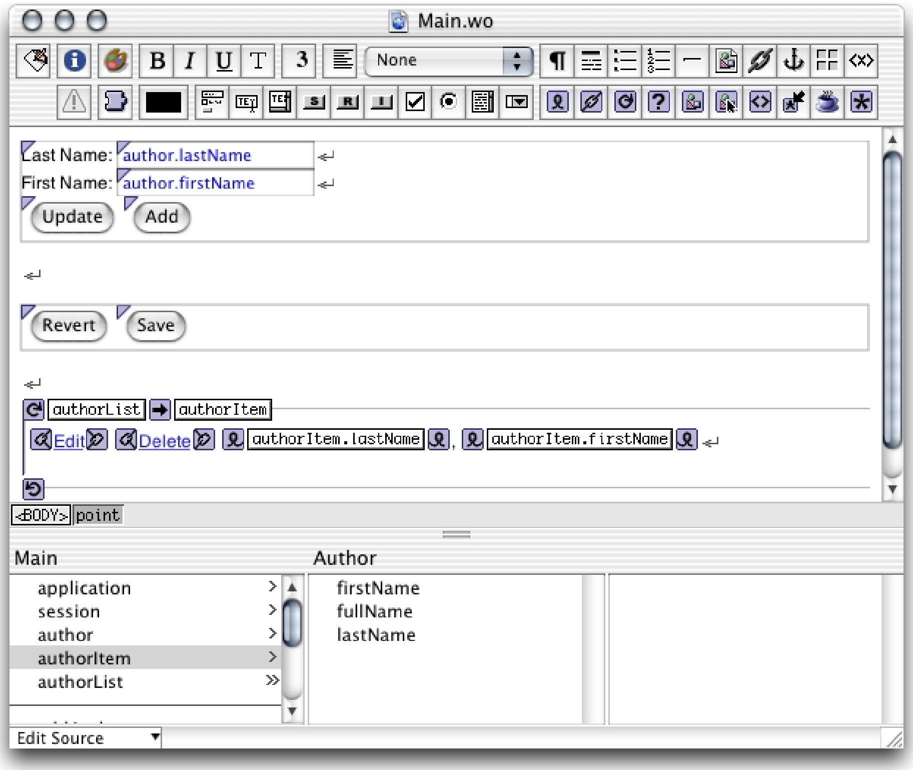
Using Custom Logic

In this section you modify `Main.wo` to use the `fullName` derived attribute of the Author enterprise object (`Author.java`) to display an author's full name.

1. Open the `Main.wo` component in WebObjects Builder.

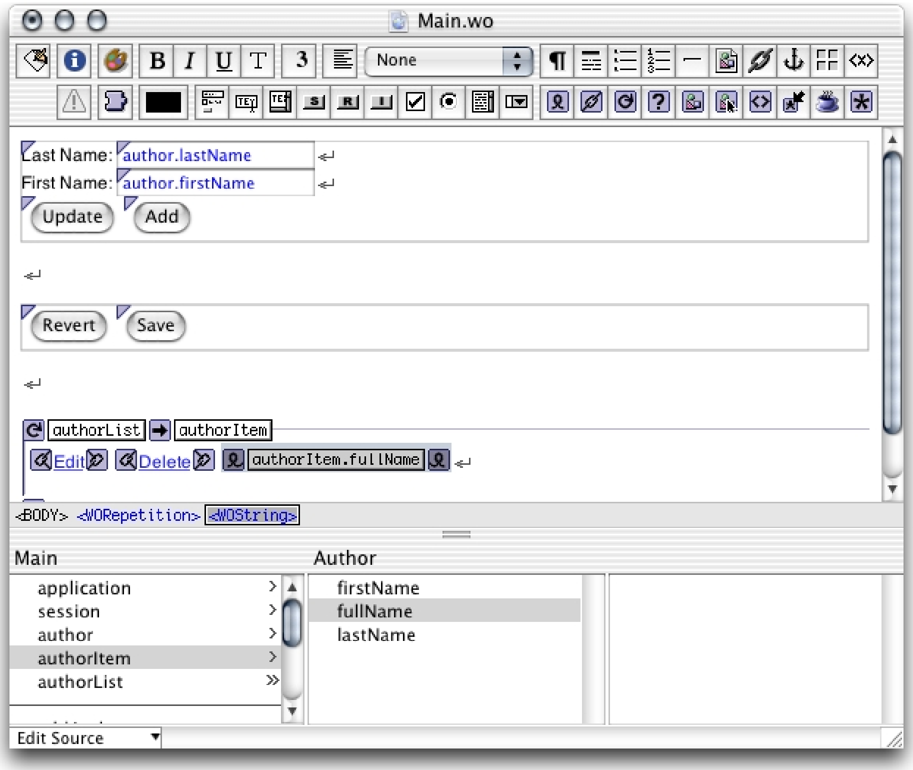
   Figure 11-1 shows that WebObjects Builder recognizes the type of `authorItem` as Author. Also, a browser for the Author class appears next to the browser for the Main class. Notice that the new method, `fullName`, is represented as an attribute of the `authorItem` variable.

Figure 11-1 The `Main.wo` component after adding the `fullName` derived attribute to `Author.java`



2. Remove the comma and the WOString that displays `authorItem.firstName`.
3. Bind the remaining WOString's `value` attribute to `authorItem.fullName`. You can now drag to perform the binding because WebObjects Builder has more information about `authorItem` than when it was an EOGenericRecord. Your component should look like Figure 11-2.

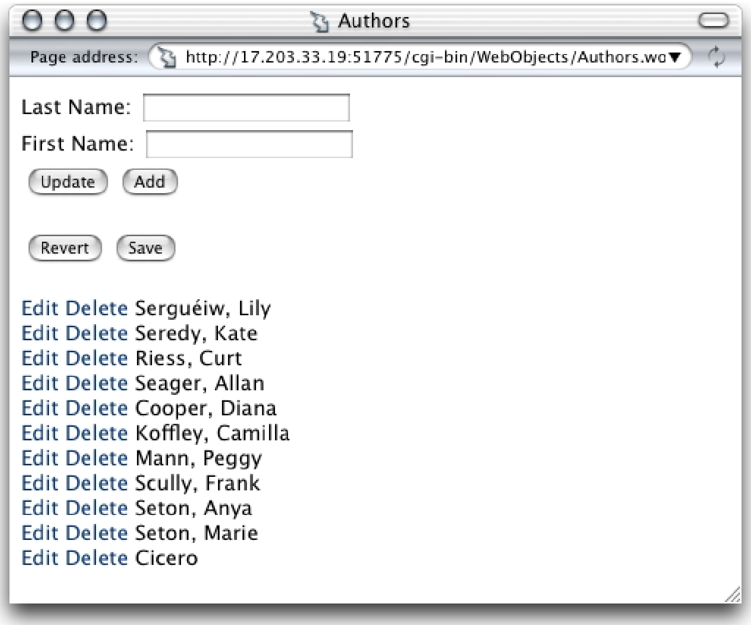
Figure 11-2 The `Main.wo` component using the `fullName` derived attribute



4. Save `Main.wo`, and build and run the application.

   Figure 11-3 shows that when an author's first name is missing, the comma is not displayed, as it was before.

Figure 11-3 The Authors application using the `fullName` method to display author information



http://17.203.33.19:51775/cgi-bin/WebObjects/Authors.wo

Last Name:

First Name:

Edit Delete Serguéiw, Lily
Edit Delete Seredy, Kate
Edit Delete Riess, Curt
Edit Delete Seager, Allan
Edit Delete Cooper, Diana
Edit Delete Koffley, Camilla
Edit Delete Mann, Peggy
Edit Delete Scully, Frank
Edit Delete Seton, Anya
Edit Delete Seton, Marie
Edit Delete Cicero

Setting Default Values

When an Author enterprise object is instantiated, its `firstName` and `lastName` attributes have no value. Sometimes default values should be provided for attributes of your enterprise objects. There are several ways to do accomplish this.

You could assign initial values in the same method that creates the new instance. Do this by simply invoking the `setLastName` and `setFirstName` methods on the new instance with the appropriate arguments. One advantage of this approach is that it allows you to create new instances with different defaults depending on certain circumstances.

Another option is to provide initial values in the Author class itself, so that no values need to be set when the instance is created.

Note: You can also combine these methods—setting a default in the class and overriding it in the specific cases that require it.

In this example, you set initial values in the Author class. Modify the constructor in `Author.java` so that it looks like Listing 11-4.

Listing 11-4 The constructor in `Author.java`—setting default value for `lastName`

```
public Author() {
    super();

    // Set default value for lastName.
    setLastName("*required*");
}
```

When you build and run the application, "*required*" appears in the Last Name text field whenever an Author enterprise object is created.

C H A P T E R 1 2

Working With Relationships

Relationships between entities are an integral part of developing Enterprise Objects applications. In this chapter you explore how to implement the relationships described in "Relationships" (page 117) by adding a Book entity and creating to-one and to-many relationships between Author and Book.

