

---

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.
- “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 (**ids**).

### 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 `NSString`s, `NSArray`s, and `NSDictionary`s.”

WebScript only supports one data type: objects (**ids**). 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 `NSDate` 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:index] 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 don't define new classes in WebScript, but 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, NSArray, and NSDictionary. 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, NSArray, and NSDictionary, 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\")"];
```

Some classes only let you create either mutable or immutable objects. The following examples show how you can create and work with `NSDate`s, which are always immutable:

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

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

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

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

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

### Creating Constant `NSString`s, `NSArray`s, and `NSDictionary`s

`NSString`s, `NSArray`s, and `NSDictionary`s 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 `NSString`s, `NSArray`s, and `NSDictionary`s 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, NSArray, or NSDictionary inside the value being assigned. For example:

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

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

For more information on NSStrings, NSDictionary, and NSArray, 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 `WOWebScriptApplication` object that's automatically created when you run your script, or a `WOWebScriptComponentController` 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 `WOWebScriptApplication` methods in the application script. You implement `WOWebScriptComponentController` 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 = [WOWApp 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 (either the **WOWWebScriptApplication** object or the **WOWWebScriptComponentController** 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 **WOWWebScriptApplication** or **WOWWebScriptComponentController** object associated with the script to perform its **giveMeARaise** method.



## 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

In most cases, a script 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 a corresponding `WOWebScriptComponentController` to prepare responses to user requests. Most commonly a `WOWebScriptComponentController` represents a single page in a WebObjects application, though it can also represent just a portion of a page.

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**. Each group of three files is called a *component*. The script associated with a component (in this example, **Main.wos**) is called a *component script*.

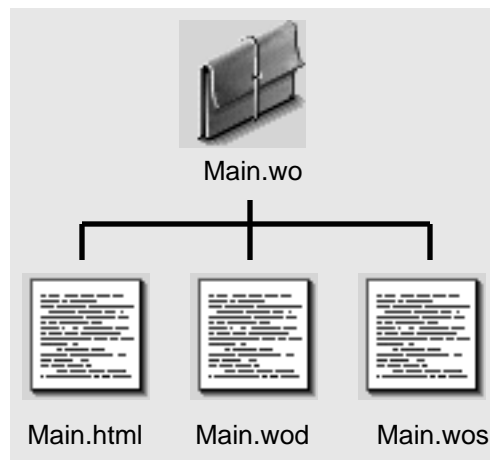


Figure 6. The Contents of a Component Directory

### The Application Script

Often, in addition to having multiple components (that is, subdirectories that contain a script, a declarations file, and an HTML template), a WebScript application also has an *application script*. This script is where you declare and initialize global and session variables. For more information on global and session variables, see the section “Variables and Scope.”

The application script has the name **Application.wos**, and it resides immediately under the application directory.

### 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 as well as the total number of visitors to the page:



**Visitors**

---

To record your visit, please enter your name below.

**submit**

---

Number of visitors to this page: **8**

Most recent visitor: **Thibault**

**Figure 1.** The Visitors Example

The Visitors application includes the following directories and files:

```
/Visitors
  Application.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** and **Main.wos** are listed in the following sections.

### **Application.wos**

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

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

```

- awake {
    // Obsolete sessions that have been inactive for more than 2 minutes
    [WObj setSessionTimeout:120];
    visitorNum = 0;
}

```

### Using the awake Method

The **Application.wos** script includes a method called **awake**. In an application or component script, it's common to implement an **awake** method to prepare the associated page and its variables for use during the processing of the page.

For a given page, the **awake** method is invoked exactly once for each 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 the best place to initialize variables whose values remain static for the life of the page, 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.

### 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 = [WObj visitorNum];
        number++;
        [WObj setVisitorNum:number];
    }
    return self;
}
- recordMe
{
    if ([aName length]) {
        [WObj setLastVisitor:aName];
        [self setAName:@""]; // clear the text field
    }
}

```

The concepts introduced by this example application are discussed in the following sections

## 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 global and session variables) or in a component script (where they're called transaction and persistent variables).

### Global variables

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

```
id globalVar;
```

### Session Variables

Whereas all users of an application see a global variable with the same value, each session has its own version 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. 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 application script outside a method as follows:

```
session id sessionVar;
```

Note that because the `WOWebScriptApplication` object owns global and session variables, in a component script you access those variables by sending a message to the application:

```
id value = [WObj mySessionVariable];  
[WObj setMyGlobalVariable:newValue];
```

### Transaction Variables

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

```
id myVar;
```

A variable with transaction scope lasts for the duration of a transaction, or HTML request. A transaction is defined as a request coming in and a response (usually an HTML page) going out. By the time the response is returned to the client, the variable’s value is no longer preserved. Transaction variables are visible to all of the methods within the script in which they’re declared.

### Persistent Variables

A persistent variable is declared in a component script outside a method using the **persistent** keyword, as follows:

```
persistent id myVar;
```

A persistent variable remains valid for a particular page for the duration of a session. It is stored right before a response is generated for a user request and restored when the client performs an action on the new page.

Whenever possible, you should refrain from using session and persistent variables since they tend to degrade performance. It’s preferable to

programmatically recreate variables in the script's **awake** method instead of declaring them as session or persistent, for example:

```
id myLinks; // just use a regular transaction variable...

