Integrating Your Code Into the Request-Response Loop

Most of your program's activity is a reaction to messages the application sends out during cycles of the request-response loop. These messages travel through the objects of an application: from application to session object, from session to component object, and from a component to its static and dynamic HTML elements. They enable the application to store user input, trigger action methods, and generate a response, usually in the form of an HTML page.

For background on the request-response loop, and on the roles the various WebObjects classes play in request handling, see "How WebObjects Works.".

For WebObjects developers, the emitted messages are *hooks* into the request-response loop allowing them to invoke custom application behavior. You can influence what happens during request handling by taking advantage of these hooks. For example, you can determine what page to return based on user input, modify the header lines of a generated HTTP response, initialize variables from database records, or substitute a page for the requested page.

There are three types of hooks that can be implemented in either scripts or compiled classes:

- *Initialization methods* establish the beginning state and behavior of objects. The **init** method is invoked when one of the objects involved in request handling is created. On the other hand, **awake** is invoked when an object becomes involved in each cycle of request handling.
- *Action methods* are associated with a particular user action such as clicking a button or hyperlink.
- *Request-handling methods* that are invoked at a particular point in the request-response loop if you implement them in a subclass or a script:

  takeValuesFromRequest:inContext:
  invokeActionForRequest:inContext:
  appendToResponse:inContext:

Initialization Methods

The application, session, and component objects in a WebObjects application receive two messages that allow the objects to initialize themselves. The two methods, **init** and **awake**, are invoked in both scripted and compiled instances of WOApplication, WOSession, and WOComponent subclasses. The main difference between the methods is *when* they are invoked.

When init and awake are Sent

When objects are created, they receive an **init** message. The **init** method gives the receiver an opportunity to initialize its state and behavior just after it is created. The initialization is effective over the lifetime of an object; this lifetime varies according to the object's type:

- The WOApplication object exists from the time the application is started until it is terminated (either explicitly or through application time-out).
- The WOSession object persists through a session. As session is a period during which a particular user is accessing the application, and during which resources are allocated accordingly. An application can have multiple concurrent sessions. The application object creates a WOSession object if a request is the first from a user (that is, there is no session ID associated with it). Sessions can be explicitly terminated or can end when a *session time-out*—a period of no user activity—occurs.
- The WOComponent objects of an application are created each time a page is directly requested via URL and each time a **pageWithName:** message is sent to the application object. There are a couple exceptions to this. If page caching is turned on, as it is by default, the session object stores each component instance (page) at the end of each request-response loop. If the user backtracks through a session, the application restores page instances from this cache. Also, if the request component returns **self** or (preferably) **nil** in an action method, the application returns the cached instance of the request component rather than re-creating a new instance.

An object's **awake** method, on the other hand, is invoked at that point in each cycle of the request-response loop that the object begins to participate in request handling. This usually occurs right after **init**, except in those two cases noted above where an page is restored from a cache rather than created. In these cases, the **awake** method is invoked without a prior invocation of the **init** method.

See "How WebObjects Works" for a discussion of **init** and **awake** in the context of the request-response loop.

The Structures of init and awake

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

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

```
}
```

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

```
- awake {
    /* initializations go here */
}
```

The sleep and dealloc Methods

Complementing **awake** and **init**, respectively, are the **sleep** and **dealloc** methods. These methods let objects deallocate their instance variables and perform other clean-up tasks. The **sleep** method is invoked at the end of an object's involvement in a transaction. The **dealloc** method is invoked just before an object is destroyed.

In Objective-C you deallocate instance variables by sending them **release**. In WebScript, on the other hand, all you need to do (in **sleep**) is set the instance variables to **nil**. WebScript has a "garbage-collection" mechanism that automatically disposes of unreferenced objects. For this reason, there's seldom a reason for implementing **dealloc** in a script.

When Use init, When Use awake?

Since both **init** and **awake** are entry points for an object's involvement in request handling, they are both suitable places for initializations. So which method is a better place for this? When is it better to use **init**, and when is it better to use **awake**?

