

---

## Compiling and Debugging a WebObjects Application

This chapter describes how to use compiled code in WebObjects application. It includes the following topics:

- When do you use compiled code?
- Creating and building a project
- Accessing compiled code from scripts
- Accessing scripts from compiled code
- Using C and C++ in WebObjects applications
- Debugging a WebObjects application

This chapter uses a small sample application, “Registration,” to illustrate how to integrate compiled code into a WebObjects application.

## When Do You Use Compiled Code?

There are two primary reasons you use compiled code in WebObjects: to boost performance and to provide your own custom classes (remember, WebScript doesn't allow you to create new classes).

Providing your own custom business classes is one common use of compiled Objective-C. Another is to subclass the WebObjects classes that are the building blocks of a WebObjects application:

- Component

Instead of using a component script to provide behavior in your application's components, you can instead provide a compiled component by subclassing `WOComponentController`. Just like component scripts, compiled components have associated declarations and HTML templates. For examples of applications that subclass `WOComponentController`, see `HelloWorldObjC` and `TimeOffObjC` in the WebObjects Examples directory.

- Application

You effectively extend the behavior of `WOWebScriptApplication` when you implement methods in an application script. However, there may be times you need to override `WOApplication` methods to change the fundamental behavior of the application object. For example, you might want to provide a custom state storage solution. For an example of subclassing `WOWebScriptApplication`, see the chapter “Managing State.”

- Dynamic element

You can provide your own dynamic elements by subclassing `WODynamicElement`.

Many applications use some combination of compiled code and scripts. For example, it's common to write your business logic as compiled Objective-C code and to then use `WebScript` to provide your interface logic. "Interface logic" refers to activities such as page navigation, capturing the data entered in forms, and managing the appearance of the user interface. Business logic, on the other hand, refers to the behavior associated with custom objects. For example, you could have an `OrderProcessing` object that validates orders to ensure that their data is correct and then checks them against available inventory.

## Creating Compiled Code

To create the compiled code that will eventually be integrated into your application, you need to follow these basic steps:

1. Use your development environment to create a project.
2. Implement a `main()` function.
3. Add to your project the libraries to which your application needs to link.
4. Create your classes and add them to your project.
5. Compile and link your code.

Once you've created and built your project, you can write your application's scripts, HTML templates, and declarations files. While you can choose to provide all of your application's behavior in compiled code, it's common to use some combination of compiled code and `WebScript`.

These steps are described in more detail in the following sections.

### Creating a Project

The first step in writing compiled code that can be integrated into a `WebObjects` application is to use your development environment to create a project.

If you're using ProjectBuilder on NEXTSTEP, you must set the project's type to be "Tool" in the New Project Panel.

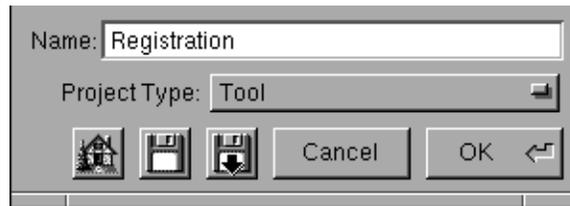


Figure 1. Creating a Project

Setting a project's type to "Tool" means that it won't include a nib file or an application object.

## Implementing a main() Function

When you use compiled code in a WebObjects application, you have to implement your own `main()` function. This function creates the autorelease pool, adaptor, and application objects used in your application.

To implement a `main()` function:

1. Using any text editor, open a new text file and give it a name that has the extension `.m`.

For the Registration project, for example, create a file called **Registration.m**.

2. Add the following text to the file:

```
#import <WebObjects/WOWebScriptApplication.h>
#import <WebObjects/WOApplicationAdaptor.h>
#import <foundation/NSAutoreleasePool.h>

void main(int argc, char *argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    WOApplicationAdaptor *adaptor =
        [[[WOApplicationAdaptor alloc] init] autorelease];
    WOWebScriptApplication *application = [WOWebScriptApplication
        sharedInstance];
    [adaptor runWithApplication:application];
    [pool release];
    exit(0);
}
```

The function begins by creating an autorelease pool that's used for the automatic deallocation of objects that receive the autorelease message. Next,

it creates an adaptor object that will be used to exchange data between the HTTP server and the WebObjects application object, which is created in the next statement. It then runs the adaptor and associates it with the newly created application. “Running” means that the adaptor will forward incoming requests from the server to the application and outgoing responses from the application to the server. The last statement releases the autorelease pool, which sends a release message to any object that has been added to the pool since the application began.

