Compiling and Debugging a WebObjects Application

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

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

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:

- WOComponent (see the HelloWorldObjC sample application for an example)
- WOApplication (see "Managing State" for an example)
- WOSession
- WODynamicElement

Note: If you subclass WOApplication or WOSession, name the subclass Application and Session respectively. If you follow this naming convention, you are able to add further functionality to your object in **Application.wos** or **Session.wos**. If you name your subclass something else, you shouldn't use **Application.wos** or **Session.wos**.

Creating a script file is the same as subclassing. For example, writing methods in **Application.wos** is the same as subclassing WOApplication. Creating a component is the same as subclassing WOComponent. Usually, the only reason you might want to create a subclass in Objective-C is to improve performance.

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 frameworks 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.

On Windows NT and Mach platforms, you can use the Project Builder application to create the project. Set the project's type to be "WebObjectsApplication" in the New Project Panel.

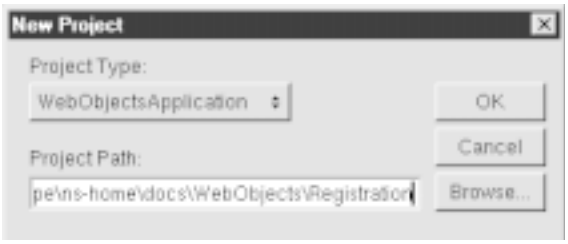

Figure 1. Creating a Project

Setting a project's type to "WebObjectsApplication" does much of the remaining work for you. It creates the following:

- the **main()** function (under Other Sources)
- an empty application script (under Other Resources)
- an empty session script (under Other Resources)
- an empty Main component (under Interfaces)

In addition, it sets up the makefiles and adds the appropriate frameworks to the project so that they are linked in when the executable is built.

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 and application objects used in your application. (If you're using Project Builder, it creates the **main()** function for you.)

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 example, **main.m**).

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

2. Add the following text to the file:

```
#import <WebObjects/WebObjects.h>
#import <Foundation/Foundation.h>

void main (int argc, const char *argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    WOApplication *application = [[[WOApplication alloc] init] autorelease];
```

```
    [application run];

    [pool release];
    exit(0);
}
#ifdef WIN32
#import <EOAccess/EOAccess.h>

void _referenceEOFrameworks()
{
    static id a;
    a = [EOEntity new];          // EOAccess
}
#endif
```

The **main()** function begins by creating an autorelease pool that's used for the automatic deallocation of objects that receive the **autorelease** message. Next, it creates the WebObjects application object and runs it. The **run** method starts the request-response loop. 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.

If you intend to run the application on the Windows NT platform, you must include the **_referenceEOFrameworks()** function as well. This function makes sure the Enterprise Objects DLLs are loaded at the appropriate time.

3. Add the file to your project.

Adding Frameworks

The next step is to add to your project the frameworks to which your application needs to link. If you're using Project Builder, the frameworks are included for you. The table below lists frameworks Project Builder includes in a WebObjectsApplication project.

Framework	Contains
WebObjects	The WebObjects classes such as WOComponent, WOApplication and WOSession.
Foundation	Contains the root class, NSObject, plus other basic object classes (such as NSString, NSDictionary, and NSNumber) that most applications use.
EOAccess	Contains the Enterprise Objects Access Layer. You need this framework only if you are writing a database application or if you are writing any WebObjects application that runs on Windows NT.
EOControl	Contains the Enterprise Objects Control Layer, which is the bridge between the database access layer and the interface layer. You need this framework only if you are writing a database application.

Frameworks are installed in ***NeXT_Root*/NextLibrary/Frameworks** (where ***NeXT_Root*** is defined at installation time). Link against a framework using the **-framework** option of the gcc compiler. For example:

```
/bin/cc -o executableName  -LobjectFiles -framework WebObjects -framework
Foundation
```

Be sure to link to the WebObjects framework before the Foundation framework.

On the Solaris platform, you also need to link to the framework **NextLibrary/PrivateFrameworks/MultiScript.framework**. If you use the provided makefiles, this is set up for you.

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.

Creating Your Components

You can create the scripted components in WebObjects Builder as you normally would. There's no need to add these to the project, as they don't get built with the compiled code. If you're using Project Builder, the **Main.wo** component is added anyway under Interfaces.