In this chapter, you

- use EOModeler to add the Book entity to the Authors model
- use EOModeler to add the BOOK table to the Authors database
- use EOModeler to create relationships between the Author and Book entities
- traverse relationships using Enterprise Objects
- construct a fetch specification
- perform an in-memory sort

Completing the Authors Model

To complete the Authors model, you add the Book entity to it. After defining the entity's attributes, you add the BOOK table to the Authors database. Then you add the relationships between Author and Book.

Defining the Book Entity

This section shows you how to create the Book entity and define its attributes, including its primary and foreign keys.

1. Open the Authors model in EOModeler by double-clicking `Authors.eomodeld` in the Resources group of Project Builder's Group & Files list.
2. Create a new entity.
    a. Choose Property > Add Entity.
    b. Choose Tools > Inspector.
    c. Enter `Book` in the Name and Class text fields.
    d. Enter `BOOK` in the Table Name text field.
3. Add and configure Book's attributes.

    The Book entity has one major attribute, `title`, which stores a book's title. It also needs a primary-key attribute, `bookID`, to ensure that all the rows in the BOOK table are unique. Finally, it requires an additional attribute, a foreign key, which is used to link a book to its author. This last attribute is named `authorID`.

    Add the `title` attribute by following these steps:

    a. Make sure the Book entity is selected in entity list.
    b. Add a new attribute and name it `title`.
    c. Enter `TITLE` as the column name.
    d. Enter `char` as the external type.
    e. Choose String as the internal data type.
    f. Enter `50` in the External Width text field.
    g. Select the Allow Null Value option in the Advanced Attribute Inspector.

    Add the `bookID` attribute:

    a. Add a new attribute and name it `bookID`.
    b. Enter `BOOK_ID` as the column name.
    c. Enter `int` as the external data type.
    d. Choose Integer as the internal data type.
    e. Make sure the Allow Null Value option is not selected.

Add the `authorID` attribute (this is the foreign key that relates a book to its author):

   a. Add a new attribute and name it `authorID`.

   b. Enter `AUTHOR_ID` as the column name.

   c. Enter `int` as the external type.

   d. Choose Integer as the internal data type.

   e. Make sure the Allow Null Value option is not selected.

4. Select the primary-key attribute for the Book entity.

   a. In the `bookID` row of the Book Attributes list, click in the column with a key as its heading so that a key appears in the row.

   b. Click in the diamond column of the `bookID` row so that the diamond disappears (the value of the `bookID` attribute is not relevant to the application).

5. Make `authorID` a hidden attribute.

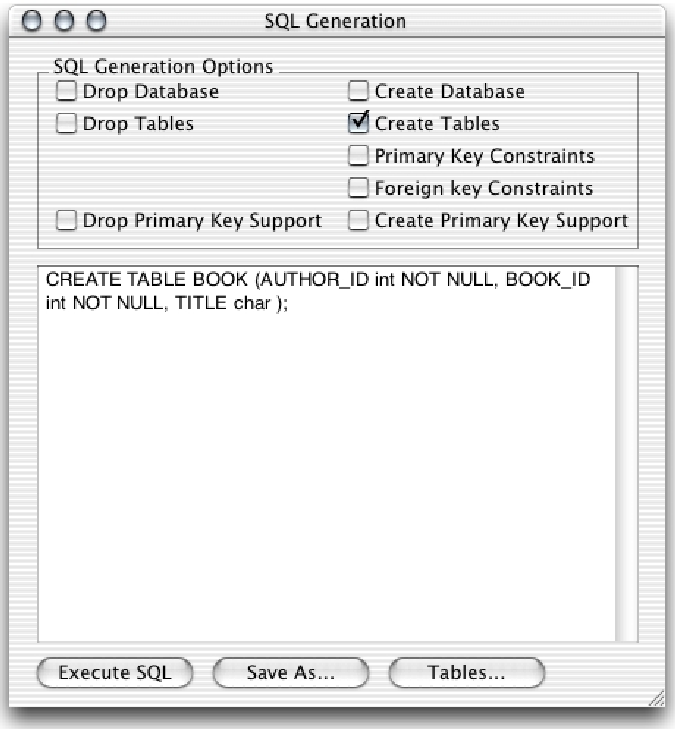
   As for `bookID`, the value of `authorID` is of no interest to the application.

   Click in the diamond column of the `authorID` row in the Book Attributes list, so that the diamond disappears.

Creating the BOOK Table

In this section, you create the BOOK table, just like you created the AUTHOR table in "Creating the AUTHOR Table" (page 144).

1. Select the Book entity from the entity list.
2. Choose Property > Generate SQL.
3. Make sure that only the Create Tables option is selected.



```
CREATE TABLE BOOK (AUTHOR_ID int NOT NULL, BOOK_ID
int NOT NULL, TITLE char );
```

4. Click Execute SQL.

Defining the Model's Relationships

Now that the Book entity is defined, you relate it to the Author entity.

The relationship between Author and Book is bidirectional. Each author can have many books, while each book has only one author.

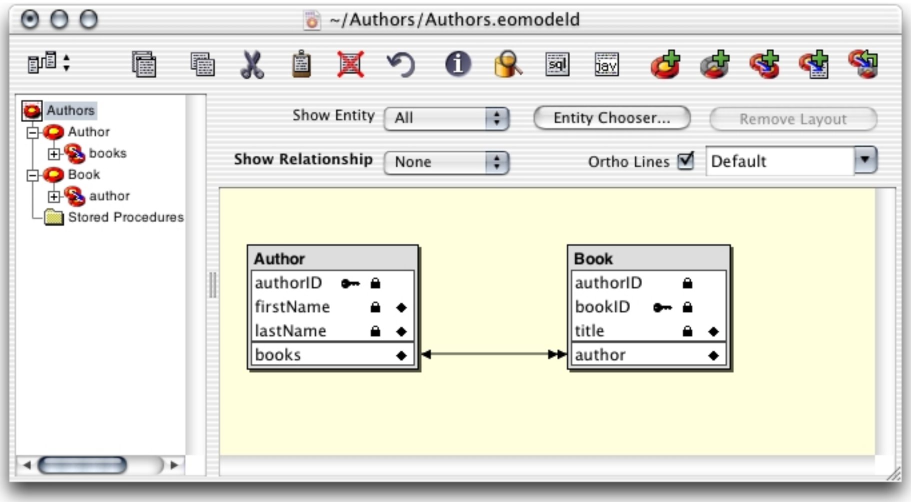
Create the relationships by following these steps:

1. Choose Tools > Diagram View.
2. Control-drag from `Author.authorID` to `Book.authorID`.

This creates two relationships: a to-many relationship from Author to Book, using `authorID` as the linking attribute; and a to-one relationship from Book to Author, again using `authorID` as the linking attribute.

Figure 12-1 graphically depicts the two relationships. Book is linked to Author by a single-headed arrow, meaning that a book can have one author. Whereas Author is linked to Book by a double-headed arrow, meaning that an author can have more than one book.

Figure 12-1 Relationships in the Authors model



Deletion can become complex due to the relationships between entities. For example, if you delete an Author object, what should happen to the Book objects associated with it? You can define the behavior you desire by using delete rules in your model.

What Are Delete Rules?

Each relationship has a **delete rule** that tells Enterprise Objects what to do when you try to delete the source object. These are the possible behaviors:

- **Nullify.** Delete the object and nullify any relationships that point back to it from other entities. (The value of the foreign-key property in target objects is set to `null`.)
- **No action.** Delete the object and perform no other action.
- **Cascade.** Delete the object and all the objects that are targets of the relationship (child objects).
- **Deny.** Do not delete the object if child objects exist. This rule is typically used when child-object deletion should receive special processing before the parent is deleted.

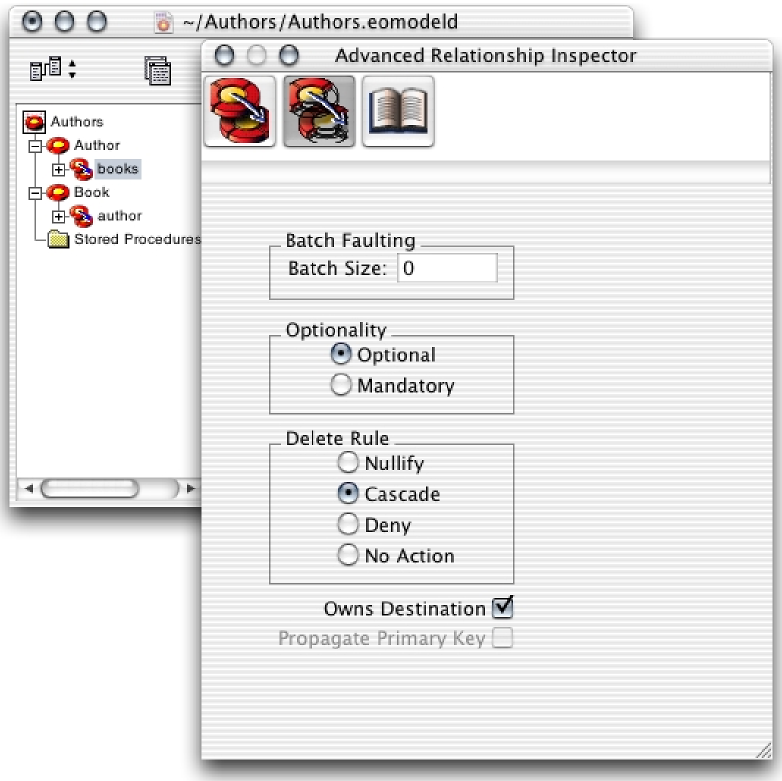
Delete rules are one part of the referential-integrity checks that Enterprise Objects can perform for you. **Referential integrity** refers to the rules governing the consistency of relationships and data. For more information on referential integrity and delete rules, see *Inside WebObjects: Using EOModeler*.

