

# Chapter 4

## Building Projects

At this point you've created a project, added files to the project, and written source code. If you're making a Core Data application, you've created your data model as well. Now you need to take what you've created and make it into a working program. This chapter shows you how to build your project into a working program.

### Targets

Xcode projects consist of one or more targets. A target is a set of instructions to build a final product from the files in your project. Common final products are applications, frameworks, and libraries. To get as much as you can out of Xcode, you need to learn about targets and what you can do with them. In this section you'll learn about the following target-related topics:

- Inspecting and modifying target settings.
- Adding targets to your project.
- The types of targets you can add to your project.
- The build phases that make up building a target.

### Inspecting Target Settings

Xcode targets have lots of settings for you to configure. To inspect and edit a target's settings, open the target's inspector by choosing Project > Edit Active Target. If your project has multiple targets, you can open the inspector for other targets by selecting the target from the Groups and Files list and clicking the Info button in the project window toolbar. The target inspector has five tabs.

- General
- Build
- Rules
- Properties
- Comments

Legacy targets and external targets do not have the Build, Rules, and Properties tabs. To change the settings for a legacy target or an external target, double-click the target in the Groups and Files list. A window will open for you to change the target's settings.

## General

The general inspector is where you specify the target's name, direct dependencies and linked libraries. A *direct dependency* is a target Xcode must build before it builds the current target. Your project must have multiple targets before you can set direct dependencies. Aggregate targets require direct dependencies. The direct dependencies are the targets that make up the aggregate target.

Another common case where a target uses direct dependencies is when an application requires a library or framework to be built before building the application. The library or framework's target is the direct dependency for the application's target.

To add a direct dependency to a target, click the + button. A window opens that has a list of your project's targets. Select a target from the list and click the Add Target button to create the dependency.

The Linked Libraries section that lists the frameworks and libraries that will be linked to create this target. This section provides a convenient way to add libraries to the target. Click the + button to add a framework or library. A sheet of all frameworks, dynamic libraries, and object files for the SDK you're using will appear. Select the frameworks, libraries, and object files you want to add and click the Add button to add them to the project and target.

If you want to add a library or framework that's not part of your SDK, click the Add Other button in the sheet. Navigate to the library or framework you want to add and click the Add button.

## Build

The build inspector is where you configure the build settings for a target. There are two places you can configure build settings: the target's inspector and the project's inspector. Both inspectors have the same settings to configure. The build settings for a project apply to all targets in the project. The build settings for a target apply to that target only. Modifying a build setting in the target overrides whatever value you gave to the build setting in the project. Where should you configure a build setting, in the target or in the project?

Configure a build setting in the target when you want that setting to apply only to that target. An example of a build setting you should configure in the target is the product name, which is the application's name for application projects. Each target should have its own product name. Giving every target in your target the same product name makes little sense.

Configure a build setting in the project when you want that setting to apply to all targets. An example of a build setting you should configure in the project is header search paths. When adding header search paths, you normally want Xcode to use them for all targets, not a single target.

If you're not sure where to configure the build setting, configure it in the project. Read the "Build Settings" section later in this chapter for information on individual build settings.

## Rules

The rules inspector is where you specify the programs Xcode should use to process files. A lot of the rules are no-brainers; Xcode has rules to compile data model files with the data model compiler, Interface Builder files with the Interface Builder compiler, and DTrace source files with DTrace. Most of you won't have to deal with the rules inspector. The most common case where you would need to add rules is when you have source code files in your project that Xcode doesn't support natively. Create a rule that tells Xcode to use the appropriate compiler for those source code files.

Click the + button to add a rule. Two pop-up menus appear. The first menu is where you specify the file type. If your file type doesn't appear in the menu, choose Source files with names matching. Use the text field to enter the file names. Two common prefixes for programming languages are `sourcecode` and `text.script`. To see some examples, open Xcode's preferences and click the File Types button in the toolbar.

The second menu is where you specify the compiler for the file. If the compiler you want is not in the menu, choose Custom script. Enter the script in the text field. Use the + button underneath the text field to add output files for the script to create.

You can also modify the built-in rules. When you try to change one of the rules Xcode supplies for your project, an alert opens. The alert tells you that you must make a copy of the rule before changing it. Click the Make a Copy button to create the copy. Use the copy to change the rule.

## Properties

The target properties inspector is where you modify the target's property list. The property list tells the Finder information about the program such as its name, its icon, its version, and the types of document files it can open. Projects that don't have a property list, such as command-line tool projects, do not have a properties inspector.

The target properties inspector has three areas. At the top are general settings. In the middle are Cocoa-specific settings. At the bottom are the document types the program handles.

### General Property Settings

The general property settings provide basic information about the target. The Executable field is the name of the target. It has the following value:

`${EXECUTABLE_NAME}`

Don't change this field. If you want to change the name of the application, change the Product Name build setting.

The Identifier field is a string that uniquely identifies your program; Mac OS X uses it to create preference files. Apple recommends the identifier take the form:

`com.CompanyName.ProgramName`

The Type field is a four-character code identifying the kind of target: application, bundle, framework, library, etc. I can't see why you would need to change the target type. You'd be better off creating a new target.

The Creator field is a four-character code identifying your application as the creator of any document files the application creates. Ignore this field. Mac OS X 10.6 ignores creator codes so there is little point in giving your application a creator code.

If you designed an icon for your application, you would type its file name in the Icon File field. If you don't specify an icon file, you will get the default application icon.

The Version field lets you specify a version number for your program. Don't worry about program versions until you're close to finishing the program.

### Cocoa-Specific Settings

Only Cocoa programs use the Principal Class and Main Nib File fields. In most cases the principal class is `NSApplication`. The main nib file is the nib file the application loads when the user launches the application.

### Document Types

The Document Types table lists the file types your program supports, the file formats it can read and write. Table 4.1 lists the information you can set for document types. You will have one document type for each file format your application can read or write. If your program doesn't read or write files, you can ignore the Document Types table.

