Using WebScript

This chapter provides an overview of WebScript, the WebObjects scripting language. The chapter includes the following major sections:

- "The WebScript Language" describes basic WebScript language syntax and the "modern" variation on this syntax.
- "Using WebScript in a WebObjects Application" describes using WebScript within the context of a WebObjects application. This section uses a simple example application to explain the issues that arise in creating a WebObjects application, as well as special WebObjects features.
- "WebScript Language Summary" provides a reference to the WebScript language.

For a detailed discussion of the structure of a WebObjects application, see the chapter "Getting Started."

The WebScript Language

This section describes WebScript language features and syntax. For a complete WebScript example and a discussion of how scripts operate within the larger context of a WebObjects application, see the section "Using WebScript in a WebObjects Application."

Declaring Variables

To declare a variable in WebScript, use the syntax:

```
id myVar;
id myVar1, myVar2;
```

A value can also be assigned to a variable at the time it is declared:

```
id myVar3 = 77;
```

WebScript only supports one data type: objects (**id**s).

The id Data Type

The **id** type is defined as a pointer to an object—in reality, a pointer to the object's data (its instance variables). Like a C function or an array, an object is identified by its address. All objects, regardless of their instance variables or methods, are of type **id**.

Making Assignments

The basic syntax for making assignments in WebScript is straightforward:

```
myVar = aValue;
```

The value you assign to a variable can be either a constant or another variable. For example:

```
// assign another variable to a variable
myVar = anotherVar;

// assign a string constant to a variable
myString = @"This is my string.";
```

The syntax `myString = @"This is my string";` is a way of creating instances of the class NSString. For more discussion of this syntax, see the section "Creating Constant NSStrings, NSArrays, and NSDictionaries."

WebScript only supports one data type: objects (**id**s). However, if you assign a literal integer or floating point value to a variable:

```
id myInt = 167;
```

WebScript represents it as an NSNumber object. In this sense WebScript can be said to support integers and floats.

Messaging in WebScript

To get an object to do something in WebScript, you send it a message telling it to perform a method. In WebScript, message expressions are enclosed in square brackets:

[*receiver message*]

The receiver is an object, and the message tells it what to do. For example, the statement:

```
[aString length];
```

tells the object **aString** to perform its **length** method, which returns the string's length. Methods can also take arguments. For example, this statement:

```
[aString isEqual:anotherString];
```

tells the object **aString** to perform its **isEqual**: method, which takes another object as an argument and tests it against **aString** for equality. A method can take multiple arguments. For example the statement:

```
[aString insertString:anotherString atIndex:3];
```

inserts the characters of **anotherString** into **aString** at the specified index. Note that the method name **insertString:atIndex:** has two colons, one for each of its arguments. The colons are preceded by keywords that describe their arguments (for example, **atIndex:** takes as its argument an integer representing an index).

One message can also be nested inside another. Here the **description** method returns the string representation of an NSCalendarDate object **myDate**, which is then appended to **aString**.The resulting string is assigned to **newString**:

```
newString = [aString stringByAppendingString:[myDate description]];
```

To give another example, here the array **anArray** returns an object at a specified index. That object is then sent the **description** message, which tells the object to return a string representation of itself, which is assigned to **desc**:

```
id desc = [[anArray objectAtIndex:anIndex] description];
```

Sending a Message to a Class

Most commonly, the object receiving a message is an *instance* of a class. For example, in the statement:

```
[aString length];
```

the variable **aString** is an instance of the class NSString.

However, sometimes you send messages to a class. You send a class a message when you want to create a new instance of that class. For example the statement:

```
aString = [NSString stringWithString:@"Fred"];
```

tells the class NSString to invoke its **stringWithString:** method, which returns an instance of NSString that contains the specified string. Note that a class is represented in a script by its corresponding class name—in this example, NSString.

The classes you use in WebScript include both class and instance methods. Most class methods create a new instance of that class, while instance methods provide behavior for instances of the class. The following example shows how you use an NSString class method to create an instance of NSString, and then use instance methods to operate on the instance **myString**:

```
// Use a class method to create an instance of NSString
id myString = [NSString stringWithFormat:@”The next word is %@”, word];

// Use instance methods to operate on the instance myString
length = [myString length];
lcString = [myString lowercaseString];
```

In a class definition, class methods are preceded by a plus sign (+), while instance methods are preceded by a minus sign (-). You can define new classes in WebScript: see "Scripted Classes" for details. Or you can take advantage of existing classes. For more information, see the chapter "A Foundation for WebScript Programmers: Quick Guide to Useful Classes."