Delete Rules in the Authors Model

In the case of deletion of a book, it makes the most sense to delete the book and remove it from the Author entity's `books` relationship. This is an example of the Nullify delete rule. If you examine the `author` relationship of the Book entity in the Advanced Relationship Inspector, you see that it is already configured with the Nullify delete rule selected. Therefore, you don't need to alter it. However, that default is not appropriate when an author is deleted.

Follow these steps to configure the `books` relationship of the Authors entity so that all of an author's books are deleted when the author is removed from the object store:

1. Select the `books` relationship of the Author entity.
2. Open the Inspector.
3. Display the Advanced Relationship Inspector.
4. Select Cascade as the delete rule.
5. Select the Owns Destination option.



Batch Faulting
Batch Size: 0

Optionality
[✓ Optional] Mandatory

Delete Rule
Nullify [✓ Cascade] Deny No Action

Owns Destination [✓]
Propagate Primary Key [ ]

6. Save `Authors.eomodeld`.

Using Relationships in Your Code

In this section you add to the Authors application the ability to maintain an author's books.

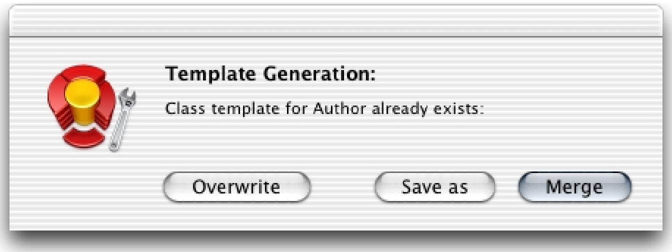
Before you can begin, you need to add the Java classes for Author and Book to your project. Because you've customized `Author.java`, you need to merge the new code generated by EOModeler with your own.

Update the Project's `Author.java` File

First, generate the `Author.java` file that includes the `books` relationship.

1. In EOModeler, select the Author entity in the entity list.
2. Choose Property > Generate Java Files.

   EOModeler tells you that `Author.java` already exists and asked how you wish to proceed.



Template Generation:

Class template for Author already exists:

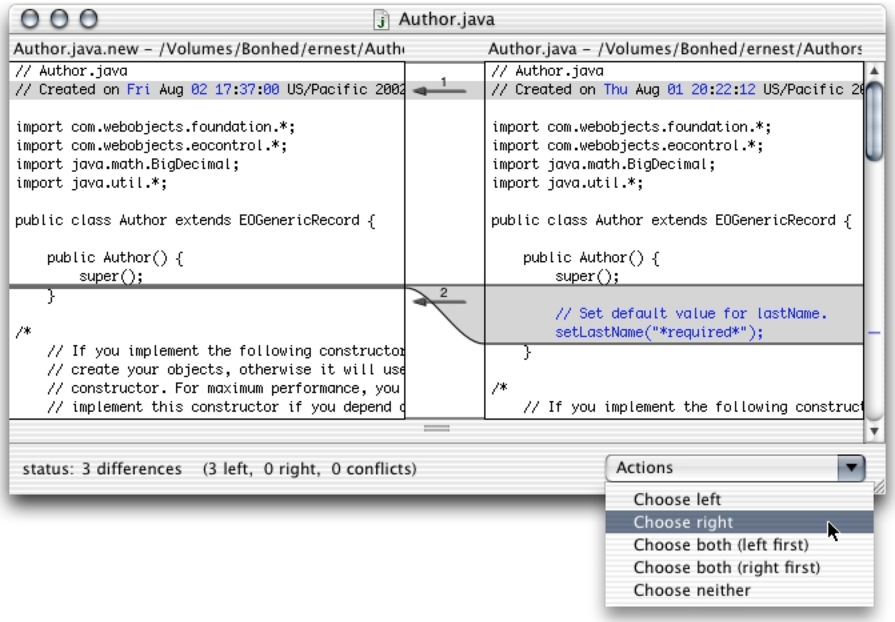
   Click Merge. The FileMerge application launches.

Now, merge the new `Author.java` file generated by EOModeler with the `Author.java` file of the Authors project.

FileMerge outlines and separates the differences between the two files. The file generated by EOModeler is on the left, and the one in the project, which you customized, is on the right. You should see three differences between the two files:

1. The comment line that indicates the file's creation time. You can ignore this.
2. The invocation of the `setLastName` method in the constructor of the file you customized.

   To keep this change in the new version of `Author.java`, click the left-pointing arrow, and choose "Choose right" from the Actions pop-up menu.


Author.java
Author.java.new – /Volumes/Bonhed/ernest/Auth
Author.java – /Volumes/Bonhed/ernest/Authors
1
2
Actions
Choose left
Choose right
Choose both (left first)
Choose both (right first)
Choose neither

```
// Author.java
// Created on Fri Aug 02 17:37:00 US/Pacific 2002

import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import java.math.BigDecimal;
import java.util.*;

public class Author extends EOGenericRecord {

    public Author() {
        super();
    }

/*
    // If you implement the following constructor
    // create your objects, otherwise it will use
    // constructor. For maximum performance, you
    // implement this constructor if you depend o
```

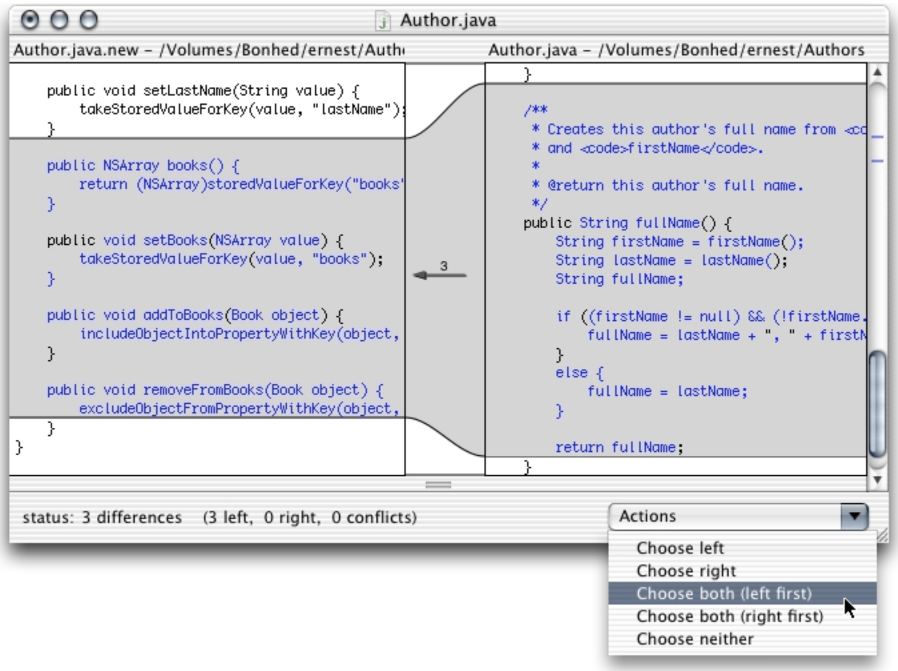
```
// Author.java
// Created on Thu Aug 01 20:22:12 US/Pacific 20

import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import java.math.BigDecimal;
import java.util.*;

public class Author extends EOGenericRecord {

    public Author() {
        super();

        // Set default value for lastName.
        setLastName("*required*");
    }

/*
    // If you implement the following construct
```

status: 3 differences (3 left, 0 right, 0 conflicts)

The arrow should now be pointing right, indicating that the three lines highlighted in the `Author.java` file on the right side of the window will be added to the merged file.

3. The methods that implement the `books` relationship on the left and the `fullName` method on the right are changes that must be present in the merged `Author.java` file.

   To keep both changes, click the left-pointing arrow, and choose "Choose both (left first)" form the Actions pop-up menu.