The short answer is that, because objects are typically persistent to some degree, **init** is the better place to initialize an object. A WebObjects application, by default, stores page instances. Those component objects usually persist through a number of transactions (as specified in **setPageCacheSize:**) and are restored when the user backtracks to them. Pages that return **nil** in an action method also restore a cached instance of themselves.

Page caching, however, can impose a penalty in terms of scalability for some applications. If scalability is a problem, you can optimize an application by initializing component instance variables in **awake**. Then, in **sleep**, you can deallocate these variables by setting them to **nil**.

Even when optimizing, however, an important consideration is the cost of initializing operations as offset against the cost of storing page instances. For example, it is sensible to perform static initializations in **awake**, but it is

prohibitive to do database fetches in **awake**. Database and file system operations are expensive and so should not be repeated needlessly.

You might want finer control over page persistence than that afforded by the page-caching mechanism; for instance, you may want the action method invoked by a Submit button to return the most recent instance of a page instead of new page. To achieve this finer control, you can always cache selected pages and component variables in the session object and restore them when these pages and variables are requested. See the chapter "Managing State" for a discussion of storage strategies and techniques.

In the final analysis, what you do in **init** and what you do in **awake** are a matter of common sense, given the object involved and the frequency of invocation. For example, if you want to tally the number of transactions a page is involved in, the component's **awake** method is the logical place to increment a counter. On the other hand, you need to set the session time-out period only once, at the beginning of a session, so the obvious place to do that is in the **init** method of the session object.

Application Initialization

The application **init** method is invoked when the application is launched—either from the command line or by autostarting—and never again. It's common to initialize application variables in an application **init** method. For example, the follow excerpt from the **Application.wos** script in the DodgeDemo example initializes the **models**, **categories**, and **priceRange** application variables.

```
id models, categories, priceRange;
- init {
    id modelSource, categorySource;

    [super init];
    [self logWithFormat:@"Welcome to DodgeDemo!!!"];

    /* code not shown... */
    // Create Model data source and fetch the models
    modelSource = [[[EODatabaseDataSource alloc] initWithEditingContext:
        editingContext entityName:@"Model"] autorelease];
    if (!modelSource) {
        [self logWithFormat:@"Cannot create model data source"];
        [self terminate];
    }
    models = [modelSource fetchObjects];
    // Create Category data source and fetch the categories
    categorySource = [[[EODatabaseDataSource alloc] initWithEditingContext:
```

```
        editingContext entityName:@"Type"] autorelease];
    if (!categorySource) {
        [self logWithFormat:@"Cannot create category data source"];
        [self terminate];
    }
    categories = [categorySource fetchObjects];

     // Price range for price browsers
    priceRange = @(8000, 10000, 12000, 14000, 16000, 18000, 20000, 25000,
        30000, 50000, 90000);

    return self;
}
```

When scripted applications are run, WebObjects automatically creates an instance of a special subclass of WOApplication and adds to it the code from the application script. When you send **init** to **super** in an application script, you invoke the **init** method of the superclass of the instance: WOApplication. You can also create your own subclass of WOApplication and override **init** to perform any necessary initialization. It is more common, however, to implement the **init** method in an application script.

Because all applications are not reparsed after the first time, changing a scripted application **init** method has no effect on a running WebObjects application . To have changes to a scripted application's **init** method take effect, you must restart the application.