Creating Objects

There are two different ways to create objects in WebScript. The first approach, which applies to all classes, is to use class creation methods. The second approach applies to just NSStrings, NSArrays, and NSDictionaries. For these classes WebScript provides a convenient syntax for initializing constant objects.

Using Creation Methods

All classes provide creation methods that you can use to create an instance of that class. Depending on the class and the particular creation method, the instances of the class you create might be either mutable (modifiable) or immutable (constant). When you use creation methods to create NSStrings, NSArrays, and NSDictionaries, you can choose to create either an immutable or a mutable object. For clarity, it's best to use immutable objects wherever possible. Only use a mutable object if you need to change its value after you initialize it.

Here are some examples of using creation methods to create mutable and immutable NSString, NSArray, and NSDictionary objects:

```
// Create a mutable string
string = [NSMutableString stringWithFormat:@"The string is %@", aString];

// Create an immutable string
string = [NSString stringWithFormat:@"The string is %@", aString];

// Create a mutable array
array = [NSMutableArray array];
anotherArray = [NSMutableArray arrayWithObjects:@"Marsha", @"Greg", @"Cindy", nil];

// Create an immutable array
array = [NSArray arrayWithObjects:@"Bobby", @"Jan", @"Peter", nil];

// Create a mutable dictionary
dictionary = [NSMutableDictionary dictionary];

// Create an immutable dictionary
id stooges = [NSDictionary
        dictionaryWithObjects:@("Mo", "Larry", "Curley")
        forKeys:@("Stooge1", "Stooge2", "Stooge3")];
```

The instances of some classes are always mutable or always immutable. The following examples show how you can create and work with NSCalendarDates, which are always immutable:

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

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

// Using the creation method dateWithString:, create an NSCalendarDate
// instance 'newDate' from 'dateString'
newDate = [NSCalendarDate dateWithString:dateString
  calendarFormat:@"%B %d, %Y"];

// Return a new date in which newDate's day field is decremented
date = [newDate addYear:0 month:0 day:-1 hour:0 minute:0 second:0];
```

For a detailed discussion of these classes and a more complete listing of methods, see the chapter "A Foundation for WebScript Programmers: Quick Guide to Useful Classes."

Creating Constant NSStrings, NSArrays, and NSDictionaries

NSStrings, NSArrays, and NSDictionaries are the classes you use most often in WebScript. WebScript provides a convenient syntax for initializing constant objects of these types. In such an assignment statement, the value you're assigning to the constant object is preceded by an at sign (@). You use parentheses to enclose the elements of an NSArray, and curly braces to enclose the key-value pairs of an NSDictionary. The following are examples of how you use this syntax to assign values to constant NSStrings, NSArrays, and NSDictionaries in WebScript:

```
myString = @"hello world";
myArray = @("hello", "goodbye");
myDictionary = @{"key" = 16};
anotherArray = @(1, 2, 3, "hello");
aDict = @{ "a" = 1; "b" = "hello world"; "c" = (1,2,3);
    "d" = { "x" = 1; "r" = 2 }};
```

The following rules apply when you use this syntax to create constant objects:

- The value you assign must be a constant (that is, it can't include variables). For example, the following is not allowed:

  ```
  // This is not allowed!!
  myArray = @("hello", aVariable);
  ```

- You shouldn't use @ to identify NSStrings, NSArrays, or NSDictionaries inside the value being assigned. For example:

```
// This is not allowed!!
myDictionary = @(@"value" = 3);

// Do this instead
myDictionary = @("value" = 3);
```

For more information on NSStrings, NSDictionaries, and NSArrays, see the chapter "A Foundation for WebScript Programmers: Quick Guide to Useful Classes."

Writing Your Own Methods

You can write your own methods in WebScript. The methods you write can be associated with one of two types of objects: the WOApplication object that's automatically created when you run your script, or a WOComponent object that's associated with a particular grouping of a script, an HTML template, and a declarations file (for more information, see the section "The Role of Scripts in a WebObjects Application"). When you write your own methods, you're effectively extending the behavior of the object associated with the script.

You implement WOApplication methods in the application script. You implement WOComponent methods in a *component script*—that is, a script that has a corresponding HTML template and declarations file. This grouping of three files most commonly maps to a single, dynamically generated HTML page, but this isn't always the case—a component can also represent just a portion of a page.

To define a new method, simply put its implementation in the appropriate application or component script file. You don't need to declare it ahead of time. For example, the following method **addFirstValue:toSecondValue:** adds one value to another and returns the result:

```
- addFirstValue:firstValue toSecondValue:secondValue {
  id result;
  result = firstValue + secondValue;
  return result;
}
```

In this example, note the following:

- There is no type information supplied for the method's arguments and return types. These types are assumed to be (and must be) **id**, and if you supply any type information, you will get an error.

```
// This is fine.
- aMethod:anArg {

// NO!! This won't work.
- (void) aMethod:(NSString *)anArg {

// This won't work either.
- (id)aMethod:(id)anArg {
```

- This method returns a value, stored in **result**. If a method doesn't return a meaningful value, you don't have to include a return statement (and, as stated above, even if a method returns no value you shouldn't declare it as returning **void**).

To invoke the **addFirstValue:toSecondValue:** method shown above from another method in the same script, you'd simply do something like the following:

```
id sum, val1 = 2, val2 = 3;
sum = [self addFirstValue:val1 toSecondValue:val2];
```

To access the method from another script, you'd first return the page associated with the script in which the method is implemented. You'd then ask the page object to perform the method:

```
id sum, val1 = 2, val2 = 3;
// Get the page in which the method is implemented
id computePage = [WOApp pageWithName:@"Compute"];
// Send the page object to perform the method
sum = [computePage addFirstValue:val1 toSecondValue:val2];
```

The **pageWithName:** method is discussed in more detail in the section "Accessing and Sharing Variables."

What is self?

In WebScript, **self** is available in every method. It refers to the object (the WOApplication object, the WOSession object, or the WOComponent object) associated with a script. When you send a message to **self**, you're telling the object associated with the script to perform a method that's implemented in the script. For example, suppose you have a script that implements the method **giveMeARaise**. From another method in the same script you could invoke **giveMeARaise** as follows:

```
[self giveMeARaise];
```

This tells the WOApplication, WOSession, or WOComponent object associated with the script to perform its **giveMeARaise** method.

Categories

Categories are a feature of WebScript borrowed from Objective-C. They allow you to add methods to an existing class without having to create a subclass of that class. The existing class can be a WebObjects public class or any custom or NeXT-provided Objective-C class. The methods added by the category become part of the class type. You can invoke them on any object of that type within an application.

To create a category you must implement it within an **@implementation** block, which is terminated by the **@end** directive. The category name appears in parentheses after the class name. Unlike Objective-C categories, no typing of method arguments or return values is allowed. The category can be in any script file of the application.

The following example is a simple category of WORequest that gets the sender's Internet e-mail address from the request headers ("From" key) and returns it (or "None").

```
@implementation WORequest(RequestUtilities)
- emailAddressOfSender {
    id address = [self headerForKey:@"From"];
    if (!address) address = @"None";
    return address;
}
@end
```

Elsewhere in your WebScript code, you invoke this method on WORequest objects just as you do with any other method of that class:

```
- takeValuesFromRequest:request inContext:context {
    [super takeValuesFromRequest:request inContext:context];
    [self logWithFormat:@"Email address of sender: %@",
        [request emailAddressOfSender]];
}
```

Scripted Classes

You can create an Objective-C class in a script file, then load that class into your application at run time and generate instances from it. The instances will behave as any other Objective-C object.

As with categories, no typing is permitted. You must specify the class interface in an **@interface...@end** block and the class implementation in an **@implementation...@end** block. For the sake of loading, the scripted class code should be in its own ".wos" file. The following example is in a file named **Surfshop.wos:**

```
@interface Surfshop:NSObject {
id name;
id employees;
}
@end

@implementation Surfshop
- initWithName:aName employees:theEmployees {
    name = [aName copy];
    employees = [theEmployees retain];
    return self;
}
@end
```

To use the class, you locate it in the application, load it, and then allocate and initialize instances using the class object. For example:

```
id allSurfshops;
- init
{
    id scriptPath;
    id surfshopClass;