To add a document type, click the + button in the lower left corner of the panel. To remove a document type, select it and click the minus button. To modify any of the columns besides Role and Package, double-click an entry and type in the new information. Use the pop-up cell to change the role, and use the checkbox to change the package information.

**Table 4.1 Document Type Fields**

Field	Description
Name	The name you want to give to the document type. A Quake level editor might have the document name Quake Level.
UTI	A list of uniform type identifiers (UTI). A UTI is a string that uniquely identifies an abstract type, such as a file format. If you have a custom document type, it should take the form <code>com.CompanyName.DocumentType</code> .
Extensions	The file extension for the document type. Don't place a period before the extension. Adobe Acrobat would have a document type with extension <code>pdf</code> .
MIME Types	List of MIME (Multimedia Email Extensions) file types, file types a web browser uses. A PDF file has the MIME type <code>application/pdf</code> and a JPEG file has the MIME type <code>image/jpeg</code> .
OS Types	Four-character codes for the document's file type. Before Mac OS X, Mac applications used OS Types instead of file extensions to identify themselves as the creator of a particular file. Today, you should use file extensions and UTIs to identify a document type.
Class	The subclass of Cocoa's <code>NSDocument</code> class. Only Cocoa applications have to worry about the Class field.
Icon File	If you create an icon for document files, put the icon file's name here. Documents without an icon file will have the default Apple document icon.
Store Type	The Core Data store the document uses: binary, SQLite, XML, or in-memory. Only Core Data applications need to worry about this field.
Role	Documents can have three possible roles. Editors can view, edit, and save documents. Viewers can view documents, but can't edit or save them. Documents with the None role can't view, edit, or save.
Package	If the Package checkbox is selected, the document is a file package, which is a directory of files that looks like a single file to the end user. Otherwise it's a single file.

## **Adding Targets**

When you create a project, Xcode supplies one target for the project. You can usually get away with using the target Xcode provides, but there are instances where you may need to create multiple targets. Suppose you're writing a library and you want to write a test application to make sure the code you wrote for the library is correct. In this case you would have two targets: one target to build the library and one target to build the test application.

To add a target to one of your projects, choose **Project > New Target**. The **New Target Assistant** window opens. It shows the types of targets you can add. I will discuss the types of targets you can add shortly. Select the target you want to make, and click the **Next** button. Give your target a name, and choose the project you want to add the target to. By default Xcode adds the target to the current project. Click the **Finish** button and you've created a new target.

### **Cocoa Targets**

Cocoa targets use the Cocoa framework. You can create application, dynamic library, framework, loadable bundle, object file, shell tool, static library and unit test bundle targets. A shell tool is a command-line program without a GUI.

### **Application Plug-In**

There is only one target in the application plug-in section: Automator action.

### **BSD Targets**

BSD targets use the BSD Unix APIs instead of the Cocoa framework. You can create dynamic library, shell tool, and static library targets. A shell tool is a command-line program without a GUI.

### **System Plug-In Targets**

There are two system plug-in targets. The **Generic Kernel Extension** target creates a kernel extension, and the **IOKit Driver** target creates a IOKit driver. You're more likely to create a kernel extension project than a kernel extension target.

## Special Targets

Special Targets reside in the Other section. Most of the targets you can add correspond to Xcode's project types, but there are four special targets: aggregate, copy files, external, and shell script. Use an aggregate target to build a group of targets. The aggregate target depends on the targets that make up the aggregate. When you tell Xcode to build the aggregate target, it builds each target in the aggregate target.

The copy files target copies files to a specific location. Use a copy files target when multiple targets need to copy the same set of files.

An external target uses a program other than Xcode to build the target. External build system projects have external targets.

A shell script target runs a shell script. Use a shell script target when multiple targets need to run the same shell script.

## iPhone Targets

iPhone projects can have three possible targets: application, static library, and unit testing bundle.

## Unit Testing Bundles

Unit testing involves testing your program's functions to make sure they're correct. Xcode has unit testing bundle targets for Cocoa and iPhone programs, which makes unit testing easy for Objective-C programmers. Adding the unit testing bundle target is the first step to unit testing. There are a few more steps to take.

1. Add a direct dependency for the unit test target.
2. Configure the unit test bundle.
3. Add unit testing classes.
4. Write the test cases.
5. Run the tests.

Adding a direct dependency is not a mandatory step, but it allows you to run your unit tests automatically. To be able to unit test your program, Xcode must build your program before building the unit test target. The direct dependency ensures the program builds when you build the unit test target. In this scenario your program is a direct dependency of the unit test target. To add a direct dependency:

1. Select the unit test target from the Groups and Files list.
2. Click the Info button to open the target's inspector.
3. Click the General tab in the inspector.
4. Click the + button to add a direct dependency.
5. A sheet with a list of your project's targets opens. Select a target from the list and click the Add Target button.

You must configure the unit test bundle if you're unit testing an application. To configure the unit test bundle, modify the Bundle Loader and Test Host build settings. You must set the value of these two build settings to the path to the application's executable file.

```
$(BUILT_PRODUCTS_DIR)/MyApplication.app/Contents/MacOS/  
MyApplication
```

You can avoid entering the path twice. Enter it for the Bundle Loader build setting, and give the Test Host build setting the following value:

```
$(BUNDLE_LOADER)
```

Choose File > New File to add a unit testing class to your project. The Cocoa unit testing class is in the Cocoa Class group. The iPhone unit testing class is in the Cocoa Touch Class group. When you add a unit testing class, you must tell Xcode what targets the class should be added to. The unit testing class should be added to the unit testing target only. Xcode initially does the wrong thing with regard to unit testing. It sets the class to be added to the application (or whatever the non-unit testing target is) target, not the unit testing target. Make sure the unit testing class is added only to the unit testing target.

After adding unit testing classes, write the test cases that test your program's functions. Writing test cases is beyond the scope of this book. When you're finished writing the test cases, build the unit test target to run the tests. If a test fails, it will appear as a build error.