3. Add the file to your project.

If you’re using Project Builder, drag the file into the Other Sources suitcase in your project.

### Adding Libraries

The next step is to add to your project the libraries to which your application needs to link: **libWebObjects.a** and **libFoundation\_s.a**. You must add the libraries to your project in such a way that **libWebObjects.a** is linked first.

If you’re using Project Builder on NEXTSTEP, you include the libraries in your project by double-clicking the Libraries suitcase in Project Builder’s File Viewer, and then double-clicking the files **libFoundation\_s.a** and **libWebObjects.a** to add them.

Once you add the libraries to ProjectBuilder, you need to reorder them so that **libWebObjects.a** is linked first. To reorder the libraries, select the Libraries category and control-drag **libWebObjects.a** to the top of the list, as shown in Figure 2:

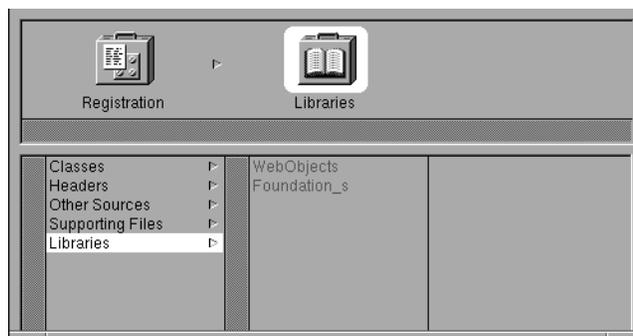


Figure 2. Reordering the Libraries

## Creating Your Classes and Adding Them to Your Project

Once you set up your project, you're ready to create the classes you'll compile and use in your WebObjects application. A project and its classes can go anywhere in your application directory. You may want to create a subdirectory within your application directory to contain your project and all of its related files.

## Building Your Code

Once you've created your `.h` and `.m` files and added them to your project, you're ready to build and link your code. There's no special magic required for this step—just use the process you normally would in your development environment.

If you're creating your application on NT, you need to make a copy of your application after you build it that doesn't have the extension `.exe`. This is required so that your application can be autostarted. You should also maintain a copy of your application that has the extension `.exe`, though, since this is the version you need to use to run the application from a command prompt.

# The Registration Application

This section uses the Registration application to describe how you integrate compiled Objective-C code into a WebObjects application.

The Registration application takes information about a user as input, validates it, and writes it out to a file. The first page of the application is shown in Figure 1.

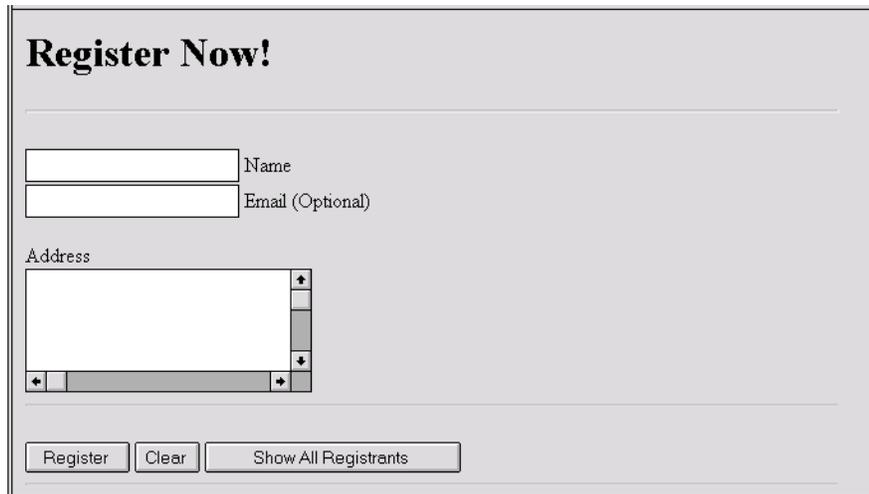


Figure 1. The Registration Application

The following table lists the files in the Registration application:

File	Description
Registration.m	Defines the <code>main()</code> function, which creates autorelease pool, adaptor, and application objects.
Person.[hm]	A custom Objective-C class whose primary function is to validate data entered by users.
RegistrationManager.[hm]	A custom Objective-C class whose primary functions are to register new users by writing their data to the <code>People.array</code> file and to return an array of all registrants.
People.array	A file that contains data about registrants in a property list format.
Application.wos	The application script. It creates and maintains a RegistrationManager object as a global variable.
Main.wos	The script for the application's first page. <code>Main.wos</code> has an associated declarations file ( <code>Main.wod</code> ) and HTML template ( <code>Main.html</code> ).
Registrants.wos	The script for the application's Registrants page, which lists all of the registered people. <code>Registrants.wos</code> has an associated declarations file ( <code>Registrants.wod</code> ) and HTML template ( <code>Registrants.html</code> ).

The scripted components `Main` and `Registrants` contain the application's interface logic. `Main.wos` includes methods for capturing user input and clearing the forms on the first page. `Registrants.wos` has just an `awake` method in which it retrieves data for display. The `Person` and `RegistrationManager` classes, on the other hand, contain the application's business logic. They validate user input and manage the application's data.

## Objective-C Classes in the Registration Application

The Registration application includes the `Person` and `RegistrationManager` classes.

### Person Class

When users enter data in the `Main` page of the Registration application, the data is stored in an `NSDictionary` that's used to initialize an instance of the `Person` class. The `Person` class includes a `validate` method that's used to check whether the data entered by the user includes values for a name and address. The `validate` method returns an `NSDictionary`. This dictionary contains a status message and a validation flag that indicates whether the registration should be allowed to proceed. If the user failed to enter a name or address, the validation flag value is "No," which disallows the registration. The status message then prompts the user to supply the missing information.

The `Person` class also includes the `name` and `personAsDictionary` methods. The `name` method is simply used to return the `Person`'s name, while the method `personAsDictionary` returns a dictionary representation of the `Person`. The dictionary representation is used when the `Person`'s data is written out to a file in a property list format (this is described in more detail in the section on the `RegistrationManager` class).

The header (`.h`) and implementation (`.m`) files for the `Person` class are listed below.

#### Person.h

```
// Person.h
#import <foundation/NSObject.h>

@class NSDictionary, NSString;

@interface Person: NSObject {
    NSDictionary *personRecord;
}
+ initWithDictionary:(NSDictionary *)personDict;
- initWithDictionary:(NSDictionary *)personDict;
- (NSDictionary *)validate;
- (NSString *)name;
```

```
- (NSDictionary *)personAsDictionary;

@end

Person.m
// Person.m
#import "Person.h"
#import <WebObjects/WebObjects.h>

@implementation Person

+ initWithDictionary:(NSDictionary *)personDict
{
    return [[[self class] alloc]
            initWithDictionary:personDict]
            autorelease];
}

- initWithDictionary:(NSDictionary *)personDict
{
    [super init];
    personRecord = [personDict copy];
    return self;
}

- (void)dealloc
{
    [personRecord release];
    [super dealloc];
}

- (NSDictionary *)validate
{
    NSMutableDictionary *isValid = [NSMutableDictionary dictionary];

    if(![[personRecord objectForKey:@"address"] length] &&
        ![personRecord objectForKey:@"name"] length){
        [isValid setObject:@"You must supply a name and address."
            forKey:@"failureReason"];
        [isValid setObject:@"No" forKey:@"isValid"];
    }
    else if (![personRecord objectForKey:@"name"] length){
        [isValid setObject:@"You must supply a name."
            forKey:@"failureReason"];
        [isValid setObject:@"No" forKey:@"isValid"];
    }
    else if (![personRecord objectForKey:@"address"] length){
        [isValid setObject:@"You must supply an address."
            forKey:@"failureReason"];
        [isValid setObject:@"No" forKey:@"isValid"];
    }
}
```

```

        else {
            [isValid setObject:@"Yes" forKey:@"isValid"];
        }
        return isValid;
    }

- (NSString *) name
{
    return [personRecord objectForKey:@"name"];
}

- (NSDictionary *) personAsDictionary
{
    return personRecord;
}

@end

```

### RegistrationManager Class

The RegistrationManager class has two primary functions: it registers a new Person (which entails writing the Person's data out to the **People.array** file), and it returns an array containing all of the registrants.