    [super init];
    scriptPath = [WOApp pathForResourceNamed:@"Surfshop" ofType:@"wos"];
    surfshopClass = [WOApp scriptedClassWithPath:scriptPath];
    allSurfshops = [NSMutableArray array];
    [allSurfshops addObject:[[[surfshopClass alloc] initWithName:
      "Banana Surfshop" employees:@("John Popp", "Jenna de Rosnay")] autorelease]];
    [allSurfshops addObject:[[[surfshopClass alloc] initWithName:
        "Rad Swell" employees:@("Robby Naish", "Nathalie Simon")] autorelease]];

    return self;
}
```

"Modern" WebScript Syntax

When you designate an action method in WebObjects Builder, it can emit code similar to this (if the appropriate preference is selected):

```
function submit() {
}
```

This is an instance of “modern” syntax, a variation of WebScript designed to appeal to programmers more familiar with such languages as Visual Basic and Java. The rules for transformming “classic” WebScript to “modern” WebScript can be illustrated by examples that map one to another.

Method Definition

Classic:

```
- submit {
    // <body>
}
```

Modern:

```
function submit() {
    // <body>
}
```

Method Invocation — No Argument

Classic:

```
[self doIt];
```

Modern:

```
self.doIt();
```

Method Invocation — One Argument

Classic:

```
[guests addObject:newGuest];
```

Modern:

```
guests.addObject(newGuest);
```

Method Invocation — Two or More Arguments

Classic:

```
[guests insertObject:newGuest atIndex:anIndex];
```

Modern:

```
guests.insert(object := newGuest, atIndex := anIndex);
```

Note that in this last example that the left parenthesis can occur at any point before the first argument. You can even have no keyword on the left side of the first assignment. Thus the following two mappings would be valid as well:

```
guests.insertObjec(t := newGuest, atIndex := anIndex); // not recommended!
guests.insertObject(newGuest, atIndex := anIndex);
```

However, if the "modern" message is to map to an existing Objective-C method (which, of course, follows "classic" WebScript syntax), then the characters just inside the left parenthesis (that is, on the left side of the first assignment) are significant. When WebScript transforms "modern" to "classic" syntax internally, it capitalizes this character before concatentating the keywords of the selector. Thus the first example immediately above would change to "classic" syntax as:

```
[guests insertObjecT:newGuest atIndex:anIndex];
```

By default WebObjects Builder emits "classic" WebScript syntax. If you want to work with "modern" syntax, choose the appropriate option in the Language display of Preferences.

Using WebScript in a WebObjects Application

This section discusses using WebScript in the context of a WebObjects application. For a detailed discussion of the structure of a WebObjects application, see the chapter "Getting Started."

The Role of Scripts in a WebObjects Application

In developing WebObjects applications, you usually write your business logic as compiled Objective-C code (though you can write entire applications using just WebScript). You then use WebScript to provide your "interface logic." A WebScript script typically includes the following ingredients:

- Variable declarations
- The instantiation of objects that get bound to HTML elements
- Action methods that define a response to user actions
- Logic for performing page navigation

Component Scripts

Most scripts are for *components*. A component is a page or a identifiable part of a page that can dynamically generate itself and send action messages when users interact with it. A component contains one or more dynamic HTML elements and usually some static HTML elements as well. Components can exist on the

server or on the client browser. (See “Java Client-Side Components” for an example of the latter.)

On the server side, components are instances of a WOComponent subclass. (WOComponent is an abstract class that defines the interface and behavior of WOComponent objects.) At run time, WebObjects creates an instance of a special subclass for each component script and dynamically makes the script code the class implementation. Applications can, and often do, have multiple components.

In most cases, a script for a server-side component has a corresponding declarations file and HTML template. The declarations file provides a mapping between the actions and variables defined in the script, and the HTML elements that will be dynamically generated and then substituted in the HTML template.The three files in a group have the same base name but different extensions; for example, **Main.wos** (script), **Main.wod** (declarations), and **Main.html** (template). These application resources are used by the WOComponent object to prepare responses to user requests.

In a WebObjects application you generally put each group of three files (the script, the declaration, and the HTML template) into a directory that has the same base name and the extension **.wo**. So, for example, you can have a directory **Main.wo** that contains the files **Main.wos**, **Main.wod**, and **Main.html**. The script associated with a component (in this example, **Main.wos**) is called a *component script*.

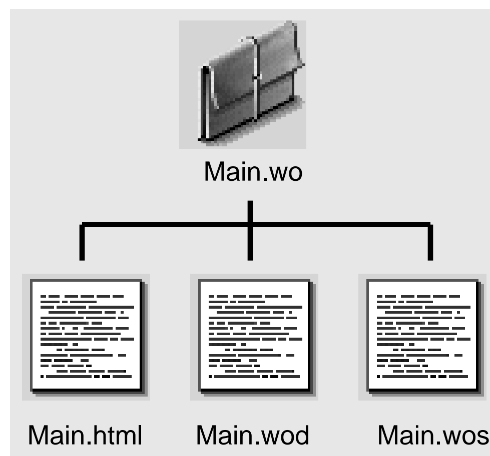

Figure 1. The Contents of a Component Directory

The Application and Session Scripts

In addition to having one or more components, a WebScript application can also include an *application script* and a *session script*. The application script is where you declare and initialize application variables, and where you perform tasks that affect the entire application. The session script is where you declare, initialize, and store variables that persist throughout a session; in a session script you also perform tasks that affect the session as a whole. For more information on application, session, and other variables, see the section “Variables and Scope.”

The application script has the name **Application.wos** and the session script is named **Session.wos**. Both files reside immediately under the application (**.woa**) directory. Similar to component scripts, the script code is dynamically made the implementation code of special WOApplication (**Application.wos**) and WOSession (**Session.wos**) subclasses from which instances are generated at run time.

Visitors Example

To explain how a WebScript operates within the larger context of a WebObjects application, this section uses the Visitors application as an example. The Visitors application takes the name of the current visitor, and displays the most recent visitor, the total number of visitors to the page, and the time remaining in the session:

Figure 2. The Visitors Example

The Visitors application includes the following directories and files:

```
/Visitors.woa
  Application.wos
  Session.wos
  /Main.wo
       Main.html
       Main.wod
       Main.wos
```

To view the contents of **Main.wod** and **Main.html**, see the on-line Visitors example. The contents of **Application.wos**, **Session.wos**, and **Main.wos** are listed in the following sections.

Application.wos

Application.wos is the application script for the Visitors application. It declares two application variables: **visitorNum** and **lastVisitor**. Application variables can be accessed throughout the application, and they live for the duration of the application. For more information on application variables, see the section "Variables and Scope."

```
id lastVisitor;
  // the most recent visitor
id visitorNum;
  // the total number of visitors the page

- init
{
    [super init];
    lastVisitor = @"";
    [self setTimeOut:7200];
    return self;
}
```

Implementing the init Method

The **Application.wos** script includes a method called **init**. The **init** method is where you can initialize the variables associated with the object. Thus, in an application script, it's common to implement an **init** method to initialize application variables. In a component script, on the other hand, you use **init** to prepare the associated page and its variables for use during the processing of the page.

As illustrated in the above example, an implementation of **init** should always begin by invoking the **init** method of **super** (the superclass object). It should always end by returning **self**.

Session.wos

In **Session.wos** of the Visitors application, an **init** method also initializes declared variables. These variables have session-wide visibility and persistence. But this **init** does much more than initialze variables.

- It sets a time-out period for the session.
- It creates a timer scheduled to fire every second.
- It implements a method that is invoked when the timer is fired. This method increments a "seconds-counter" variable, which is bound to a WOString on the page.

```
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;
}

- timeOfSession
{
    timeSinceSessionBegan++;
}
```

As this example shows, you can do many things in the **init** method to set up the associated object besides initializing variables. This code example also illustrates a couple specific aspects of WebScript. The "hidden" variable **self** in this script refers to a WOSession object, and so the method invoked (**setTimeOut:**) must be declared by the WOSession class. Second, you can invoke any method of the Foundation framework, such as NSTimer's **scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:**.

Note: The example above illustrates a syntax difference between WebScript and Objective-C: the way you refer to selectors and similar entities. In Objective, you use the **@selector()** directive; in WebScript, because "@" has special significance, you simply quote the selector.

Main.wos

The script associated with the first (and in this example, only) page of the Visitors application is **Main.wos**. This script increments the number of visitors to the page (**visitorNum**), and assigns the name (**aName**) entered in the application's text field to the last visitor (**lastVisitor**). It then clears the text field by assigning an empty string to **aName**.