## Duplicating Targets

Duplicate a target if you want to create a new target that differs slightly from an existing target. To create a copy of an existing target, select the target you want to duplicate from the Groups and Files list. Choose Edit > Duplicate to make the copy.



## Target Build Phases

Build phases are steps Xcode takes to build a target. Table 4.2 lists the possible build phases a target. Each target type has its own set of build phases. A Cocoa application target initially has three build phases: Copy Bundle Resources, Compile Sources, and Link Binary with Libraries. Click a target's disclosure triangle in the Groups and Files list to see the target's build phases.

**Table 4.2 Target Build Phases**

Build Phase	Description
Copy Files	Copies files from the project directory to a location you specify.
Run Scripts	Executes shell scripts when building the project.
Copy Headers	Copies header files to the proper location in the final product. If you're creating a framework, you need the Copy Headers build phase.
Copy Bundle Resources	Copies files that support your source code files, such as nib files, property list files, audio files, and image files, to the proper location in the final product.
Compile Sources	Compiles source code files.
Compile AppleScript Files	Compiles AppleScript scripts.
Link Binary with Libraries	Links the object files Xcode creates during compilation to the frameworks and libraries you added to your project.
Build Java Resources	Copies files that support your Java source code files to the proper location in the final product.
Build Resource Manager Resources	Compiles resource files that contain Resource Manager resources. Carbon applications are the targets most likely to require this build phase.

## Adding Build Phases

Choose Project > New Build Phase to add a build phase to a target. The Copy Files, Run Scripts, and Build Resource Manager Resources build phases are the ones you're most likely to add. Use the Copy Files build phase when your program needs a framework or a library to be in a specific location. Use the Run Scripts build phase if your program includes files written in languages that Xcode doesn't directly support. Write a shell script to compile the files that Xcode can't compile for you.

Use the Build Resource Manager Resources build phase for Carbon programs that use resource files. If you're going to use resource files in your program, make sure each file has the extension `.r` or `.rsrc`. When you give each resource file the proper extension, Xcode merges them into one resource file that automatically loads when your program launches.

If you read the “Special Targets” section, you remember that Xcode has copy files and shell script targets. When should you use a target and when should use a build phase? Use a target when multiple targets need to copy the same set of files or run the same shell script. Use a build phase when one target needs to copy files or run a shell script.

## **Reordering Build Phases**

When Xcode builds your project, it goes through the build phases in the order shown in the Groups and Files list. The default order works in most cases, but if you need to change the order, select a build phase and drag it to where you want it to appear in the build order.

Be careful when ordering build phases. If you place build phases in the wrong order, Xcode won't be able to build your project correctly. Placing the Link Binary with Libraries build phase before the Compile Sources build phase will cause problems.

## **Moving a File to a Different Build Phase**

When you add a file to your project, Xcode automatically places it in a build phase. Normally Xcode places the file in the right build phase, but sometimes you need to move a file to a different build phase. If you add a Resource Manager resource file to your project, Xcode adds it to the Copy Bundle Resources build phase. You must move the file to the Build Resource Manager Resources build phase so Xcode can compile the file properly. Placing a resource file in the Build Resource Manager Resources build phase tells Xcode to compile the file and merge it with any other resource files into one resource file that automatically loads.

Clicking the disclosure triangle next to a build phase shows the files that are in that build phase. To move a file to a different build phase, select the file and drag it to the build phase where you want it to appear.

## **Configuring the Compiler**

Compilers are complex pieces of software with many settings. This section shows you how to configure the compiler so Xcode builds your project the way you want it to.

## Build Configurations

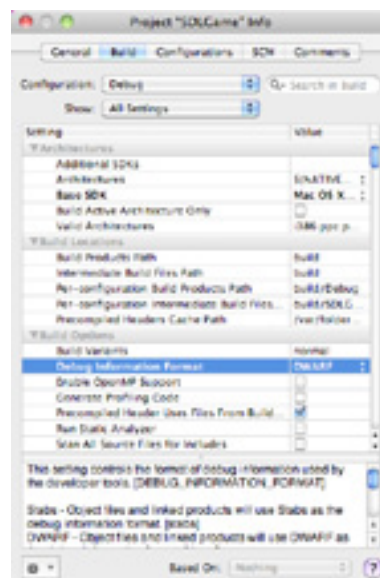
A target is a set of instructions to build a final product from the files in your project. The way you want to build the final product changes as you get closer to finishing your program. When you start writing your program, you want to turn on debugging symbols and turn off code optimization so you can examine your program and correct any mistakes you made. When you have your code ready for release, you want to turn off debugging symbols to reduce the size of the final product and turn on code optimization to make your code run faster.

Build configurations solve the problem. They allow you to build the same target in different ways, reducing the need for you to create new targets. Targets tell Xcode what to build. Build configurations tell Xcode how to build the target.

Xcode supplies two build configurations for each project: Debug and Release. The Debug build configuration contains settings that make debugging easier. The Release build configuration contains settings to build a release version of your program. To look at your project's build configurations, choose Project > Edit Build Settings.

Figure 4.1 shows the inspector for build configurations. Use the Configuration pop-up menu to select the build configuration you want to view. Xcode initially sets your project to use the Debug build configuration.

When you choose a build configuration from the Configuration pop-up menu, your choice is for viewing and making changes to that build configuration. Choosing a build configuration from the pop-up menu does not make the chosen build configuration the active one, the one Xcode uses to build your project. Choose Project > Set Active Build Configuration to change the active build configuration.



**Figure 4.1**

Build configuration inspector

To add a build configuration to the list, click the Configurations tab. Select a configuration from the list and click the Duplicate button. Enter the name you want to give the build configuration. The Debug and Release build configurations Xcode supplies are sufficient for most of you, but here are some build styles you may want to add:

- A static analysis build configuration that runs the static analyzer when building the project.
- A code coverage build configuration, which generates information for the gcov code measurement tool.
- Build styles for different code optimization levels.

To delete a build configuration, select it from the list and click the Delete button.