Author.java
Author.java.new – /Volumes/Bonhed/ernest/Auth
Author.java – /Volumes/Bonhed/ernest/Authors
3
Actions
Choose left
Choose right
Choose both (left first)
Choose both (right first)
Choose neither

```
    public void setLastName(String value) {
        takeStoredValueForKey(value, "lastName");
    }

    public NSArray books() {
        return (NSArray)storedValueForKey("books'
    }

    public void setBooks(NSArray value) {
        takeStoredValueForKey(value, "books");
    }

    public void addToBooks(Book object) {
        includeObjectIntoPropertyWithKey(object,
    }

    public void removeFromBooks(Book object) {
        excludeObjectFromPropertyWithKey(object,
    }
}
```

```
    }

    /**
     * Creates this author's full name from <co
     * and <code>firstName</code>.
     *
     * @return this author's full name.
     */
    public String fullName() {
        String firstName = firstName();
        String lastName = lastName();
        String fullName;

        if ((firstName != null) && (!firstName.
            fullName = lastName + ", " + firstN
        }
        else {
            fullName = lastName;
        }

        return fullName;
    }
```

status: 3 differences (3 left, 0 right, 0 conflicts)

The FileMerge window has two panes. You can reveal the second pane by dragging the divider up. That pane shows you how the merged file will look when you save it. You can edit this pane if the merge operation wasn't performed to your satisfaction. For example, FileMerge might have not added a closing brace (curly bracket) to the `removeFromBook` method.

After performing the appropriate corrections, choose File > Save Merge.

Add the `Book.java` file to the Authors Project

1. In EOModeler, select the Book entity in the entity list.
2. Choose Property > Generate Java Files.
3. Save `Book.java` in your project's directory.
4. Add `Book.java` to the project.

(For details on adding a custom Java class to a project, see "Adding a Java Class to the Project" (page 163).)

To-One Relationships in Java

The `author` method in Listing 12-1 implements the `author` relationship, a to-one relationship from Book to Author.

Listing 12-1 The methods that implement the `author` relationship in `Book.java`

```
    public Author author() {
        return (Author)storedValueForKey("author");
    }

    public void setAuthor(Author value) {
        takeStoredValueForKey(value, "author");
    }
}
```

Enterprise Objects follows the procedure described in "To-One Relationships" (page 118) when you access the `author` property of Book objects.

To-Many Relationships in Java

The code in Listing 12-2 implements the to-many relationship between Author and Book, `books`.

Listing 12-2 The methods that implement the `books` relationship in `Author.java`

```
    public NSArray books() {
        return (NSArray)storedValueForKey("books");
    }

    public void setBooks(NSMutableArray value) {
        takeStoredValueForKey(value, "books");
    }
    public void addToBooks(Book object) {
        includeObjectIntoPropertyWithKey(object, "books");
    }

    public void removeFromBooks(Book object) {
        excludeObjectFromPropertyWithKey(object, "books");
    }
}
```

Notice that the books associated with a particular author can be retrieved as an NSArray simply by calling the `books` method. The NSArray returned is an array of Book enterprise objects. Changes made to them are automatically tracked by the editing context and are saved when `saveChanges` is called. Further, a book can be added to or removed from an author's list using the two provided methods, `addToBooks` and `removeFromBooks`. In that case, however, the editing context has to be notified of the change.

At the database level, the AUTHOR_ID column in the BOOK table corresponds to the AUTHOR_ID of the owning AUTHOR row. Adding a book to an author's array is actually a matter of setting AUTHOR_ID on the new BOOK row to the same value as the author's AUTHOR_ID. When you use the `addToBooks` method, Enterprise Objects takes care of updating the value of the `authorID` attribute of the Book enterprise object for you.

Create the AuthorBookEdit Component

In this section you create the component that allows your application's users to maintain the books of a specific author.

AuthorBookEdit.wo

After performing the steps below, your `AuthorBookEdit.wo` component should look like Figure 12-2 (page 188).

1. Add the AuthorBookEdit component to the project.

   This component allows editing the list of books an author has written.

   a. Select Web Components from the Groups & Files list.

   b. Choose File > New File.

   c. Under WebObjects, select Component and click Next.

   d. Name the component `AuthorBookEdit` and click Finish.

2. Design `AuthorBookEdit.wo`.

   a. Open `AuthorBookEdit.wo` in WebObjects Builder.

   b. Add the `author` key to the component, choose Author as its type, and include accessor methods.

   c. Add the `bookItem` key to the component, choose Book as its type, and do not include accessor methods.

   d. Add an action called `deleteBook` that returns `null`.

   e. Add an action called `addBook` that returns `null`.

   f. Add an action called `returnToMain` that returns an object of type Main.

   g. Enter the label "`Books by `", add a WOString after it, and press Shift-Enter.

   h. Select the line containing the label and the WOString, and choose Elements > Heading > H3.

   i. Bind the WOString to `author.fullName`.

   j. Add a WORepetition element.

      Bind the `list` attribute to `author.books`.

      Bind the `item` attribute to `bookItem`.

   k. Add the WORepetition's content.

      Add a WOHyperlink, a space character, and a WOTextField inside the WORepetition and press Shift-Enter.

      Enter `Delete` as the WOHyperlink's caption, and bind its `action` attribute to `deleteBook`.

      Bind the WOTextField's `value` attribute to `bookItem.title`.

   l. Add two WOSubmitButtons below the WORepetition.

      Enter "`Done`" for the `value` attribute of the first WOSubmitButton, and bind its `action` attribute to `returnToMain`.

      Enter "`Add`" for the `value` attribute of the second WOSubmitButton, and bind its `action` attribute to `addBook`.

m. Add a WOForm element that encompasses all the elements you've added so far.

   Select all the elements by choosing Edit > Select All.

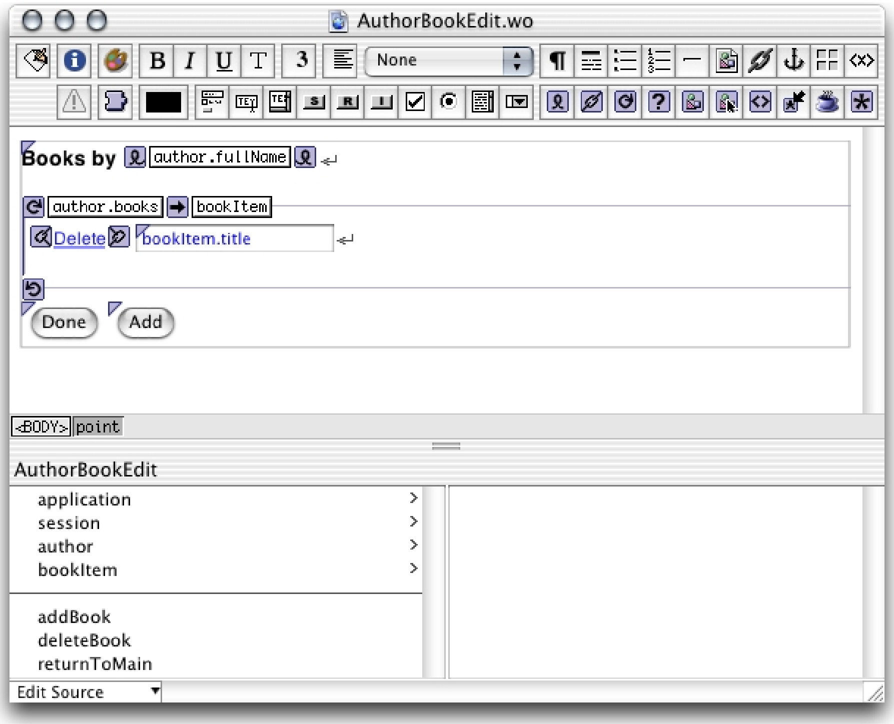
   Choose Forms > WOForm.

   Click anywhere inside a WOForm where there are no elements.

   Choose `true` for the `multipleSubmit` attribute of the WOForm in the WOForm Binding Inspector.

3. Save `AuthorBookEdit.wo`.

Figure 12-2 The `AuthorBookEdit.wo` component in WebObjects Builder



AuthorBookEdit.java

The Java code of the AuthorBookEdit component needs to implement the procedures required to add and delete books from the books relationship of Author enterprise objects. However, all you have to do is maintain an array of books, in the same way as you would manipulate an NSArray (add an object to the array to add a book, and remove objects from it to delete books). The only additional code you need to include is to notify the editing context of the changes made to the array.

1. Modify the deleteBook method so that it looks like Listing 12-3.

Listing 12-3 The deleteBook method in AuthorBookEdit.java

```
public WOComponent deleteBook() {
    // Get editing context from book.
    EOEditingContext ec = bookItem.editingContext();

    // Delete book from its editing context.
    ec.deleteObject(bookItem);

    // Remove book from relationship.
    author.removeObjectFromBothSidesOfRelationshipWithKey(bookItem,
"books");

    return null;
}
```

The deleteBook method first removes the book from the books array of author. It then notifies the editing context that the enterprise object in question should be deleted the next time changes are saved.

When the page refreshes, the book in question is no longer displayed in the list because it has been removed from the books array by the removeObjectFromBothSidesOfRelationshipWithKey method.