// ... then re-initialize it in the script's awake method
- awake {
    myLinks = @"(My Autobiography", "Pictures of my Dog)";
    return self;
}
```

There are two cases in which you should use session and persistent variables:

- If you need to save a variable whose value can be modified (that is, if the variable's value can't be restored in **awake** by making a static assignment).
- If initializing a variable requires an expensive operation that shouldn't be performed unnecessarily, such as fetching rows from a database. However, if the variable represents a large amount of data, the cost of storing that data has to be weighed against the cost of recreating it

### **Remember: Pages Aren't Persistent!**

It's important to remember that pages aren't persistent in an application. They are created at the beginning of a transaction, and they disappear at the end. The life of a page actually spans two transactions:

1. First, the `WOWebScriptComponentController` associated with the page generates a response for a given request.
2. The `WOWebScriptComponentController` then handles the subsequent incoming request (such as a request triggered by a user clicking on a hyperlink).

Between these two occurrences, the `WOWebScriptComponentController` associated with a page is destroyed and reconstructed. Any variables in your application that aren't declared as persistent, session, or global are lost. Consequently, variables whose values the page depends on need to be either made persistent or recreated in **awake**. The preferred approach is to recreate them in **awake**, as described in the preceding section.

There is one significant exception to the general statement that pages aren't persistent. If an application has caching enabled (which can be achieved either by running the application from the command line with the `-c` option or by using `WOApplication`'s `setCachingEnabled:` method), transaction variables will behave exactly like global variables. In other words, users could potentially see transaction variables with the values assigned to them by other users. For this

reason, it's good practice to reinitialize transaction variables in your script's **awake** method.

## Variables and Scope: a Summary

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

Variable Type	Where It's Declared	How You Declare It	Where It's Visible	How Long It Lives
Local	Inside a method in either an application or a component script	<code>id myVar;</code>	Only inside the method in which it's declared	For the duration of the method
Transaction	Outside a method in a component script	<code>id myVar;</code>	Inside the script in which it's declared	For the duration of a transaction, which is defined as a request coming in and a response (usually an HTML page) going out
Persistent	Outside a method in a component script	<code>persistent id myVar;</code>	Inside the script in which it's declared	For the duration of a session
Session	Outside a method in an application script	<code>session id myVar;</code>	In the application script. Component scripts can access session variables by messaging the application. Every session has its own version of a session variable.	For the duration of the session
Global	Outside a method in an application script	<code>id myVar;</code>	In the application script. Component scripts can access global variables by messaging the application. Every session sees global 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 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 global or session variables declared in the application script. When you work with global and session variables, remember that they're owned by the application object **WOWebScriptApplication**. To set or return their values, you send a message to the **WOWebScriptApplication** object. For example, the **Main.wos** script in the Visitors example includes these statements:

```
number = [WOApp visitorNum];  
[WOApp setVisitorNum:number];  
[WOApp setLastVisitor:[WOApp aName]];
```

**WOApp** refers to the application object. The global variable **WOApp** is short for the following statement:

```
[WOApplication sharedInstance];
```

This statement returns the single **WOWebScriptApplication** object that's accessed by all users of an application.

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

```
id anotherPage = [WOApp 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 = [WOApp 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
persistent
session
action
```

### 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 (**ids**). 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 refers to the object (either the `WOWebScriptApplication` object or the `WOWebScriptComponentController` 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.

## persistent

The **persistent** keyword is used in a component script to identify a variable whose state is maintained for the duration of the current session (as opposed to transaction variables, which cease to exist at the end of a transaction).

## session

The **session** keyword is used in an application script to identify a variable whose state is maintained for the duration of an application, and of which every session has its own version (as opposed to a global variable, which has the same value in all sessions).

## action

The **action** keyword is used in a child component script to identify a `WOAction` object that the parent component associates with a method.

## 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 *NEXTSTEP 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 <b>id</b> 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 <b>#import</b> or <b>#include</b> 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 **NSNumber**s). 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*.