## **WARNING**

Xcode gives you no warning dialog asking you if you're sure you want to delete a build configuration, even for the Debug and Release configurations Xcode makes for every project. Avoid accidentally clicking the Delete button.

## **Build Settings**

Build settings have three types of controls. The type of control depends on the setting. Settings that can be either on or off have a checkbox. Select the checkbox to turn on the setting. Settings that have several possible values have a pop-up button cell. Make your choice from the menu. The rest of the settings require you to enter text. To change the settings that require you to enter text, double-click the setting from the inspector. A sheet opens for you to enter the setting's value. Click the OK button when you're finished typing.

If you're a beginning programmer, the number of settings can be intimidating, but don't freak out. Xcode configures the settings in a way that works well in general cases. You can stick with Xcode's default settings initially. If you find that Xcode isn't building your project the way you want, you can go back and configure the settings. Depending on your requirements, you'll end up not touching 80-99% of the build settings.

I'm going to focus on the build settings you're most likely to modify. If you're interested in a build setting I don't explain in the book, select it in the inspector. The bottom of the inspector will contain information about that setting.

## Architectures

The Architectures collection specifies the processor architectures to build for and the SDKs used to build the project. Xcode supports the following architectures: Intel, PowerPC, and iPhone. The Intel and PowerPC architectures are used when building Mac programs.

For Mac OS X projects, Xcode 3.2 will build a 64-bit Intel binary in the Debug build configuration. In the Release build configuration, it will build a 3-way universal binary: 32-bit PowerPC, 32-bit Intel, and 64-bit Intel. On iPhone projects, Xcode is set to build for the iPhone architecture. Xcode's initial configuration works well in most cases.

The main reason you would have to mess with the Architectures build setting is if you didn't want to build a universal binary. Click the Value menu to change the architectures. There are the following options for Mac projects:

- 32-bit Universal, which builds a 32-bit version for Intel and PowerPC Macs.
- 32/64 bit Universal, which builds 32 and 64-bit versions for Intel and a 32-bit version for PowerPC Macs.
- Intel 64-bit, which builds a 64-bit version for Intel only.
- Native Architecture of Build Machine, which builds a version for your Mac's architecture.
- Other, which lets you manually specify the architecture. This is useful if you want to build a 32-bit version for one architecture.

There are three architecture options for iPhone.

- Standard, which builds for armv6. Choose Standard to build an iPhone-only application.
- Optimized, which builds for armv6 and armv7. Choose Optimized if you want to build a universal iPhone/iPad application.
- Other, which lets you manually specify the architecture. The only reason to choose Other is if you need to build specifically for armv7.

Remember that the more architectures you build for, the longer Xcode takes to build your project. If you're building 32/64-bit Universal, Xcode has to build three versions of your program: 32-bit Intel, 64-bit Intel, and 32-bit PowerPC. Building three versions takes longer than building one. Build only for the architectures you need.

The Valid Architectures setting contains a list of architectures you can build for. Xcode provides the following valid architectures:

- `i386` is 32-bit Intel.
- `x86_64` is 64-bit Intel.
- `ppc` is 32-bit PowerPC.
- `ppc64` is 64-bit PowerPC.

- `ppc7400` is the PowerPC 7400, which appears in G4 Macs.
- `ppc970` is the PowerPC 970, which appears in G5 Macs. Because the PowerPC 970 was the only 64-bit PowerPC chip Apple ever used, there is no difference between the `ppc64` and `ppc970` architectures.
- `armv6` is the architecture for iPhone and iPhone 3G.
- `armv7` is the architecture for iPhone 3GS and iPad.

Do not modify the Valid Architectures build setting. Its purpose is to show you the valid values so you can set the architecture manually.

The Build Active Architecture Only build setting tells Xcode to build only for the active architecture, which is the architecture of your Mac initially. You can change the active architecture by choosing Project > Set Active Architecture. This setting can be used to toggle the building of universal binaries.

The Base SDK build setting can be useful if you're supporting older versions of Mac OS X. Suppose you want to support Mac OS X 10.5 and 10.6. You could set the Base SDK in the Debug version to 10.5 to make sure everything compiles and set the Base SDK in the Release version to 10.6 to take advantage of any bug fixes in the 10.6 SDK.

Additional SDKs are for sparse SDKs, which are SDKs supplied by third parties or built by you. Sparse SDKs are used more in iPhone development because dynamically linked libraries are not allowed on the iPhone.

## **Build Locations**

When Xcode builds your project, it creates files. The files Xcode creates depends on the project type and the programming languages used. Some files that Xcode creates are object files, libraries, frameworks, and executable files. Xcode initially places these files in the build folder inside your project folder. When you build the project, Xcode creates a folder with the name of the active build configuration inside the build folder and places the final build product (application, library, or framework) in that folder. If you build an application project with the Release build configuration, the application will be in the following folder:

`ProjectName/build/Release`

Where `ProjectName` is the folder containing your project.

If you don't want the build products inside the build folder, use the Build Locations collection to specify a new location for your build products. You can specify the directory where the final build product appears as well as the directory where intermediate files like object files appear.



## Build Options

The most interesting setting is the Debug Information Format setting. There are two options: DWARF, and DWARF with dSYM File. Release builds should use DWARF with dSYM because the debugging symbols get placed in an external dSYM file. Placing the symbols in an external dSYM file reduces the executable file size, which you want for release builds. Refer to the section “Choosing a Debugging Format” in Chapter 5, “Debugging with Xcode” for more information on debugging formats.

Activating the Run Static Analyzer build setting tells Xcode to run the Clang static analyzer when it builds your project. You don’t have to activate this build setting to run the analyzer. Choosing Build > Build and Analyze will run the static analyzer for you. The point of the Run Static Analyzer build setting is to automatically run the analyzer when building your project.

If you’re writing a framework or library, the Build Variants build setting will help you. There are three options: normal, debug, and profile. The normal variant builds the base library or framework. The debug variant adds debugging information to help people using the library debug their code. The profile variant adds profiling information to help people using the library measure their code’s performance.