2. Modify the addBook method so that it looks like Listing 12-4.

Listing 12-4 The `addBook` method in `AuthorBookEdit.java`

```
public WOComponent addBook() {
    // Get the default editing context.
    EOEditingContext ec = session().defaultEditingContext();

    // Create a Book object.
    Book newBook = new Book();
    newBook.setTitle("New Book");

    // Insert the new book into the editing context.
    ec.insertObject(newBook);

    // Add the book to Author.books relationship and set its author.
    author.addObjectToBothSidesOfRelationshipWithKey(newBook, "books");

    return null;
}
```

The `addObjectToBothSidesOfRelationshipWithKey` method takes care of adding the new book to the `books` array of `author`, as well as setting the `author` attribute for the new book. Alternatively, you could have set each relationship individually, as shown in here:

```
author.addToBooks(newBook);
newBook.setAuthor(author);
```

Modify `Session.java`

The major change you need to make to the `Session.java` class is to add the `authorList` instance variable. Each Session object also needs to fetch the list of authors during its creation.

1. Cut the `fetchSpec` instance variable definition from `Main.java` and paste it in `Session.java`:

   ```
   private EOFetchSpecification fetchSpec;
   ```

2. Edit the constructor so that it matches Listing 12-5.

Listing 12-5 The constructor in `Session.java`

```
public Session() {
    super();

    // Build the fetch specification.
    fetchSpec = new EOFetchSpecification("Author", null, null);

    // Fetch authors.
    fetchAuthorList();
}
```

3. Add the `fetchAuthorList` method in Listing 12-6.

Listing 12-6 The `fetchAuthorList` method in `Session.java`

```
/**
 * Fetches authors from the object store.
 */
public void fetchAuthorList() {
    // Get default editing context.
    EOEditingContext ec = defaultEditingContext();

    // Fetch.
    authorList = new
NSMutableArray(ec.objectsWithFetchSpecification(fetchSpec));
}
```

4. Add the `addAuthor` method in Listing 12-7. (You can copy and paste the `addAuthor` method in `Main.java` and make the necessary modifications.)

Listing 12-7 The `addAuthor` method in `Session.java`

```
/**
 * Adds an author to authorList and to the default editing context.
 */
public boolean addAuthor(Author author) {
    boolean authorAdded = false;
```

```
        // Add only if the author is not already in the list.
        if (!authorList.containsObject(author)) {
            // Add author to authorList.
            authorList.addObject(author);

            // Insert author into editing context.
            defaultEditingContext().insertObject(author);

            authorAdded = true;
        }

        return authorAdded;
    }
```

5. Add the `deleteAuthor` method in Listing 12-8. (You can copy and paste the `deleteAuthor` method in `Main.java` and make the necessary modifications.)

Listing 12-8 The `deleteAuthor` method in `Session.java`

```
    /**
     * Removes an author from authorList and
     * from the default editing context.
     */
    public void deleteAuthor(Author author) {
        // Remove author from authorList.
        authorList.removeObject(author);

        // Get the author's editing context.
        EOEditingContext ec = author.editingContext();

        // Remove author from the editing context.
        ec.deleteObject(author);
    }
```

6. Save `Session.java`.
7. Add the `authorList` instance variable to the `AuthorBookEdit.wo` component.
    a. In WebObjects Builder, select session from the AuthorBookEdit browser.
    b. Control-click in the Session browser and choose Add Key to Session.

c. Name the key `authorList`, set its type to a mutable array of Author, and generate accessor methods.

Modify the `Main.wo` Component

The `Main.wo` component needs to display the AuthorBookEdit page, so that the application's users can edit an author's books. To accomplish that, a WOHyperlink and its action need to be added to `Main.wo`.

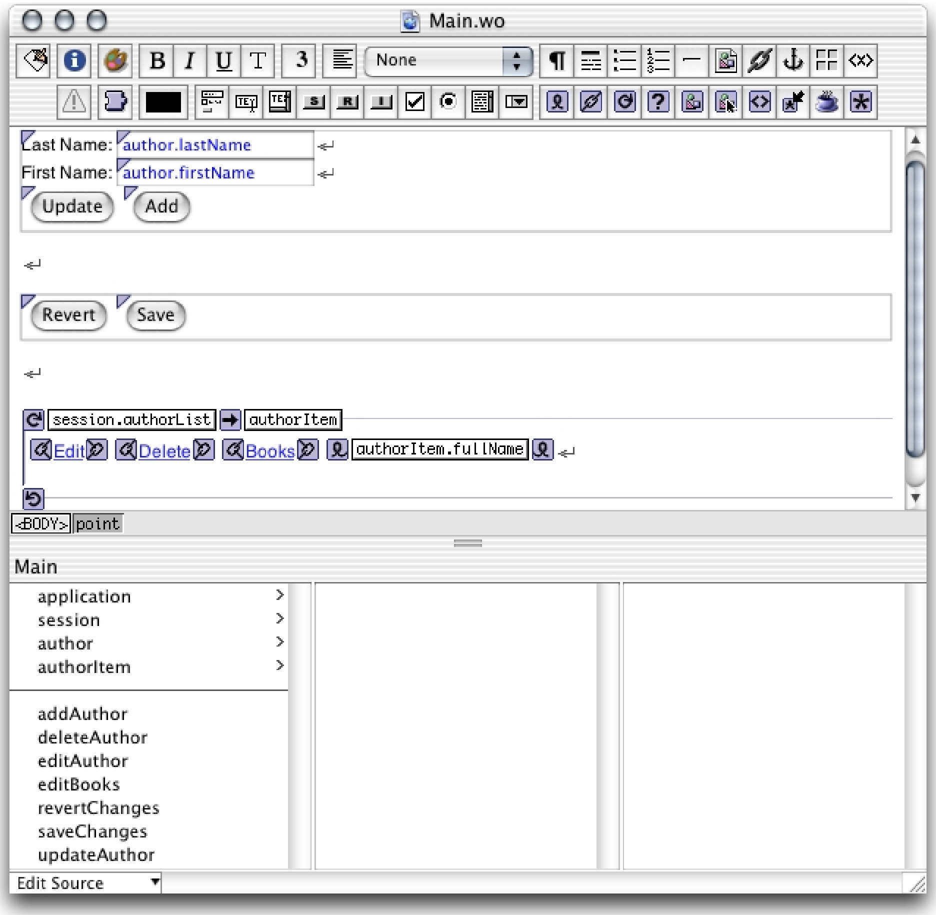
`Main.wo`

A new action, `editBooks`, needs to be added to the component. It also needs a new WOHyperlink, which invokes the new action. After completing the required changes, `Main.wo` should look like Figure 12-3 (page 194).

1. Open `Main.wo` in WebObjects Builder.
2. Add a new action named `editBooks` that returns AuthorBookEdit.
3. Add a WOHyperlink between the Delete WOHyperlink and the WOString inside the WORepetition.

   Set the caption to `Books` and bind its `action` attribute to `editBooks`.
4. Bind the WORepetition's `list` attribute to `session.authorList`.
5. Delete the `authorList` key.
   a. Select `authorList` in Main's browser.
   b. Choose "Delete authorList" from the Edit Source pop-up menu.
6. Save `Main.wo`.

Figure 12-3 `Main.wo` with the `editBooks` action and the Books WOHyperlink



`Main.java`

1. Add a session instance variable.

```
/**
 * Stores the Session object.
 */
private Session session;
```

2. Edit the constructor so that it looks like Listing 12-9.

Listing 12-9 The constructor in `Main.java`

```
public Main(WOContext context) {
    super(context);

    // Get the session.
    session = (Session)session();

    // Get the default editing context.
    editingContext = session.defaultEditingContext();

    // Create an Author object (where form data is stored).
    author = new Author();
}
```

3. Edit the `addAuthor` method so that it looks like Listing 12-10.

Listing 12-10 The `addAuthor` method in `Main.java`—uses the `addAuthor` method in `Session.java`

```
public WOComponent addAuthor() {
    if (session.addAuthor(author)) {
        // Create an Author object.
        author = new Author();
    }

    return null;
}
```

4. Edit the `deleteAuthor` method so that it looks like Listing 12-11.

Listing 12-11 The `deleteAuthor` method in `Main.java`—uses the `deleteAuthor` method in `Session.java`

```
public WOComponent deleteAuthor() {
    session.deleteAuthor(authorItem);
```

```
        return null;
    }
```

5. Edit the `editBooks` method so that it looks like Listing 12-12.

Listing 12-12 The editBooks method in Main.java—sends Author object to AuthorBookEdit component

```
public AuthorBookEdit editBooks() {
    AuthorBookEdit nextPage =
(AuthorBookEdit)pageWithName("AuthorBookEdit");

    nextPage.setAuthor(authorItem);

    return nextPage;
}
```

The `editBooks` method needs to send the author whose books are to be edited to the AuthorBookEdit component (the next page to be displayed).

6. Edit the `revertChanges` method so that it matches Listing 12-13.

Listing 12-13 The `revertChanges` method in `Main.java`—uses default editing context and the `fetchAuthorList` method in `Session.java`

```
public WOComponent revertChanges() {
    // Discard the changes made in the editing context.
    editingContext.revert();

    // Refetch.
    session.fetchAuthorList();

    return null;
}
```

This method now uses the session's `fetchAuthorList` method because the author list is stored in the session, not the Main component.