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. 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 use to run the application from a command prompt.

Deploying an Application With Compiled Code

When your application is ready to be deployed, install it in ***NeXT_Root*/NextLibrary/WOApps**.

If the application's executable is in **WOApps**, you can also place the application's **.woa** directory in **WOApps**. This is one way to ensure the **.woa** directory's privacy; if you place the **.woa** under the document root and outside users have read access on **.wos** and **.wod** files, they have access to the application's source.

If the application imports any images or sounds, you must leave a "sparse" copy of the application in the document root so that the client's browser can find these resources. In this case "sparse" means that the application's directory structure is reproduced in the document root, but the only files it contains are the static resources that the server must dispense to a client's browser.

Note: You can't autostart an application installed in **WOApps**. It must be started from the command line as described in "Launching From the Command Line."

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. Figure 2 shows the first page of the application.

Figure 2. The Registration Application

The following table lists the major files in the Registration application:

File	Description
Registration.m	Defines the **main()** function, which creates autorelease pool 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 **People.array** 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.
Main.wos	The script for the application's first page. **Main.wos** has an associated declarations file (**Main.wod**) and HTML template (**Main.html**).
Registrants.wos	The script for the application's Registrants page, which lists all of the registered people. **Registrants.wos** has an associated declarations file (**Registrants.wod**) and HTML template (**Registrants.html**).

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 <WebObjects/WebObjects.h>
#import <Foundation/Foundation.h>


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

```
- (NSString *)name;
- (NSDictionary *)personAsDictionary;

@end
```

Person.m

```
// Person.m
#import "Person.h"

@implementation Person

+ personWithDictionary:(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 <WebObjects/WebObjects.h>
#import <Foundation/Foundation.h>

@class Person;

@interface RegistrationManager : NSObject
{
    NSMutableArray *registrants;
}

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

@end
```

RegistrationManager.m

```
// RegistrationManager.h
#import "RegistrationManager.h"
#import "Person.h"

@implementation RegistrationManager

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

```
- init
{
    NSString *path = [[WOApplication application] pathForResourceNamed:@"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 = [[WOApplication application] pathForResourceNamed:@"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;
}

- (NSArray *)registrants
{
    return registrants;
}


@end
```

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

```
NSString *path = [[WOApplication application] pathForResourceNamed:@"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** can be initialized from the file **People.array** is because the file contains data in a property list format. A property list is a compound data type that consists of NSStrings, NSArrays, NSDictionaries, and NSDatas. Property lists can be represented in an ASCII format, and property list objects such as NSDictionaries and NSArrays 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;

- init
{
    [super init];
    manager = [[RegistrationManager manager] retain];
    return self;
}

- dealloc
{
    [manager release];
    [super dealloc];
}
```

Main.wo

Figure 3 shows the **Main.wo** component as it appears in WebObjects Builder.

Figure 3. Main.wo component

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;
```

```
- init {
   [super init];
   if (!newPerson) {
        newPerson = [NSMutableDictionary dictionary];
   }
   message = @"";
   return self;
}

- dealloc {
   [newPerson release];
   [message release];
   [super dealloc];
   }

/*
 * Ask the RegistrationManager 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
{
   id aPerson, results;
   aPerson = [Person personWithDictionary:newPerson];
   results = [self.application.manager registerPerson:aPerson];

  // Set message from the validation dictionary.
  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 = [self.application pageWithName:@"Registrants"];
  return registrants;
```

```
}
```

Registrants.wo

Figure 4 shows the **Registrants.wo** component as it appears in WebObjects Builder.

Figure 4. Registrants.wo component

The **Registrants.wos** script accesses the list of all registered people through the application's **manager** object.

```
id currentItem;
id myNamesArray;

- awake {
  myNamesArray = [self.application.manager registrants];
}
```

Accessing Compiled Code From a Script

Application.wos, **Main.wos**, and **Registrants.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 = [self.application.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. If you're accessing a method in the application or session script, you can use WOApplication methods to access the object:

```
[[WOApplication application] applicationScriptMethod];
[[[WOApplication application] session] sessionScriptMethod];
```

To access a component's methods, you must store the component in the session and then access it through the session.