Activating the Generate Profiling Code setting tells the compiler to generate profiling code. The only reason to activate this setting is if you want to profile your code with the command-line program `gprof`. Because `gprof` does not work on Intel Macs, you most likely will not want to activate the Generate Profiling Code setting. Apple’s profiling tools do not require Generate Profiling Code to be activated.

If you’re using OpenMP, you must activate the Enable OpenMP support build setting. OpenMP is an API for writing programs that take advantage of multi-core processors. Most Intel Macs have multi-core processors. To use OpenMP, you must compile your code with GCC 4.2. Refer to the “Compiler Version” section to learn how to set the compiler version to GCC 4.2.

## Code Signing

The Code Signing build settings collection deal with code signing, which lets you provide a signature for your application that identifies you as the author of the code. Code signing provides security benefits if your program is available for people to download. It lets the person downloading the program know the code came from you and it wasn’t altered.

There are two things to keep in mind about code signing. First, code signing is meant for applications that are available to the general public. If you’re using Xcode to learn programming or to write programs for personal use, don’t worry about code signing.

Second, code signing is something you do when your application is ready for release. Don't worry about code signing at the start of development. For those of you writing iPhone applications, you don't need code signing until you're ready to move your application to an iPhone for testing. When you start development, you'll be testing your application on the iPhone Simulator that comes with the iPhone SDK. You can run your application on the simulator without code signing it.

### **Code Signing iPhone Applications**

iPhone applications require code signing to load them on your iPhone and to get them to appear in Apple's App Store. To code sign an iPhone application, you must join the iPhone Developer Program and pay the annual membership fee.

Joining the developer program gets you a signing certificate, an application ID, and a provisioning profile. It also gets you access to the iPhone Developer Program Portal, which provides information on running your applications on your device and distributing your applications.

### **Code Signing Mac Applications**

Mac applications don't require code signing, but it can help for some applications running on Mac OS X 10.5 and later. Code signing helps most when updating your applications. If you use the same code signature for both versions, the operating system knows the updated version came from you and grants access to the user's keychain without asking the user to verify.

### **Creating a Code Signing Identity**

To use code signing, you must create a code signing identity, which is a certificate identifying you as the creator of your application. Launch the Keychain Access application to create a certificate. You can find it in your Applications folder under Utilities. Those of you writing Mac applications should choose Keychain Access > Certificate Assistant > Create a Certificate to create your certificate.

iPhone developers should choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority. Select the Request is Saved to disk radio button and the Let me specify key pair information checkbox. Click the Continue button and pick a location to save the certificate. Choose 2048 for Key Size and RSA for the algorithm. Click the Continue button to finish creating the code signing identity.



## Code Signing Build Settings

After creating a code signing identity, you can use Xcode's code signing build settings. There are four settings.

- Code Signing Entitlements
- Code Signing Identity
- Code Signing Resource Rules Path
- Other Code Signing Flags

Code Signing Entitlements is the name of the entitlements file. If you use the iPhone's ad-hoc distribution, you need an entitlements file. You can add one to your Xcode project by choosing File > New File. The entitlements file is in the Code Signing section under iPhone OS. There are two main scenarios where you would use ad-hoc distribution. The first scenario is beta testing your application. Ad-hoc distribution gets your application to the testers. The second scenario is if you're writing an enterprise application for your company to use. Ad-hoc distribution gets the application to the users without them having to go through the App Store.

The Code Signing Identity is the name of the certificate you created in Keychain Access. You will get a build error if the certificate is missing, invalid, or its name is misspelled.

The Code Signing Resource Rules Path contains the path to a resource rules file. Resource rules files are optional. They contain instructions for copying resources like audio and graphics files to the application bundle when building the application. You would need a resource rules file if you wanted to override the default method of copying resources to the application bundle. If you need a resource rules file, you can add one to your project by choosing File > New File. The resource rules file is in the Code Signing section under iPhone OS.

Xcode uses the command-line tool `codesign` to create code signatures. The Other Code Signing Flags build setting lets you add flags to `codesign` that aren't covered by the other code signing build settings.

## Compiler Version

The CompilerVersion build settings collection has only one setting, C/C++ CompilerVersion, which lets you choose the compiler version for C, C++, and Objective-C programs. In Xcode 3.2 the default compiler is GCC 4.2. Applications compiled with GCC 4.2 will run on Mac OS X 10.5 and later. If you need to support earlier versions of Mac OS X, set the compiler version to GCC 4.0. iPhone projects should have no need to change the compiler version.

There are two compiler versions that use LLVM: LLVM GCC 4.2 and Clang LLVM 1.0. Both versions use LLVM on the back end to perform optimizations and create object code. Applications compiled with LLVM should run faster than ones compiled with GCC. Like GCC 4.2, applications compiled with LLVM run on Mac OS X 10.5 and later.

The difference between the two LLVM compiler versions is in the front end. LLVM GCC 4.2 uses GCC for the front end while Clang LLVM 1.0 uses Clang for the front end. Compiling with Clang is faster, but Clang has one limitation. It does not currently support C++. If you're writing a C++ or Objective-C++ program, you cannot use Clang.

## **Deployment**

The Deployment collection determines how the final product is installed on the user's machine. You can tell Xcode where to install the product, the user that owns the product, the group that owns the product, and the file permissions to install the product files. Settings of particular interest are the Deployment Target build setting, the Targeted Device Family build setting, and the settings related to stripping symbols.

### **Deployment Target**

The *deployment target* is the earliest version of Mac OS X or iPhone OS your program will run on. When you create an Xcode Mac project, Xcode sets the deployment target to the version of Mac OS X you're running, which will be 10.6 if you're running Xcode 3.2, or the version of the iPhone SDK you've installed. Unless you're writing a cutting-edge application that uses the latest features in Mac OS or iPhone OS, you'll want to change the deployment target.