7. Save `Main.java`.

Run the Application

Build and run the application. When the user clicks an author's Book link, Main's `editBook` method creates an AuthorBookEdit object and tells it which author it is to process by invoking its `setAuthor` method.

The AuthorBookEdit component displays the books associated with the author by iterating through the `books` relationship of `author` (an NSMutableArray) in the WORepetition. When the user clicks Add, the `addBook` method creates a new Book object and adds it to the `books` relationship of `author` and the editing context. Similarly, when the user clicks Delete for a book in the list, the book is removed from the `books` relationship and deleted from the editing context. When the user is done editing the books of the author, she clicks Done to return to the Main page.

Deleting Authors

When the user deletes an author, she doesn't get a warning telling her that all the books related to that author are going to be deleted as well. In this section, you add a component that displays such a warning.

Though conceptually more complex, the design and implementation of the logic for deleting authors is just as simple as that for books. The only significant difference is that the application asks the user for confirmation before deleting the author, because this action has the side effect of removing additional objects from the object store that the user may not be aware of.

You add a component that displays the author that the user wants to delete, along with all related books, and asks for confirmation. If the user changes her mind, she's returned to the Main page. If the user clicks Delete, the author and related books are deleted from the editing context (the actual delete transaction takes place when the user clicks Save on the Main page).

Create the ConfirmAuthorDelete Component

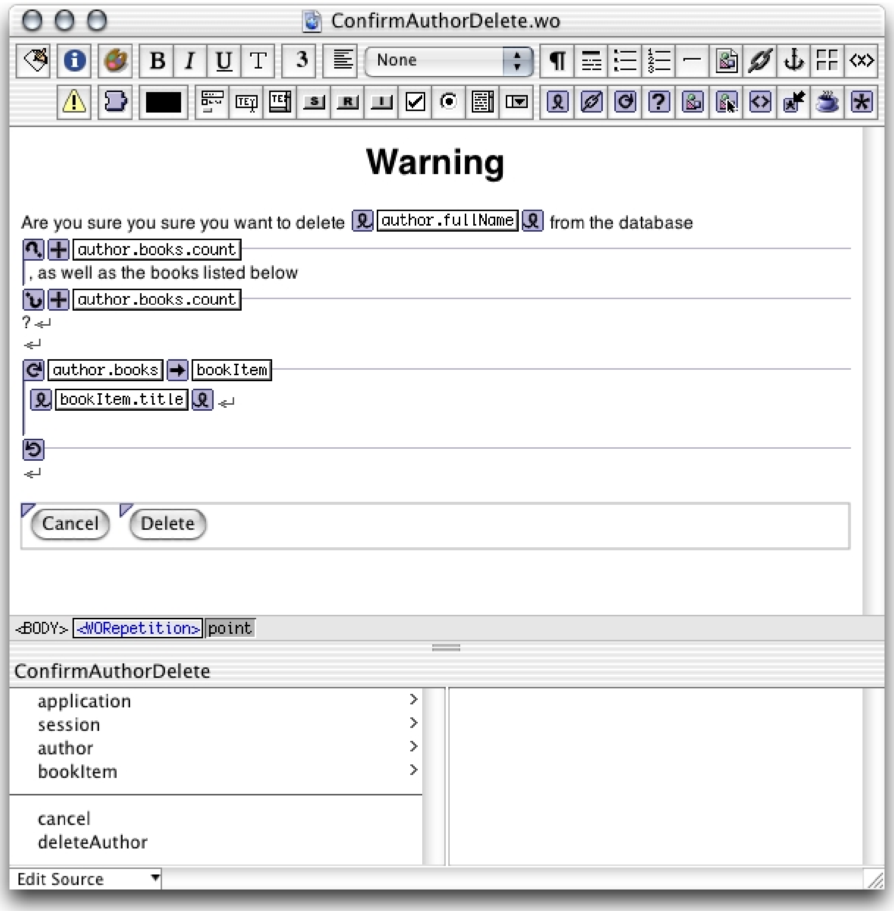
When you're done with the steps below, the component should look like Figure 12-4 (page 200).

1. Add a new component and name it `ConfirmAuthorDelete`.

   (See "Defining a New Component" (page 89) for details.)
2. Open `ConfirmAuthorDelete.wo` in WebObjects Builder.
3. Add the following instance variables:
   a. `author` (Author), with accessor methods
   b. `bookItem` (Book), without accessor methods
4. Add the following actions:
   a. `cancel` (Main)
   b. `deleteAuthor` (Main)
5. Add a warning heading:
   a. Enter `Warning` in the content pane and press Return.
   b. Select the word "Warning" and choose Elements > Heading > H1.
   c. ChooseFormat > Alignment > Align Center.
6. Add the warning line:
   a. Enter `Are you sure you want to delete`.
   b. Add a space character, a WOString, and a second space character.
   c. Enter `from the database`.
   d. Add a WOConditional.
   e. Enter "`, as well as the books listed below`" inside the WOConditional.
   f. Add a question mark after the WOConditional and two line-feed characters (press Enter or Shift-Return two times).
   g. Bind `author.fullName` to the WOString.
   h. Bind `author.books.count` to the WOConditional's `condition` attribute.

7. Add the WORepetition that lists an author's books:
   a. Add a WORepetition after the second line-feed character.
   b. Add a WOString and a line-feed character inside the WORepetition.
   c. Add a line-feed character after the WORepetition.
   d. Bind `author.books` to the WORepetition's `list` attribute.
   e. Bind `bookItem` to the WORepetition's `item` attribute.
   f. Bind `bookItem.title` to the WOString's `value` attribute.
8. Add the Cancel and Delete WOSubmitButtons inside a WORepetition:
   a. Add a WOForm after the last line-feed character.
   b. Set the `multipleSubmit` attribute of the WOForm to `true`.
   c. Add two WOSubmitButtons separated by a space inside the WOForm.
   d. Set the value attribute of the first WOSubmitButton to `"Cancel"`.
   e. Set the value attribute of the second WOSubmitButton to `"Delete"`.
   f. Bind the `cancel` action to the Cancel button.
   g. Bind the `deleteAuthor` action to the Delete button.

Figure 12-4 The `ConfirmAuthorDelete.wo` component



Save the `ConfirmAuthorDelete.wo` component.

Modify `ConfirmAuthorDelete.java`

Edit the `deleteAuthor` method so that it looks like Listing 12-14.

Listing 12-14 The `deleteAuthor` method in `ConfirmAuthorDelete.java`

```
public Main deleteAuthor() {
    Main nextPage = (Main)pageWithName("Main");

    // Get the session.
    Session session = (Session)session();

    // Delete the author form the session’s author list.
    session.deleteAuthor(author);

    return nextPage;
}
```

Modify `Main.java`

The Main component needs to display the ConfirmAuthorDelete component when its `deleteAuthor` action is invoked. You accomplish that by modifying the `deleteAuthor` method in `Main.java` so that it looks like Listing 12-15.

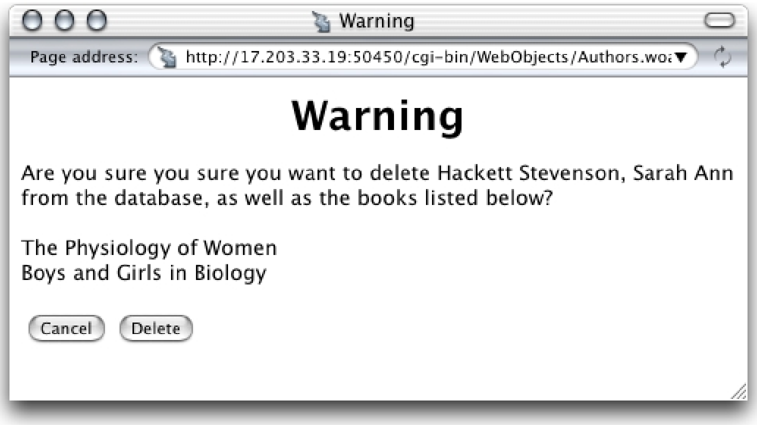
Listing 12-15 The `deleteAuthor` method in `Main.java`—returns the ConfirmAuthorDelete component

```
public ConfirmAuthorDelete deleteAuthor() {
    // Create an instance of the ConfirmAuthorDelete component.
    ConfirmAuthorDelete nextPage =
(ConfirmAuthorDelete)pageWithName("ConfirmAuthorDelete");

    // Set ConfirmAuthorDelete’s author.
    nextPage.setAuthor(authorItem);

    return nextPage;
}
```

Run the Application

Build and run the application. Create a new author, add several books, and save your changes. (You can use EOModeler to browse the tables's contents and confirm that the new information has been added to the database.) Click Delete for the newly added author. You should be presented with a confirmation page similar to Figure 12-5.

Figure 12-5 The ConfirmAuthorDelete page in action



Warning

Are you sure you sure you want to delete Hackett Stevenson, Sarah Ann from the database, as well as the books listed below?