In addition to using the application **init** method to initialize application variables, you can also use it to configure the application's behavior. For example, you can set the application time-out period and specify the page-cache size:

```
// Set the number of transactions pages are persistent
[self setPageCacheSize:10];
// Set application timeout
[self setTimeOut:43200];
```

Session Initialization

The session's **init** method is invoked when the application creates the WOSession object for the current session, which happens when the application receives the first request of a new user. Initialize variables in the session **init** method that should retain their values between transactions throughout the session. For example, the **Session.wos** script in the Visitors example initializes the session variable **timeSinceSessionBegain** before setting up a timer that will result in the variable's value being incremented:

```
id timeSinceSessionBegan;
id timer;
- init
{
    [super init];
    timeSinceSessionBegan = 0;
    timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self
         selector:"timeOfSession" userInfo:nil repeats:YES];
    [self setTimeOut:120];
    return self;
}
```

Note: An important side effect of using a timer object in a WebObject's application is that the method invoked when the timer fires is outside the request-response loop. In other words, invocation occurs after the transaction concludes, and thus the method has no access to the WORequest, WOResponse, and WOContext of the transaction.

When a session begins in a scripted application, WebObjects automatically creates an instance of a special subclass of WOSession and adds to it the code from the session script. When you send **init** to **super** in an session script, you invoke the **init** method of the superclass of the instance: WOSession. You can also subclass WOSession and override **init** to perform any necessary initialization. It is more common, however, to implement the **init** method in an session script.

The WOSession object's **awake** method is invoked just after the object is created (and receives **init**) and immediately after being restored for each subsequent transaction.

Component Initialization

It's common in a component's **init** method to initialize component variables. For example, the **Department.wos** script in the EmployeeBook example application uses **init** to initialize the **departments** component variable:

```
id departments;
- init {
    id departmentsPath;

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

The WOComponent class—an abstract class that implements basic component behavior—defines the **init** method for components and implements it to initialize some basic attributes. When a component object must be generated in a scripted application, WebObjects automatically creates an instance of a special subclass of WOComponent and adds to it the code from the component script. When you send **init** to **super** in an component script, you are invoking the **init** method of the superclass of the instance: WOComponent. You can also subclass WOComponent and override **init** to perform any necessary initialization. It is more common, however, to implement the **init** method in a component script.

A component's **init** method is invoked only when the component must be created. This happens at the start of a transaction *except* when the component is restored from the page cache as a result of the user backtracking or a request component returning itself as the response page. Even then, **init** is invoked only in cycles in which the component is participating. Generally, a component participates in a cycle of the request-response loop if:

- It represents the request page—the page associated with the request.
- It represents the response page—the page returned to the server.
- It's nested in either the request or response page.
- It's messaged in any other way during the current cycle.

The **awake** method is immediately invoked in a component after **init** and after each time the component is restored from the page cache. Just as in **init**, you can implement a component **awake** method that initializes component variables. For example, the **Main.wos** script in the CyberWind application uses **awake** to initialize the **options** component variable:

```
- awake {
    options = @("See surfshop information", "Buy a new sailboard");
}
```

You can subclass WOComponent and override **awake** to perform any necessary initialization, but it is more common to implement the **awake** method in a component script.

Action Methods

An action method is a method that's associated with a user action. You associate methods with a user action using a dynamic element. For example, WOSubmitButton has an attribute named **action** to which you can assign a method. When the submit button in the corresponding HTML page is clicked, the action method is invoked in the subsequent cycle of the request-response

loop. This declaration in the HelloWorld application associates the action method **sayHello** with a submit button:

```
SUBMIT_BUTTON: WOSubmitButton {action = sayHello};
```

Clicking the submit button sends a request to the HelloWorld application, initiating a cycle of the request-response loop in which **sayHello** is invoked.

Note: The WOActiveImage, WOHyperlink, and WOForm dynamic elements can also be used to associate action methods to a user action.

Action methods take no arguments and return a page that will be packaged with an HTTP response. For example, the **sayHello** action method of the HelloWorld example is defined as follows:

```
- sayHello
{
    id nextPage = [WOApp pageWithName:@"Hello"];
    [nextPage setNameString:nameString];
    return nextPage;
}
```

As in **sayHello**, most action methods perform page navigation. It is common for action methods to determine the response page based on user input. For example, the following action method returns an error page if the user has entered an invalid part number (stored in the component variable **partnumber**) or an inventory summary otherwise:

```
- showPart {
    id errorPage;
    id inventoryPage;

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

Action methods don't have to return a new page. They can instead direct the application to regenerate the request page. When an action method returns **nil**, the application uses the request component as the response component.