Changing the deployment target lets your program run on older versions of Mac OS X or iPhone OS, assuming your code uses APIs that work on the deployment target. If you use Core Animation and set the deployment target to Mac OS X 10.4, your code isn't going to run on 10.4 because Core Animation was introduced in Mac OS X 10.5.

Mac applications should set the Mac OS X Deployment Target build setting. iPhone applications should set the iPhone OS Deployment Target build setting.

### **Targeted Device Family**

The Targeted Device Family build setting is for iPhone applications. It determines what devices the application is built for. There are three possible values: iPhone, iPad, and iPhone/iPad. Choose iPhone/iPad if you want to build a universal application.

## Stripping Symbols

If you read the section on project symbols in Chapter 1, you may remember that symbols make up most of the code you write: classes, variables, functions, enumerated data types, constants, and macros. There are also debugging symbols that make it possible to debug your programs. In the Debug build configuration the project and debugging symbols get copied into the binary file. This is a good thing because having the symbols makes debugging a lot easier. Imagine how difficult debugging would be if you didn't have the names of your variables.

But when you're ready to release your program, copying the symbols becomes a problem. In a release build you want to strip the symbols out of the binary to make the file size smaller. Activating the Strip Linked Product build setting strips the linked product (application, library, or framework) of symbols. Activating the Strip Debug Symbols During Copy build setting strips symbols out of any files that are part of the Copy Files and Copy Bundle Resources build phases.

The Strip Style build setting determines what symbols get stripped. The initial value is All Symbols, which means everything gets stripped. You can also use two other styles. Non-global symbols strips all non-global symbols but saves external symbols. Debugging symbols strips debugging symbols, but saves local and global symbols. There is not much point to stripping only debugging symbols. Setting the Debug Information Format build setting to DWARF with dSYM accomplishes the same thing; the debugging symbols go into an external file.

Activating the Use Separate Strip setting tells Xcode to make a special call to the `strip` tool instead of stripping during linking.

## Kernel Module

The Kernel Module collection is used only by kernel extension projects so most of you can ignore this collection. The Kernel Module collection lets you set a kernel module's name, start routine, stop routine, and version.

## Linking

The Linking collection contains settings for the linker. When Xcode builds your project, the compiler compiles your source code files into object files. The linker links the object files with frameworks and libraries to create a final product, such as an executable file or a library.

The two settings you're most likely to use are Other Linker Flags and Dead Code Stripping. If you don't see an equivalent build setting in the Linking collection, add the flag to the Other Linker Flags build setting. Linking a library to your project is a common case where you would add a linker flag. You would use the following flag:

```
-lLibraryName
```

Activating the Dead Code Stripping build setting tells the linker to strip any unused code out of the executable. Dead code stripping is good for release builds because it can reduce the size of your executable file.

If you're writing a C++ program and you get linker errors, activating the Display Mangled Names build setting may help you. Because multiple C++ functions can have the same name, the compiler mangles the function names to give each function a unique name. Displaying the mangled names can help locate the source of a link error.

Programmers using OpenMP should use the OpenMP Linker Flags build setting to add any OpenMP flags to the linker. Make sure the Enable OpenMP Support build setting is activated. Refer to the Build Options section for more information about OpenMP.

If you're unit testing an application, supply the path to your application's executable file to the Bundle Loader build setting.

```
$(BUILT_PRODUCTS_DIR)/MyApplication.app/Contents/MacOS/  
MyApplication
```

## **Packaging**

The Packaging collection provides options on packaging the product Xcode creates when it builds your program. The most interesting setting in this collection is the Product Name build setting. The product name is initially set to be the name of your project. If you want a different name for your application, change the product name. If you're going to change the product name, you'll want to change it in the target. The target's build settings override the project's build settings.

## **Search Paths**

The Search Paths collection is where you tell Xcode to search for header files, libraries, and frameworks. If you limit yourself to Apple's frameworks, you shouldn't need to change search paths.

Programs that use third-party libraries and frameworks are the ones most likely to specify search paths. If you add a third-party library or framework to your project, you may get compiler or linker errors even though there's nothing wrong with the code. The errors may be caused by Xcode being unable to find the libraries and frameworks you added. In this case you'll need to add search paths for the libraries and frameworks you added.

## Unit Testing

Unit testing involves testing your program's functions to make sure they're correct. There are many tools available to make unit testing easier. The Unit Testing collection contains settings for using unit testing tools.

The Test Rig setting is where you specify the testing tool to use. Supply the path to the unit testing tool as the Test Rig setting's value. You should not have to modify this setting if you're using Objective-C and you use the unit testing bundle and unit test classes that come with Xcode.

Only application projects use the Test Host setting. The unit testing tool inserts the unit tests in the host. Supply the path to the application's executable file as the Test Host setting's value.

```
$(BUILT_PRODUCTS_DIR)/MyApplication.app/Contents/MacOS/  
MyApplication
```

Refer to the “Unit Testing Bundle” section earlier in this chapter for more information on unit testing.

## Versioning

The Versioning build settings collection lets you add build numbers to your software. There are two settings you'll need to change. The Versioning System build setting is initially empty. You must set it to apple-generic to turn it on. Once you turn on versioning, you should set the Current Project Version build setting. It must be a numeric value. A good starting value would be 1.

## **Code Generation**

The Code Generation collection contains build settings that affect the machine code the compiler generates. Many of the build settings in this collection help improve application performance. Most of you won't have to deal with most of the settings in this collection, but there are three settings you'll most likely use: Optimization Level, Generate Debug Symbols, and Objective-C Garbage Collection.

### **Optimization Level**

When you create a project, Xcode sets the optimization level to None for the Debug build configuration and sets the optimization level to Fastest, Smallest for the Release build configuration. The initial settings work for most cases.

The Fastest, Smallest optimization level produces the fastest code that doesn't increase code size. Smaller code generally runs faster than larger code. If you need speed more than small code size, change the optimization level for the release build from Fastest, Smallest to Fastest.