The Physiology of Women
Boys and Girls in Biology

If you click Cancel, you are simply returned to the Main page. Clicking Delete causes the `deleteAuthor` method in `ConfirmAuthorDelete.java` to be invoked. In turn, it invokes the session's `deleteAuthor` method, which removes the author from the `authorList` array and adds it to the editing context's list of enterprise objects to delete.

Sorting a Fetch

Fetch specifications can be created either programmatically in an application or with EOModeler and stored in the model file. Up to this point you have used a simple fetch specification, returning an unsorted list of enterprise objects. Now, you create a more elaborate fetch specification. To learn how to create fetch specifications in EOModeler, see *Inside WebObjects: Using EOModeler*.

First, you define the new fetch specification on the Author entity. Then, you amend the code in `Session.java` to use the new fetch specification. Finally, you use in-memory sorting to keep the list sorted even after you add new authors.

"Further Exploration" (page 158), explains how to sort the list of authors during the fetch. If you did so, you probably noticed that once you added authors to the list, the list didn't stay sorted. You can use Enterprise Objects's sorting mechanism on a previously fetched array of enterprise objects, as well as during a fetch.

In this section you add a new method to `Session.java`, `sortAuthorList`, which sorts the `authorList` array. You also modify the `addAuthor` method in `Main.java` to invoke the `sortAuthorList` method to re-sort the `authorList` array each time an author is added.

1. Add the `sortAuthorList` method, shown in Listing 12-16, to `Session.java`.

Listing 12-16 The `sortAuthorList` method in `Session.java`

```
/**
 * Sorts <code>authorList</code> by last name, ascending.
 */
public void sortAuthorList() {
    // Create array to store sort orderings.
    NSMutableArray sortOrderings = new NSMutableArray();

    // Create sort ordering.
    EOSortOrdering sortOrdering = new EOSortOrdering("lastName",
EOSortOrdering.CompareAscending);
```

```
        // Add sort ordering to orderings array.
        sortOrderings.addObject(sortOrdering);

        // Sort authorList using orderings.
        EOSortOrdering.sortArrayUsingKeyOrderArray(authorList,
    sortOrderings);
    }
```

2. Edit the `addAuthor` method in `Session.java` by adding the numbered lines in Listing 12-17.

Listing 12-17 The `addAuthor` method in `Session.java`

```
    /**
     * Adds an author to authorList and to the default editing context.
     */
    public boolean addAuthor(Author  author) {
        boolean authorAdded = false;

        // Add only if the author is not already in the list.
        if (!authorList.containsObject(author)) {
            // Add author to authorList.
            authorList.addObject(author);

            // Sort authorList.                                          //1
            sortAuthorList();                                            //2

            // Insert author into editing context.
            defaultEditingContext().insertObject(author);

            authorAdded = true;
        }

        return authorAdded;
    }
```

Sorting is performed only when an item is added to the author list.

Build and run the application. The author list is sorted each time you add an author. Note, however, than when the list is first created, it's not sorted. You can add the two lines you added to the `addAuthor` method to `fetchAuthorList`, after the Fetch section. That way, the author list is sorted after each fetch.

Document Revision History

Table A-1 describes the revisions to *Inside WebObjects: Web Applications*.

Table A-1 Document revision history

Date	Notes
September 2002	Document name changed to *Inside WebObjects: Web Applications*.
	Project examples now in `/Developer/Documentation/WebObjects/Web_Applications/projects`.
	Revised for WebObjects 5.2.
May 2001	Document published as *Inside WebObjects: Discovering WebObjects for HTML*.

A P P E N D I X   A

Document Revision History

Glossary

adaptor, database A mechanism that connects your application to a particular database server. For each type of server you use, you need a separate adaptor. WebObjects provides an adaptor for databases conforming to JDBC.

adaptor, WebObjects A process (or a part of one) that connects WebObjects applications to an HTTP server.

application object An object (of the WOApplication class) that represents a single instance of a WebObjects application. The application object's main role is to coordinate the handling of HTTP requests, but it can also maintain application-wide state information.

attribute In Entity-Relationship modeling, an identifiable characteristic of an entity. For example, `lastName` can be an attribute of an Employee entity. An attribute typically corresponds to a column in a database table.

business logic The rules associated with the data in a database that typically encode business policies. An example is automatically adding late fees for overdue items.

CGI (Common Gateway Interface) A standard for communication between external applications and information servers, such as HTTP or Web servers.

class In object-oriented languages such as Java, a prototype for a particular kind of object. A class definition declares instance variables and defines methods for all members of the class. Objects that have the same types of instance variables and have access to the same methods belong to the same class.

class property An instance variable in an enterprise object that meets two criteria: It's based on an attribute in your model, and it can be fetched from the database. A class property can be an attribute, a relationship, a method, or an instance variable.

column In a relational database, the dimension of a table that holds values for a particular attribute. For example, a table that contains employee records might have a LAST_NAME column that contains the values for each employee's last name.

component An object (of the WOComponent class) that represents a Web page or a reusable portion of one.

context An object that encapsulates state information for a given transaction (one cycle of the request-response loop). Context objects are implemented with the WOContext class. They encapsulate information about the URL, context ID, application, session, component, request, and response items.

WebObjects maintains a cache of WOContext objects to support the back function of Web browsers.

cookie General mechanism used by Web servers to store and retrieve persistent data on a client system (Web browser). The information stored is usually state data associated with a range of URLs.

database server A data storage and retrieval system. Database servers typically run on a dedicated computer and are accessed by client applications over a network.

delete rule A delete rule specifies the action to take when the source object of a relationship is deleted from the data store. The possible actions are nullify, cascade, deny, and none. Delete rules are defined in model files with the EOModeler application.

display group A display group collects an array of objects from a data source, sorts it and displays data from the objects in the user interface.

dynamic element A dynamic version of an HTML element. WebObjects includes a list of dynamic elements with which you can build your component.

enterprise object A Java object that conforms to the key-value coding protocol and whose properties (instance data) can map to stored data. An enterprise object brings together stored data with methods for operating on that data.

entity In Entity-Relationship modeling, a distinguishable object about which data is kept. For example, you can have an Employee entity with attributes such as `lastName`, `firstName`, `address`, and so on. An entity typically corresponds to a table in a relational database; an entity's attributes, in turn, correspond to a table's columns.

Entity-Relationship modeling A discipline for examining and representing the components and interrelationships in a database system. Also known as E-R modeling, this discipline factors a database system into entities, attributes, and relationships.

EODisplayGroup A display group that displays data in J2SE or Cocoa interface elements using EOAssociations.

EOModeler A tool used to create and edit models.

fetch In Enterprise Objects applications, to retrieve data from the database server into the client application, usually into enterprise objects.

foreign key An attribute in an entity that gives it access to rows in another entity. This attribute must be the primary key of the related entity. For example, an Employee entity can contain the foreign key `deptID`, which matches the primary key in the entity Department. You can then use `deptID` as the source attribute in Employee and as the destination attribute in Department to form a relationship between the entities.

inheritance In object-oriented programming, the ability of a superclass to pass its characteristics (methods and instance variables) on to its subclasses.

instance In object-oriented languages such as Java, an object that belongs to (is a member of) a particular class. Instances are created at runtime according to the specification in the class definition.

Java Browser A tool used to peruse Java APIs and class hierarchies.

Java Foundation Classes A set of graphical user interface components and services written in Java. The component set is known as Swing.

JDBC An interface between Java platforms and databases.

key An arbitrary value (usually a string) used to locate a datum in a data structure such as a dictionary.

key-value coding The mechanism that allows the properties in enterprise objects to be accessed by name (that is, as key-value pairs) by other parts of the application.

locking A mechanism to ensure that data isn't modified by more than one user at a time and that data isn't read as it is being modified.

many-to-many relationship A relationship in which each record in the source entity may correspond to more than one record in the destination entity, and each record in the destination may correspond to more than one record in the source. For example, an employee can work on many projects, and a project can be staffed by many employees.

method In object-oriented programming, a procedure that can be executed by an object.

model An object (of the EOModel class) that defines, in Entity-Relationship terms, the mapping between enterprise object classes and the database schema. This definition is typically stored in a file created with the EOModeler application. A model also includes the information needed to connect to a particular database server.

object A programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data. Objects are the principal building blocks of object-oriented programs.

primary key An attribute in an entity that uniquely identifies rows of that entity. For example, the Employee entity can contain an `EmpID` attribute that uniquely identifies each employee.