The header (.h) and implementation (.m) files for the RegistrationManager class are listed below.

#### RegistrationManager.h

```

// RegistrationManager.h
#import <foundation/NSObject.h>

@class Person, NSMutableArray, NSArray;

@interface RegistrationManager: NSObject {
    NSMutableArray *registrants;
}

+ manager;
- init;
- (NSDictionary *)registerPerson:(Person *)newPerson;
- (NSArray *)registrants;

@end

```

#### RegistrationManager.m

```

// RegistrationManager.h
#import <WebObjects/WebObjects.h>
#import "RegistrationManager.h"
#import "Person.h"

@implementation RegistrationManager

```

```
+ manager
{
    return [[[[self class] alloc] init] autorelease];
}

- init
{
    NSString *path = [WOApp pathForResource:@"People" ofType:@"array"];
    [super init];
    registrants = [[NSMutableArray arrayWithContentsOfFile:path] retain];
    if (!registrants)
        registrants = [[NSMutableArray alloc] init];

    return self;
}

- (void)dealloc {
    [registrants release];
    [super dealloc];
}

- (NSDictionary *)registerPerson:(Person *)newPerson
{
    int i;
    NSDictionary *results;
    NSString *currentName, *newPersonName = [newPerson name];
    NSString *path = [WOApp pathForResource:@"People" ofType:@"array"];

    results = [newPerson validate];
    if ([[results objectForKey:@"isValid"] isEqual:@"No"])
        return results;

    for (i = [registrants count]-1; i >= 0; i--) {
        currentName = [[registrants objectAtIndex:i] objectForKey:@"name"];
        if ([currentName isEqual:newPersonName]) {
            [registrants removeObjectAtIndex:i];
            break;
        }
    }
    [registrants addObject:[newPerson personAsDictionary]];
    [registrants writeToFile:path atomically:YES];
    return results;
}

// Return an array of all registrants
- (NSArray *)registrants
{
    return registrants;
}

@end
```

RegistrationManager's **init** and **registerPerson:** methods use the WOApplication method **pathForResource ofType:** to load the file **People.array** into the application. This method takes a path and the file's extension as arguments:

```
NSString *path = [WOApp pathForResource:@"People" ofType:@"array"];
```

You can use this method to load different kinds of resources into your application—for example, images, sound files, data files, and so on.

Another noteworthy feature of RegistrationManager is its use of an NSArray data source. The reason that the instance variable **registrants** object can be initialized from the file **People.array** is because the file contains data in an property list format. A property list is a compound data type that consists of NSStrings, NSArray, NSDictionary, and NSData. Property lists can be represented in an ASCII format, and property list objects such as NSDictionary and NSArray can consequently be initialized from ASCII files that use this format. The file **People.array** contains an NSArray of NSDictionaries.

## Scripts in the Registration Application

The Registration application includes the scripts **Application.wos**, **Main.wos**, and **Registrants.wos**. The contents of these scripts are listed below.

### Application.wos

The application script **Application.wos** creates a RegistrationManager object **manager** that's used by the **Main.wos** and **Registrants.wos** component scripts to register new users and return a list of all registrants.

```
id manager;  
  
- awake  
{  
    manager = [RegistrationManager manager];  
}
```

**Main.wos**

The **Main.wos** script includes methods for registering a new user, clearing the forms on the page, and returning a page that lists all of the people who have registered.

```
id newPerson;
id message;

- awake
{
    if (!newPerson) {
        newPerson = [NSMutableDictionary dictionary];
    }
    message = @"";
}

/*
 * Ask the RegistrationManager manager object to write the user's data
 * to a file. Set the value of the message string based on the results
 * of the attempted registration.
 */
- register
{
    // Set message from the validation dictionary.
    id aPerson, results;
    aPerson = [Person personWithDictionary:newPerson];
    results = [[WOApp manager] registerPerson:aPerson];

    if ([[results objectForKey:@"isValid"] isEqual:@"No"])
        message = [results objectForKey:@"failureReason"];
    else
        message = @"You have been successfully registered.";
}

/*
 * Clear all of the forms on the page.
 */
- clear
{
    [newPerson setObject:@"" forKey:@"name"];
    [newPerson setObject:@"" forKey:@"email"];
    [newPerson setObject:@"" forKey:@"address"];
    message = @"";
}

/*
 * Return a page listing all of the people who have registered.
 */
- showRegistrants
{
    id registrants = [WOApp pageWithName:@"Registrants"];
    return registrants;
}
```