### **Generate Debug Symbols**

Activating the Generate Debug Symbols build setting tells the compiler to generate debug symbols. If you want to debug your program, this setting must be activated. Refer to Chapter 5, "Debugging with Xcode", for more information.

### **Objective-C Garbage Collection**

Objective-C 2.0 added garbage collection, which removes the need for retaining and releasing the objects you allocate in your program. Garbage collection was introduced in Mac OS X 10.5. Older versions of Mac OS X do not support Objective-C's garbage collection. The iPhone does not support garbage collection.

Garbage collection is initially turned off when you create an Xcode project. To turn on garbage collection, you must modify the Objective-C Garbage Collection build setting. The setting has three possible values.

- Unsupported, which means no garbage collection.
- Supported, which means the program uses both garbage collection and the `retain` and `release` methods. On Mac OS X 10.5 the operating system uses garbage collection and ignores `retain` and `release` calls in your code. The `retain` and `release` methods are used on older versions of Mac OS X.
- Required, which means garbage collection will be used. Applications that require garbage collection will not run on anything earlier than Mac OS X 10.5.

## Language

The Language collection contains miscellaneous settings that don't fit anywhere else. Some of the more interesting things you can do in the Language collection include the following:

- Choose the language compiler.
- Choose the language standard.
- Enable exception handling
- Set compiler flags.

### Choosing the Language Compiler

The Compile Sources As setting is where you tell Xcode the language compiler to use: C, C++, Objective-C, or Objective-C++. Initially Xcode compiles your program's files according to the extension you give the files. Files with the `.c` extension use the C compiler. Files with the `.cpp` extension use the C++ compiler. Files with the `.m` extension use the Objective-C compiler, and files with the `.mm` extension use the Objective-C++ compiler. If you're writing in C you may want to compile your programs with the C++ compiler. The C++ compiler can compile C code, and it's stricter than the C compiler. Compiling your C code with the C++ compiler can catch errors you wouldn't find with the C compiler.

### Choosing the Language Standard

The C Language Dialect setting is where you tell Xcode the language standard you want to use. The ANSI group defines the standards for the C and C++ languages. The GCC compiler adds extensions to these standards. Periodically (normally every 5-10 years) the ANSI group updates the language standards. Use the C Language Dialect setting to specify the language standard you want Xcode to use to compile your program.



### Enabling Exception Handling

The Enable C++ Exceptions setting turns on exception handling for C++ programs, and the Enable Objective-C Exceptions setting turns on exception handling for Objective-C programs. Exception handling is a way for C++ and Objective-C programs to handle errors that occur when your program is running. If your program uses `try`, `catch`, and `throw` statements, make sure you turn on exception handling.

### Setting Compiler Flags

When you configure a compiler setting from the build settings inspector, you're indirectly setting command-line compiler flags. By using the information panel to configure compiler settings, Xcode saves you from having to enter dozens of compiler flags. Xcode provides a lot of compiler settings for you, but if you need to set a compiler flag that Xcode doesn't supply a setting for, use the Other C Flags and Other C++ Flags settings. C and Objective-C programs should use the Other C Flags setting, and C++ and Objective-C++ programs should use the Other C++ Flags setting.

To set a compiler flag, select the appropriate setting and double-click the Value column. A sheet opens. Click the + button to add a flag. Enter the flag and click the OK button.

### Preprocessing

The Preprocessing collection is where you define and set the values for preprocessor macros. A macro performs text substitution. The following macro:

```
#define ARRAY_SIZE 500
```

Defines a macro `ARRAY_SIZE` that is a substitute for the value 500. Before compiling your program, the preprocessor goes through your code and replaces every instance of `ARRAY_SIZE` with the value 500.

The Preprocessing collection is for conditional macros. Suppose you add code to your program that writes debugging information to a file. As you're developing the program, you want to write the debugging information. When you have the code working properly, you want to stop writing the debugging information so your program will run faster. By defining a macro you can easily turn the debugging information on and off.



```
#define DEBUG

#ifdef DEBUG
    write debugging stuff
#endif
```

Instead of defining `DEBUG` in your code, you would define it in the Preprocessing collection for the Debug build configuration. Defining `DEBUG` tells Xcode to write the debugging information. Removing the `DEBUG` definition tells Xcode to skip over the debugging code. By defining `DEBUG` in the Preprocessing collection, you can set your project so the debug build writes the debugging information and the release build doesn't.

## Warnings

The Warnings collection provides a series of checkboxes telling the compiler when to generate warnings. Compiler warnings tell you when you're doing something that's syntactically correct, but probably wrong. Turning on warnings is a good idea during development because the warnings let the compiler tell you about possible problems in your code that would be difficult to find without the warnings. Compiler warnings can save you hours during debugging.

Selecting the Treat Warnings as Errors checkbox tells Xcode to treat compiler warnings as compiler errors. If your code generates any compiler warnings, Xcode won't compile the program, forcing you to fix the warnings to build the final product. Treating warnings as errors forces you to write clean code from the start.

If you want to turn off warnings, select the Inhibit All Warnings checkbox. I don't recommend turning off all warnings because it allows problems in your code to slip past the compiler.

If there are warning flags that Xcode doesn't provide a setting for, use the Other Warning Flags build setting to add them.

## Data Model Compiler Warnings

Only Core Data projects will have a Data Model Compiler Warnings collection. The build settings in this collection suppress warnings from the compiler when it compiles your Core Data data models.

## Interface Builder Compiler

You must have an xib file in your project for the Interface Builder Compiler collection to appear. A xib file is a nib file that is saved in XML format so version control systems can track the changes. When you build your project, the Interface Builder compiler, `ibtool`, compiles the xib file into a nib file and copies the nib file into your application bundle.

Xcode's initial settings work well for basic compiling; many of you won't ever need to change the initial settings. But if you need to customize the compilation, you must add compiler flags. If you add a flag that Xcode doesn't supply a build setting for, enter the flag in the Other Interface Builder Compiler Flags build setting.