Project Builder A tool used to manage the development of a WebObjects application or framework.

property In Entity-Relationship modeling, an attribute or relationship.

record The set of values that describes a single instance of an entity; in a relational database, a record is equivalent to a row.

referential integrity The rules governing the consistency of relationships.

relational database A database designed according to the relational model, which uses the discipline of Entity-Relationship modeling and the data design standards called normal forms.

relationship A link between two entities that's based on attributes of the entities. For example, the Department and Employee entities can have a relationship based on the `deptID` attribute as a foreign key in Employee and as the primary key in Department (note that although the join attribute `deptID` is the same for the source and destination entities in this example, it doesn't have to be). This relationship would make it possible to find the employees for a given department.

reusable component A component that can be nested within other components and acts like a dynamic element. Reusable components allow you to extend WebObject's selection of dynamically generated HTML elements.

request A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the user's Web browser to a Web server that asks for a resource like a Web page.

request-response loop The main loop of a WebObjects application that receives a request, responds to it, and awaits the next request.

response A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the Web server to the user's Web browser that contains the resource specified by the corresponding request. The response is typically a Web page.

row In a relational database, the dimension of a table that groups attributes into records.

session A period during which access to a WebObjects application and its resources is granted to a particular client (typically a browser). Also an object (of the WOSession class) representing a session.

target A blueprint for building a product from specified files in your project. It consists of a list of the necessary files and specifications on how to build them. Some common types of targets build frameworks, libraries, applications, and command-line tools

table A two-dimensional set of values corresponding to an entity. The columns of a table represent characteristics of the entity and the rows represent instances of the entity.

template In a WebObjects component, a file containing HTML that specifies the overall appearance of a Web page generated from the component.

to-many relationship A relationship in which each source record has zero to many corresponding destination records. For example, a department has many employees.

to-one relationship A relationship in which each source record has exactly one corresponding destination record. For example, each employee has one job title.

transaction A set of actions that is treated as a single operation.

uniquing A mechanism to ensure that, within a given context, only one object is associated with each row in the database.

validation A mechanism to ensure that user-entered data lies within specified limits.

Web server An application that serves Web pages to Web browsers using the HTTP protocol. In WebObjects, the Web server lies between the browser and a WebObjects application. When the Web server receives a request from a browser, it passes the request to the WebObjects adaptor, which generates a response and returns it to the Web server. The Web server then sends the response to the browser.

WebObjects Builder A tool used to graphically edit WebObjects components.

WODisplayGroup A display group that displays data in WebObjects components.

Index

A

accessor methods 73, 165
action attribute 57
action method, example of adding 56
adaptor level 126
adaptors 51, 62
Add Column pop-up menu (EOModeler) 141
addObject method 104
addUser method 108, 110–111
Advanced Attribute Inspector (EOModeler) 138, 176
Allows Null column (EOModeler) 142
API files of components 44
appendToResponse method 67
Application class 35
applications
  editing contexts in 126
  as project type 27
  running 112
  and sessions 100
arrays 103–107
AuthorBookEdit component 186–190
Authors application 129–157
  custom objects in 161–174
  database editing in 129–159
  relationships in 175–205
awake method 65–66

B

Back button 68
backtracking cache 68
blob (Binary Large Object) type 143

C

cache, backtracking 68
Cascade delete rule 180
char type 143
class properties 142
Class Property column (EOModeler) 142
Classes group (Project Builder) 35
client-server applications 23
Column column (EOModeler) 142
columns in databases 116
component action request processing 63
components
  counting number of loads 55–56
  defined 19
  example of adding 89–93
  example of modifying 36, 45–50
  maintaining state in 53–59, 99–113
  parts of 44
  sharing data between 85–98
condition attribute 77
conditional display of elements 77–81
ConfirmAuthorDelete component 198
cookies 23
count method 104
custom classes 86–88, 161–165
custom logic 169–173

D

Data Browser (EOModeler) 156
database level 127
databases 115–119
  accessing 18
  browsing in EOModeler 156
  columns in 116
  connecting to 122

databases (*continued*)
  creating 129
  foreign keys in 118
  legacy 126
  not-null columns in 117
  preferred protocol of 122
  primary keys in 116
  querying languages for 121
  relationships in 117–119
  rows in 116
  structure of 115–117
  tables in 116
  translating data 122
  uniquing in 116
  updates to 127
date type 143
datetime type 143
default values of properties, setting 173
Definition column (EOModeler) 142
delete rules 179–180
deleteObject method 153
Deny delete rule 180
derived properties 81–84, 171–172
development tools 23–24
dictionaries 122, 126
direct action request processing 63
DirectAction class 35
display groups 44
Documentation group (Project Builder) 35
double type 143
dynamic publishing 20–22, 41–60

E

Edit Source menu (WebObjects Builder) 54, 78
editing contexts 122, 129–159
encapsulation of data
  benefits of 112
  custom objects and 85–88
Enterprise Object technology 121–127
enterprise objects 121–127
entities, adding to a model 136–140
Entity Inspector (EOModeler) 136
EOAccess layer 126–127
EOControl layer 125–126
EOFetchSpecification class 151, 158–159
EOGenericRecord class 124, 151, 165, 171
EOKeyValueCoding interface 124
EOModeler application
  browsing tables with 156
  creating fetch specifications 203
  generated code 164
  introduced 24
  main window 141
  opening 132
EOSortOrdering class 158
External Type column (EOModeler) 142–143

F

features of WebObjects 18–23
fetch specifications 151, 158–159, 203–205
FileMerge application 182–185
foreign keys 118, 143
frameworks 27
Frameworks group (Project Builder) 35

G

Groups & Files pane (Project Builder) 34

H

HTML files of components
  defined 44
  editing 53
  reading 51
  relationship to WOD files 50
HTML input elements 61, 69

I

in-memory sorting 203–205
input elements 61, 69
int type 143
invokeAction method 67
item attribute 107, 110

J

Java Browser 104, 158
Java class files of components 35
Java classes
  accessor methods in 165
  adding to project 86, 163
  generating from model entity 162
  in JAR files 37
Java files of components 44, 53
Java Number class 143
Java String class 143
JavaWebObjects framework 35
JDBC (Java Database Connectivity) standard 126
JDBC adaptor 133, 143
JDBC driver 133

K

keypaths 89, 101, 169
key-value coding 124, 165
key-value dictionaries 126

L

legacy databases 126
list attribute 107
loadCount variable 55
Locking column (EOModeler) 142
long type 143

M

Main component 45–50
Main subgroup (Project Builder) 35
models 24, 127
multiple users 100

N

Name column (EOModeler) 142
negate attribute 77
New Project Assistant 26–33
No action delete rule 180
not-null columns in databases 117
NSArray class 103–104
NSData object 143
NSMutableArray class 103–107
Nullify delete rule 180

O

object orientation 19
objectAtIndex method 104
OODBS (object-oriented databases) 14
OpenBase Manager 129

P, Q

page cache 68
Precision column (EOModeler) 142
Primary Key column (EOModeler) 141
primary keys 116, 143
Products group (Project Builder) 35
Project Builder
  getting started with 25–39
  introduced 24
  location of 24
  main window 33–36
  running applications 51

Project Builder Assistant. *See* New Project Assistant
projects
  choosing type of 27
  components of 34
  creating a simple one 25–33
  location for 28
properties of enterprise objects 124
  class properties 142
Prototype column (EOModeler) 143

R

Refetch button (EOModeler) 157
relationships in databases
  data deletion and 179–181, 197–202
  and database design 117–119
  example of creating 175–205
  Java in code 181–196
removeObjectAtIndex method 104
request processing 62–69
  backtracking cache 68
  generating the response 67–68
  models for 63
  request-response loop and 62
  stages of 65–67
request-response loop
  introduced 41
  overview 62
  phases of 64
  response generation and 51–52
  tracing 72–77
Resources group (Project Builder) 35
response generation 51–52
response page 62, 67
revertChanges method 154
rows in databases 116
Run pane (Project Builder) 38, 59

S

saveChanges method 153
scalability in WebObjects 18
Scale column (EOModeler) 143
Session class
  adding arrays to 104–107
  as default class in application 35
session method 101
sessions (Session objects)
  creating new instances 59
  overview of 100
  using to manage state 99–112
setUser method 110
sleep method 68
sorting
  and fetch specifications 203–205
  specifying order 158
state management 53–60, 99–113
static binding 52
submit button 71, 75–77
submitChanges method 98, 110

T

tables
  creating 144
  in databases 116
takeValuesFromRequest method 66
TimeDisplay application 45
to-many relationships 119, 179, 185
tools 23–24
to-one relationships 118, 179, 185

U

uniquing 116
user input
  and derived properties 81–84
  introduced 22
  managing 61–84
  and request processing 62–69
  user interface of 69–72
userEdit action 97
UserEdit component 89–98, 109
users (User objects)
  adding 108, 110
  deleting 108, 111
  editing 109
  multiple 100
  sessions for 100

V

Value Class (Java) column (EOModeler) 142
Value Type column (EOModeler) 143

W, X, Y, Z

web browser cache 68
Web Components group (Project Builder) 35, 45
Web Server Resources group (Project Builder) 35
WebObjects adaptor 51, 62
WebObjects Application project type 27
WebObjects Builder
  adding arrays to Session class 104
  introduced 24
  introduction to using 36
  location of 24
  main window 47
WebObjects Framework project type 27
Width column (EOModeler) 142
WOComponent class 19, 44
  See also components
WOConditional elements 77–81, 94
WOD files
  introduced 44
  merging 52
  relationship to HTML files 50
WOElement class 45
WOHyperlink elements 108, 113
WOO files 44
WORepetition elements 107, 155
WOString Binding Inspector 48
WOString elements 47–50
WOStrings 61, 169
WOTextFields 87
Write Format column (EOModeler) 143