Note: Returning **self** in an action method generally has the same effect as returning **nil**. However, there's a difference when the action method is implemented in a nested component. When a nested component—a component representing only a portion of the request page—returns **self** in an action, the application attempts to use the nested component to generate the response page. Since the component only represents a portion of a page, returning **self** is probably an error. Returning **nil** always has the effect of using the request page—the component representing the whole request page—as the response page. As a result, returning **nil** is considered to be a better practice than returning **self**.

In the Visitors example, the request page is also used as the response page. The WebScript **recordMe** action method records the name of the last visitor and clears the text field:

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

Request-Handling Methods

WebObjects defines three request-handling methods that are invoked at particular points in the request-response loop if you implement them:

- takeValuesFromRequest:inContext:
- invokeActionForRequest:inContext:
- appendToResponse:inContext:

As with **init** and **awake** methods, the request-handling methods can be implemented for an application, a session, and the application's components. All versions of a request-handling method work identically, and can be used for similar purposes. You choose to implement a method for the application, the session, or a component based on which is more appropriate for the behavior you need to provide. Generally, you implement request-handling methods in the application or the session object when the specified behavior should affect every request. You implement request-handling methods in a component when the behavior should affect a particular page.

The WOApplication, WOSession, and WOComponent classes each declare and implement the three methods that handle requests and responses. The implementations of these methods that you provide in scripts—**Application.wos**, **Session.wos**, and component scripts—are dynamically added to the appropriate subclass object that WebObjects generates at run time. Consequently, when WOApplication sends **takeValuesFromRequest:inContext:** to **self**, the **takeValuesFromRequest:inContext:** method defined in the corresponding **Application.wos** (if it exists) is invoked. In compiled subclasses of WOApplication, WOSession, and WOComponent, you can override the request-handling methods, but it is more common to implement them in a script.

In your implementations of request-handling methods you must invoke **super**'s implementation of the same methods. *Where* you invoke it is an important consideration because it can affect the request, response, and context information available at any given point. You will want to perform certain tasks before **super** is invoked and, for other tasks, after **super** is invoked.

takeValuesFromRequest:inContext:

This method is invoked during the phase of the request-response loop when the application stores user input. When this phase concludes, the request component has been initialized with the bindings made in WebObjects Builder or the assignments made in the declarations file.

The first argument to **takeValuesFromRequest:inContext:** is a WORequest object. A WORequest object encapsulates information from an HTTP request such as the method line, request headers, URL, and form values. The second argument is a WOContext object. A WOContext object contains references to information specific to the application, such as the path to the request component's directory, the version of WebObjects that's running, the name of the application, and the name of the request page.

It is common to use this method to access request and context information. For example, the following implementation of **takeValuesFromRequest:inContext:** records the kinds of browsers—user agents—from which requests are made (the "recordUserAgent:" method is assumed to be implemented in the same script):

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

When you invoke **super**'s **takeValuesFromRequest:inContext:** in your implementation of the same method, the application processes user input. So after the message

to **super** is when you could perform postprocessing of user input. For example, the following implementation takes the values for the **street**, **city**, **state**, and **zipCode** variables and stores them in **address** variable formatted as a standard mailing address.

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

invokeActionForRequest:inContext:

The second phase of the request-response loop involves **invokeActionForRequest:inContext:**. This method is invoked, in turn, in the application object, the session object, the request page, and in every dynamic element on that page. Normally, the message is forward from object to object until it is handled by the dynamic element associated with the user action (typically a WOSubmitButton, a WOHyperLink, an WOActiveImage, or WOForm).

A common use of this "hook" in **Appliation.wos**, **Session.wos**, or a component script to return a page other than the one requested. A scenario where this might occur is when the user requests a page which has a dependency on another page that the user must fill out first. The user might finish ordering items from a catalog application and want to go to a fulfillment page; but first he or she must supply credit card information.