```
id number, aName;

- awake {
    if (!number) {
        number = [[self application] visitorNum];
        number++;
        [[self application] setVisitorNum:number];
   }
}

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

Implementing the awake Method

For a given page, the **awake** method is invoked exactly once per transaction, at the beginning of that transaction. The **init** method is invoked only once, at the start of an object's lifetime (see "The Duration of a Component" for the reasons why). Because of this, it is more appropriate in the Visitor application to implement **awake** rather than **init**. We want to track each "visit" to this page. Because **awake** is invoked once per transaction, if the same page handles the request as well as generates the response (for example, the first page of an application), the **awake** method is only invoked during the request phase.

The **awake** method is a good place to initialize variables whose values are known or can be resolved at the start of the request-response cycle, such as a list of hyperlinks. The advantage of using **awake** to perform this type of initialization is that the variables are guaranteed to be initialized every time the page is displayed.

The **awake** method has a complementary method, **sleep**, in which you can explictly deallocate objects assigned to variables by assigning **nil** to the variables. As a technique for improving application scalability, you can turn off page caching, initialize variables in **awake** (rather than in **init**), and deallocate them in **sleep** .

The Duration of a Component

When users navigate to a page of an WebObjects application for the first time, a WOComponent object associated with the page is created and (in **init**) initialized. When users go to another page, the original component typically does not go away. It persists through an arbitrary number of subsequent transactions before it's deallocated. Thus when users backtrack to a page they visited earlier (as long as the transaction limit hasn't expired) things are as they left them—displayed results, entered values, and selected options.

Note: An application looks first for request pages in its cache and if they're there, it restores them. Thus pages are restored when users backtrack to a page and when a request component returns itself as the response page. In typical request-handling scenarios, the response page is created. For example, the application's **pageWithName:** method (typically invoked in an action method) returns a new instance of a page—even if that page has been visited before in the same session.

You set the number of pages the application caches, and thus the life span of an application's components, with WOApplication's **setPageCacheSize:** method. If the page-cache size attribute is not explicitly set, the default is 30 pages. You can also determine the current transaction limit by sending **pageCacheSize** to the application object.

To minimize the size of a session, you can reduce the page-cache size or you can turn off page caching altogether by setting the page-cache size to zero. If you turn off page caching, you must re-initialize each page for each transaction it's involved in. You can perform initializations in **init** or **awake**, since both are invoked with identical frequency.

Relying on the page cache to retain the state of a page for a user introduces certain issues and problems. One implication, of course, is scalability. If you want finer control over this aspect of an application, consider selectively storing page state in session variables, reinitializing variables in the **awake** method, or some combination of the two.

Variables and Scope

In WebScript, the scope of variables depends on where and how you declare them. The notion of scope in WebScript really encompasses two different ideas: a variable's visibility and its lifetime.

The simplest kind of variable in WebScript is a local variable, which is declared inside a method as follows:

```
- aMethod {
        id localVar;
        /*...*/
}
```

Local variables have no visibility outside the method in which they're declared, and no lifetime beyond the method's execution. For this reason, they're the only type of variable that can't be referenced in a declarations file.

All other variables have some degree of persistence within your application. To understand the role of these variables, it's useful to think about the flow of activity in a WebObjects application. The life of a WebObjects application is marked by the continual recurrence of *requests* (such as a user clicking a control to initiate an action), and the subsequent *responses* (such as the server returning a dynamically generated HTML page in response to a request). A request-response cycle is called a *transaction*. Processing and variable scoping in a WebObjects application is organized around transactions.

Non-local variables behave differently depending on whether they're declared in an application script (where they're called application variables), in a session script (where they're called session variables), or in a component script (where they're called component variables).

Application variables

Application variables can be accessed from all pages of an application, and they last for the duration of an application. An application variable is available across all sessions, and there is one copy of the variable per application. Application variables are declared in the application script outside a method as follows:

```
id applicationVar;
```

Session Variables

Whereas all users of an application see an application variable with the same value, each session has its own unique set of session variables. A variable with session scope lasts for the duration of a session. A session represents a browser (user) accessing a WebObjects application, which could be serving multiple users. A session is initiated when a browser (single user) connects to a WebObjects application, at which time the session is assigned a unique identifier. This session ID is embedded in the URLs of the pages associated with the application. The session ID lasts as long as the session is valid. A session is terminated either when the user quits out of his or her browser, or when the application explicitly times the session out. For more information on session time out, see the section "Setting Session TimeOut" in the chapter "Managing State."