### Registrants.wos

The **Registrants.wos** script accesses the list of all registered people through the application's **manager** object. The **Registrants** component uses a **WORepetitionElement** (declared in **Registrants.wod**, not shown) to iterate over all of the names in the list. The **anItem** variable maps to a declaration in **Registrants.wod** that defines a single element in the **WORepetition**.

```
id anItem;
id myNamesArray;

- awake
{
    myNamesArray = [[WOApp manager] registrants];
}
```

### Accessing Compiled Code From a Script

**Application.wos**, **Main.wos**, and **Recipients.wos** all send messages to compiled code. Accessing compiled code from a script is simply a matter of getting an object of the compiled class and sending it a message. For example, the **Main.wos** script includes these statements:

```
// Return a Person object by invoking Person's personWithDictionary: method
aPerson = [Person personWithDictionary:newPerson];

// Register aPerson by invoking RegistrationManager's registerPerson: method
results = [[WOApp manager] registerPerson:aPerson];
```

### Accessing Script Methods from Compiled Code

To access a scripted object's methods from compiled code, you simply get the object that implements the method and send it a message:

```
// Get the object
id mainPage = [WOApp pageWithName:@"Main"];

// Send it a message
[mainPage setMessage:@"You have won a trip to Hawaii!!"];
```

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

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

For example, suppose you have a component called `Main` that includes the method `setMessage`: (this could be an implicit variable accessor method or an explicitly implemented method in your script). To access this method from your compiled code, you'd have to include a declaration such as the following in your code's implementation file:

```
@interface Dummy:WOComponentController
- (void)setMessage:(NSString *)aMessage;
@end
```

While it's certainly straightforward to access a scripted object's methods from compiled code, you may not want to have that degree of interdependence between your scripts and your compiled code. In the Registration application, `Person`'s `validate` method could have directly set the value of the `message` variable in `Main.wos`. Instead, `validate` puts its results into an `NSDictionary` that it then returns. Likewise, you may want to minimize the interdependencies between your scripts and your compiled code to facilitate reusability.

## Using C and C++ in WebObjects Applications

In addition to using compiled Objective-C in WebObjects applications, you can also use compiled C or C++. The interface you provide to WebObjects must be in Objective-C since WebObjects can't invoke C or C++ functions. However, you can directly invoke C and C++ functions from Objective-C.

Some of the options for integrating C or C++ code into your application are as follows:

- Putting the C or C++ functions into the same file as your Objective-C code
- Putting the C or C++ functions in separate files and importing their headers into your Objective-C code
- Adding a third-party library to your project and importing its headers into your Objective-C code

## Debugging

You use the debugging facilities provided in your development environment to debug your compiled code. However, debugging the scripted portions of your application requires a different approach.

WebScript provides methods that are useful for debugging: `logWithFormat:`, and several trace methods. Using these methods in conjunction with launching your application from a command shell provides you with a fairly complete picture of your running application.

### `logWithFormat:`

The WebScript method `logWithFormat:` writes a formatted string to `stderr`. Like the `printf()` function in C, this method takes a format string and optionally, a variable number of additional arguments. For example, the following code excerpt prints the string: “The value of `myString` is Elvis”:

```
myString = @"Elvis";
[self logWithFormat:@"The value of myString is %@", myString];
```

When this code is parsed, the value of `myString` is substituted for the conversion specification `%@`. The conversion character `@` indicates that the data type of the variable being substituted is an object (that is, of the `id` data type).

Because WebScript only supports the data type `id`, the conversion specification you use must always be `%@`. Unlike `printf()`, you can't supply conversion specifications for primitive C data types such as `%d`, `%s`, `%f`, and so on.

Perhaps the most effective debugging technique you can use in WebScript is to use `logWithFormat:` to print the contents of `self`. This causes WebScript to output the values of all of your variables. For example, putting the statement:

```
[self logWithFormat:@"The contents of self in register are %@", self];
```

at the end of the `register` method in the Registration application's `Main.wos` script produces output that resembles the following:

```
The contents of self in register are <WOWebScriptComponentController 0xafe04
  message = You have been successfully registered.
  newPerson = {
    address = "Graceland\015\nNashville, TN";
    email = "elvis@graceland.com";
    name = Elvis;
  }>
```

To see the output from **logWithFormat**: statements, you have to run your application from a command shell, as follows:

1. Locate the application executable.

If you don't have compiled code and haven't built a custom executable, use the `DefaultApp` program located in **NextLibrary/WebObjects/Executables**.