For example, suppose you wanted to rewrite the Registration application so that Person's **validate** method directly sets the value of the **message** variable in **Main.wos**. You'd add the following statement to the **init** method **Main.wos**:

```
// Store the component in the session.
[self.session setObject:self forKey:@"Main"];
```

and then you can access it in Person's **validate** method this way:

```
// Get the component from the session:
WOComponent *mainPage = [[[WOApplication application] session]
     objectForKey:@"Main"];

// Send it a message
[mainPage setMessage:@"You must supply a name and address"];
```

(**Main.wo** implicitly implements the **setMessage:** method because it declares a variable named **message**.)

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 the example above, you'd add the following declaration to the **Person.m** file:

```
@interface WOComponent (RegistrationMainComponent)
- (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. You may want to minimize the interdependence 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 because 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 WebScript

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.

Launching From the Command Line

To debug your application, you should launch it from the command line so that you have better control over the executable and so that you'll be able to see messages written to standard output or standard error.

To start a WebObjects application from the command line:

1. Locate the application executable.

   If you don't have compiled code and haven't built a custom executable, use the **WODefaultApp** executable located in ***NeXT_Root*/NextLibrary/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 `-d` *DocumentRoot RelativeApplicationDirectory*

   You must provide a minimum of two arguments to the executable: the HTTP server's document root and the application directory relative to **<DocumentRoot>/WebObjects**. For example, the resources for HelloWorld are located in **<DocumentRoot>/WebObjects/Examples/HelloWorld.woa**, so HelloWorld's relative application directory is **Examples/HelloWorld**. (You must leave off the **.woa** extension.) You'd use the following command to start HelloWorld:

   ```
   WODefaultApp.exe -d c:/netscape/ns-home/docs Examples/HelloWorld
   ```

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

   ```
   Registration.exe -d c:/netscape/ns-home/docs MyApplications/Registration
   ```

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

   Note: If you're using Windows NT, be sure to use forward slashes in the arguments to the application executable, even if you're running the application from the DOS Command Prompt.

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

   ```
   http://localhost/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.

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;
}>
```

Trace Methods

WOApplication provides trace methods that log different kinds of information about your running application. These methods are useful if you want to see the call stack. The trace methods are described in the following table:

Method	Description
trace:	Enables all tracing.
traceAssignments:	Logs information about all assignment statements.
traceStatements:	Logs information about all statements.
traceScriptedMessages:	Logs information when an application enters and exits a scripted method.
traceObjectiveCMessages:	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 wherever you want to turn on tracing. Usually, this is in the **init** method of a component or the application:

```
- init {
  [self.application traceAssignments:YES];
  [self.application traceScriptedMessages:YES];
}
```

Debugging a Compiled Application

If you have an application that contains both compiled code and WebScript, do the following:

1. Launch your debugger.

2. Set breakpoints in the compiled code.

3. Launch the application's executable in the debugger.

   For example, if you are using **gdb**, you would type the following:

   ```
   (gdb) run -d /NextLibrary/WebServer/htdocs MyApplications/Registration
   ```

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

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

To debug the WebScript portion of the application, you still use **logWithFormat:** and trace statements as described in "Debugging WebScript." The output from these messages is displayed wherever your debugger displays standard error messages.

Summary

When Do I Use Compiled Code?

The primary reason for using compiled code is to boost performance.

You use compiled code when you want to subclass WOComponent, WOApplication, WOSession, 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.

If you use Project Builder, steps 2 and 3 are done for you.

How Do I Deploy a Compiled Application

Install a compiled application in ***NeXT_Root*/NextLibrary/WOApps**. You may store both the executable and the scripted part of the application in WOApps or just the executable. Image files and other resources must be installed in directories relative to the document root. (That is, the application's directory structure must be mirrored in the document root and must contain the resource files in the appropriate places.)

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 personWithDictionary: method
aPerson = [Person personWithDictionary:newPerson];

// 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 store the object in the session, get the object from the session, and then send it a message. Add this statement to the component's script file:

```
// Store the component in the session.
[self.session setObject:self forKey:@"Main"];
```

and then you can access it in the code this way:

```
// Get the component from the session
id mainPage = [[[WOApplication application] session] objectForKey:@"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 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.

If you want to trace the program flow, you can use the trace methods provided in WOApplication.