A session variable is accessible from every component script and from the application script. Its value is stored and restored at the beginning and the end of each request-response cycle. There is one copy of the variable per user session. Session variables are declared in the session script (**Session.wos**) outside a method. You can access those variables by sending a message to the current WOSession object, obtainable through a message to **self**:

```
id value = [[self session] mySessionVariable];
[[self session] setMySessionVariable:newValue];
```

Component Variables

A component variable is declared in a component script outside a method, as follows:

```
id myVar;
```

This kind of variable lasts the lifetime of a component. The WOApplication object usually stores each component instance through multiple transactions, the number of which is determined by the application's page-cache size. (See "The Duration of a Component" for more information.) Component variables are visible to all of the methods within the script in which they're declared.

Variables and Scope: a Summary

The following table summarizes the different types of variables in WebScript:

Variable Type	Where It's Declared	Where It's Visible	How Long It Lives
Local	Inside a method in either an application or a component script	Only inside the method in which it's declared	For the duration of the method
Component	Outside a method in a component script	Inside the script in which it's declared	For the duration of a component, which is determined by the application's page-cache size
Session	Outside a method in an session script	Component scripts can access session variables by sending accessor messages to the WOSession object. Every session has its own version of a session variable.	For the duration of the session

Application	Outside a method in an application script	In the application script. Component scripts can access application variables by sending accessor messages to the WOApplication object. Every session sees application variables with the same value.	For the duration of the application

Accessing and Sharing Variables

WebScript automates the process of accessing non-local variables, whether they're declared in an application script, a session script, or in a component script. For a non-local variable **myVar**, for example, you can set and return its value from the script that declares it, as follows:

```
[self myVar];
[self setMyVar:newValue];
```

You don't have to implement these methods to invoke them—WebScript does this work behind the scenes. For example, you may notice that the Visitors **Application.wos** script doesn't implement **visitorNum**, **setVisitorNum:**, or **setLastVisitor:** methods, yet the **Main.wos** script invokes them.

In these statements:

```
[self myVar];
[self setMyVar:newValue];
```

the **myVar** and **setMyVar:** messages are sent to **self**, which indicates that the variable **myVar** is declared in the script that's accessing it. Sometimes a component script has to access application or session variables declared elsewhere. When you work with application and session variables, remember that they're owned by the application and session objects, respectively. To set or return their values, you send a message to the appropriate object, which, from a component script, you can always get by sending **application** or **session** to **self**. For example, the **Main.wos** script in the Visitors example includes these statements:

```
number = [[self application] visitorNum];
[[self application] setVisitorNum:number];
[[self application] setLastVisitor:[[self application] aName]];
```

Note: The application object is also represented by the global variable WOApp. However, use of WOApp is discouraged because global variables are not permitted in some of the languages supported by WebObjects.

You can also access a non-local variable declared in one component script from another component script. This is something you commonly do right before you navigate to a new page, for example:

```
id anotherPage = [[self application] pageWithName:@"Hello"];
[anotherPage setNameString:newValue];
```

The current script uses the statement `[anotherPage setNameString:newValue];` to set the value of **nameString**, which is declared in the page entitled "Hello".

This example uses the **pageWithName:** method, which takes the name of a page as an argument and returns that page. You most commonly use **pageWithName:** inside a method that returns a new page for display in the browser. Such a method could be associated with a hyperlink or a submit button. For example:

```
- contactPsychicNetwork
{
    id nextPage;
    nextPage = [[self application] pageWithName:@"Predictions"];
    return nextPage;
}
```

WebScript Language Summary

This section summarizes the WebScript language.

Reserved Words

WebScript includes the following reserved words:

```
if
else
for
while
id
break
continue
nil
YES/NO
```

Statements

WebScript supports the following statements:

```
if
else
for
while
```

```
break
continue
return
```

In WebScript these statements behave as they do in the C language.