2. Change directories to the directory in which the application executable is located.
3. Start the application by invoking the executable as follows:

```
ApplicationExecutable RelativeApplicationDirectory
```

You must provide a minimum of one argument to the executable: the application directory relative to `<DocumentRoot>/WebObjects`. For example, the resources for `HelloWorld` are located in `<DocumentRoot>/WebObjects/Examples/HelloWorld`, so `HelloWorld`'s relative application directory is **Examples/HelloWorld**. You'd use the following command to start `HelloWorld`:

```
DefaultApp Examples/HelloWorld
```

To start a compiled application such as `Registration`, you'd use the command:

```
Registration MyApplications/Registration
```

assuming you've placed `Registration` in a directory called **MyApplications**.

4. In your browser, open the URL you'd normally use to launch your application:

```
http://myHost/cgi-bin/WebObjects/MyApplications/Registration
```

As your application runs, the output from **logWithFormat**: and other information about your application is displayed in the command shell window.

## Trace Methods

WebScript provides trace methods that log different kinds of information about your running application. The trace methods are described in the following table:

Method	Description
<code>trace:</code>	Enables all tracing.
<code>traceAssignments:</code>	Logs information about all assignment statements.
<code>traceStatements:</code>	Logs information about all statements.
<code>traceScriptedMessages:</code>	Logs information when an application enters and exits a scripted method.
<code>traceObjectiveCMessages:</code>	Logs information about all Objective-C method invocations.

To use any of the trace methods, you must run your application from a command shell.

You use the trace methods in either the `awake` or the `willPrepareForRequest:inContext:` method:

```
- awake {
    [self traceAssignments:YES];
    [self traceScriptedMessages:YES];
}
```

## Summary

### When Do I Use Compiled Code?

The two primary reasons for using compiled code are boosting performance and being able to use your own custom classes.

You use compiled code when you want to subclass `WOComponentController`, `WOWebScriptApplication`, or `WODynamicElement`. You also use compiled code to provide your own custom business classes.

### How Should I Partition My Application?

There are no hard and fast rules about how you organize a WebObjects application. However, it's common to implement your interface logic in WebScript and your business logic in compiled code.

## What Do I Need to Do to Produce Compiled Code that Can Be Used in a WebObjects Application?

To create compiled code that can be integrated into a WebObjects application, you need to follow these basic steps:

1. Use your development environment to create a project.
2. Implement a `main()` function.
3. Add to your project the libraries to which your application needs to link.
4. Create your classes and add them to your project.
5. Compile and link your code.

## How Do I Access Compiled Code from Scripts?

You access compiled code from a script by getting an object of the class and sending it a message. For example:

```
// Return a Person object by invoking Person's dictionaryWithNewPerson: method
aPerson = [Person dictionaryWithNewPerson];

// Send the object a message
[Person validate];
```

## How Do I Access Scripts from Compiled Code?

To access a scripted object's methods from compiled code, you get the object that implements the method and then send it a message:

```
// Get the page object
id mainPage = [WOApp pageWithName:@"Main"];

// Send it a message
[mainPage setMessage:@"You have won a trip to Hawaii!!"];
```

To avoid compiler warnings, you can declare the scripted methods you invoke in your compiled code.

### Can I Use C and C++ In a WebObjects Application?

Yes, but the interface you present to WebObjects must be Objective-C. You can integrate compiled C and C++ into your application in any of the following ways:

- Put the C or C++ functions into the same file as your Objective-C code
- Put the C or C++ functions in separate files and importing their headers into your Objective-C code
- Add a third-party library to your project and importing its headers into your Objective-C code

### What Is the Most Efficient Way to Debug the WebScript Portion of My Application?

You debug your compiled code using the tools provided in your development environment. To debug the scripted portion of your application, the best technique is to use the **logWithFormat:** method. It's especially effective to use **logWithFormat:** to print the contents of **self**—this outputs all of the variables' values.

To see the output from **logWithFormat:**, you must run your application from the command line.