The following **invokeActionForRequest:inContext:** method, implemented in **Session.wos**, returns a "CreditCard" page if the user hasn't supplied this information yet:

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

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

When the application receives a request for a new page (say, a fulfillment page), the session determines whether or not the user has supplied valid credit-card data by checking the value of its **verified** variable. If the value of **verified** is NO, the session returns the "CreditCard" component. As shown in the following action method, the "CreditCard" component sets the **verified** session variable to YES when the user has supplied valid credit information and returns the user to the original request page to try again.

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

Limitations on Direct Requests

By specifying a page in a URL, a user can attempt to access any page in an application without invoking an action. For example, you can access the second page of HelloWorld without invoking the **sayHello** action by opening the URL:

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

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

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

appendToResponse:inContext:

This method is invoked in the phase of the request-response loop during which the application generates HTML for the response page. You can implement this method to add to the response content or otherwise manipulate the HTTP response. For example, you can add or modify the HTTP headers. The following code excerpt sets the "Expires" header in the HTTP response to "0."

```
- appendToResponse:aResponse inContext:aContext
{
    [super appendToResponse:aResponse inContext:aContext];
    [aResponse setHeader:@"0" forKey:@"Expires"];
}
```

The first argument to **appendToResponse:inContext:** is a WOResponse object. A WOResponse object encapsulates information contained in the generated HTTP response such as the status, response headers, and response content. The second argument is a WOContext object. A WOContext object contains references to application-specific information such as the path to the request component's directory, the version of WebObjects that's running, the name of the application, and the name of the request page.

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

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

- appendToResponse:aResponse inContext:aContext
{
  id aString = [value description];
  [super appendToResponse:aResponse inContext:aContext];
  [aResponse appendContentHTMLAttributeValue:@"<p>"];
    if (isBold) {
        [aResponse appendContentHTMLAttributeValue:@"<b>"];
    }
    if (isItalic) {
        [aResponse appendContentHTMLAttributeValue:@"<i>"];
    }

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

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

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

```
- appendToResponse:response inContext:context {
   [super appendToResponse:response inContext:context];
```

```
    [[self session] terminate];
}
```

The WOSession method **terminate** schedules the destruction of state associated with the current session, but termination is deferred until the current transaction concludes. You can explicitly terminate a session anytime, anywhere in a WebObjects application.

Summary

What request-response loop "hooks" can you implement?

There are three types of methods that allow your application to influence what happens in the request-response loop:

- *Initialization methods*—**init** and **awake**— are invoked, respectively, when an object is created and just before the receiver begins to participate in each cycle of the request-response loop. The methods **dealloc** and **sleep** allow the deallocation of variables initialized in **init** and **awake**.
- *Action methods* are associated with a particular user action such as clicking a button or hyperlink.
- *Request–handling methods*, if implemented, are invoked in application, session, and component objects at particular points in the request-response loop.

You can participate in the request-response loop by implementing any of these methods.

What you can use the hooks for?

The following list summarizes common uses of the request-response loop "hooks":

- Action methods perform page navigation.
- Application **init** and **awake** methods are places to initialize application variables and configure application behavior.
- Session **init** and **awake** methods are places to initialize session variables and configure session behavior.
- Component **init** and **awake** methods are places to initialize component variables and configure component behavior.

- The **sleep** and **dealloc** methods are places to deallocate variables initialized in **awake** and **init**, respectively.
- In most situations, you can use **init** to initialize variables. To optimize applications, you might do more of your initializations in **awake**, especially if they involve inexpensive operations.
- Use **takeValuesForRequest:inContext:** methods to access request and context information and to perform postprocessing of user input.
- Use **invokeActionForRequest:inContext:** methods to substitute a different page for the response (except for initial requests).
- Use **appendToResponse:inContext:** methods to add to the response content or otherwise manipulate the HTTP response.