Arithmetic Operators

WebScript supports the arithmetic operators +, - , /, *, and %. The rules of precedence in WebScript are the same as those for the C language. You can use these operators in compound statements such as:

```
b = (1.0 + 3.23546) + (((1.0 * 2.3445) + 0.45 + 0.65) - 3.2);
```

Logical Operators

WebScript supports the negation (!), AND (&&), and OR (||) logical operators. You can use these operators as you would in the C language, for example:

```
if ( !(!a || a && !i) || (a && b) && (c || !a && (b+3)) ) i = 0;
```

Relational Operators

WebScript supports the relational operators <, <=, >, >=, ==, and !=. In WebScript these operators behave as they do in C.

Increment and Decrement Operators

WebScript supports the ++ and -- operators. These operators behave as they do in the C language, for example:

```
// Use myVar as the value of the expression and then increment myVar
myVar++;

// Increment myVar and then use its value as the value of the expression
++myVar;
```

id

WebScript supports only one data type: objects (**id**s). The **id** type is defined as a pointer to an object—in reality, a pointer to the object's data (its instance variables). Like a C function or an array, an object is identified by its address. All objects, regardless of their instance variables or methods, are of type **id**.

self

In WebScript, **self** is available in every method. It is a "hidden" variable that refers to the object (the WOApplication object, the WOSession object, or the WOComponent object) associated with a script. When you send a message to

self, you're telling the object associated with the script to perform a method that's implemented in the script.

super

As with **self**, **super** is a "hidden" variable available in every method. By sending a message to **super**, you are invoking the superclass' implementation of the method. An invocation of **super**'s **init** method should occur at the beginning of an **init** implementation.

What Are the Origins of WebScript?

WebScript is an interpreted language that uses a subset of Objective-C syntax. Objective-C is an object-oriented language that adds extensions to the C language.

You do not need to know Objective-C to use WebScript or to write WebObjects applications. However, if you're interested in learning more about the Objective-C language, see NeXT's Software's *Object-Oriented Programming and the Objective-C Language*.

A Note to Objective-C Developers

WebScript uses a subset of Objective-C syntax, but its role within an application is significantly different. The following table summarizes some of the differences.

Objective-C	WebScript
Is compiled	Is interpreted
Supports primitive C data types	Only supports the **id** data type
Requires method prototyping	Doesn't require method prototyping (that is, you don't declare methods before you use them)
Usually includes a .h and a .m file	Usually has corresponding declarations and HTML template files (unless it is an application script)
Supports all C language features	Has limited support for C language features; for example, doesn't support structures, pointers, enumerators, or unions
Methods not declared to return void must include a return statement	Methods aren't required to include a return statement
Has preprocessor support	Has no preprocessor support—that is, doesn't support the #import or #include statements

Perhaps the most significant difference between Objective-C and WebScript is that in WebScript, the only valid data type is **id**. Some of the less obvious implications of this are:

- You can't use methods that take non-object arguments (unless those arguments are integers or floats, which WebScript converts to NSNumbers). For example, in WebScript the following statement is invalid:

```
// NO!! This won't work.
string = [NSString stringWithCString:"my string"];
```

- You can only use the "at sign" character (@) as a conversion character with methods that take a format string as an argument:

```
// This is fine.
[self logWithFormat:@"The value is %@", myVar];

// NO!! This won't work.
[self logWithFormat:@"The values are %d and %s", var1, var2];
```

- You shouldn't supply any type information for a method's arguments and return types. These types are assumed to be **id**, and if you supply any type information, you will get an error.

```
// This is fine.
```

```
- aMethod:anArg {

// NO!! This won't work.
- (void) aMethod:(NSString *)anArg {

// This won't work either
- (id)aMethod:(id)anArg {
```

- You need to substitute integer values for enumerated types.

  For example, suppose you want to compare two numeric values using the enumerated type NSComparisonResult. This is how you might do it in Objective-C:

```
result = [num1 compare:num2];
if(result == NSOrderedAscending)/* This won't work in WebScript */
      /* num1 is less than num2 */
```

  But this won't work in WebScript. Instead, you have to use the integer value of NSOrderedAscending, as follows:

```
result = [num1 compare:num2];
if(result == -1)
      /* num1 is less than num2 */
```

  For a listing of the integer values of enumerated types, see the "Types and Constants" section in the *Foundation Framework Reference*.