Unless you're using third-party Interface Builder plug-ins, you shouldn't have to modify the other Interface Builder compiler build settings. If you are using plug-ins, enter them in the Plug-Ins build setting. Enter any plug-in search paths in the Plug-In Search Paths build setting.

## Conditional Build Settings

Conditional build settings are settings that apply to a particular architecture or SDK. Suppose you're writing an application that is a universal binary and you want to compile the code with Clang LLVM for the Intel version and GCC 4.0 for the PowerPC version. You would set the compiler version to Clang LLVM and add a conditional build setting for the PowerPC architecture that set the compiler version to GCC 4.0.

To add a conditional build setting, select the build setting you want to add the condition to. Click the button in the lower left corner of the inspector. Choose Add Build Setting Condition.

A condition will appear underneath the build setting. There will be one or two pop-up button cells. One will have the initial value Any SDK and lets you choose the SDK the build setting value will apply to. The second one will have the initial value Any Architecture and lets you choose the architecture. Not all build settings let you choose the architecture.

In the example I used in the first paragraph, you would select the C/C++ Compiler Version build setting and set its value to Clang LLVM 1.0. Next, you would add a build setting condition. You would leave the SDK as Any SDK and set the architecture to PowerPC. The value of the conditional build setting would be GCC 4.0.

To delete a build setting condition, select it and click the button in the lower left corner of the inspector. Choose Delete Definition at This Level.

## Adding Your Own Build Settings

After looking at all the build settings Xcode has, you might be wondering why you would need to add your own build settings. The inspector's list of build settings, while impressive, is not an exhaustive list. If you need to change a build setting that is not in the list, you must add the build setting.

You're more likely to create conditional build settings than define new build settings. The main reason to add a build setting occurs if your project has a Run Script build phase. You may want to define a build setting to use in your script.

To add a build setting, click the button in the lower left corner of the build configuration inspector. A menu will open. Choose Add User-Defined Setting.

The name of a build setting can have uppercase letters, underscore characters, and numbers in it. You cannot start a build setting name with a number. Xcode's list of build settings provides examples of legal build setting names. To see a list of Xcode's build settings, open Xcode's documentation window by choosing Help > Developer Documentation. Search for Xcode Build Setting. The *Xcode Build Setting Reference* should appear under Title on the left side of the documentation window.

To remove a build setting you created, select it and press the Delete key.

## Configuration Files

A *configuration file* is a text file that contains build settings. If you find yourself changing the same build settings for all your projects, you should use a configuration file. Add the settings you're always changing to the configuration file. Add the configuration file to your project and tell Xcode to base the build on the configuration file. Xcode will use the settings in the configuration file to build your project so you don't have to change the settings in the build configuration inspector.

### Creating a Configuration File

Because a configuration file is just a text file, you can create one in any text editor. Give the file the extension `.xcconfig` so Xcode knows the file is a configuration file. The easiest way to create a configuration file is to create it in Xcode. Choose File > New File to create a new file. Select Configuration Settings File from the list of file types. You can find the configuration settings file in the Other group under Mac OS X.

When you create a new file, Xcode sets things up so the file is added to your project's targets. You do not want to add configuration files to targets. Deselect the checkbox next to each target in your project before clicking the Finish button to create the file.

Creating a configuration file in Xcode is easy, but you're not going to want to create a new configuration file for each project you create. The point of using a configuration file is to create a list of build settings once and use that list in multiple projects. Choose Project > Add to Project to add your configuration file to other projects. Make sure you do not add the configuration file to the other projects' targets.

Xcode projects can contain multiple configuration files. You can have one configuration file of debug build settings, a second configuration file of release build settings and use both files in your project. Use the debug configuration file in your Debug build configuration, and use the release configuration file in your Release build configuration.

## What Goes in a Configuration File?

A configuration file contains a list of build settings, with one setting per line. Each setting takes the following form:

```
SETTING = value;
```

Xcode has lots of build settings, too many for me to list here. Read the *Xcode Build Setting Reference* document, which is part of the Xcode documentation, to see the list of build settings.

Selecting a build setting in Xcode's build configuration inspector shows the formal build setting name in the description at the bottom of the inspector. Build settings usually contain all uppercase letters. If you select the Optimization Level build setting, which is part of the Code Generation build settings collection, you will see the setting's formal name is `GCC_OPTIMIZATION_LEVEL`.

Suppose you want your projects to use Clang LLVM 1.0 as the compiler. You would add the following setting to the configuration file:

```
GCC_VERSION = com.apple.compilers.llvm.clang.1_0;
```

If you want to add comments to a configuration file, remember that Xcode uses the C++ `//` comment style for configuration files.

```
// This is a comment.
```

A configuration file can contain as many build settings as you want, but don't put every build setting in the configuration file. Put only the build settings you don't want to be constantly changing. If there are only three build settings you're constantly changing, put those three settings in the configuration file. Xcode will use the settings in the build configuration inspector for the rest of the build settings.

## **Telling Your Project to Use a Configuration File**

After writing your build configuration files, you must add them to your project and tell Xcode what configuration file to use. If you haven't added configuration files to your project, choose Project > Add to Project to add them, making sure you don't add them to your project's targets.

To assign a configuration setting file, open the build configuration inspector for your project. Refer to Figure 4.1 to see an example of the inspector. At the bottom of the inspector is the Based on pop-up menu. The menu contains the configuration files you added to your project. If you have not added any configuration files to your project, the menu is grayed out. Choose a configuration file from the menu. Make sure you choose a configuration file for each build configuration in your project, assuming you want to use a configuration file for all your project's build configurations.

## **Overriding the Configuration File**

Suppose you have a configuration file with 15 settings. You create a new project, but want to use only 13 out of the 15 settings in the configuration file. How do you tell Xcode to use only the 13 settings you want to use?

The solution is to use the build configuration inspector. When you change a build setting from the build configuration inspector, it overrides the setting in the configuration file. In the example from the last paragraph, you would open your project's build configuration inspector and change the two settings you want to override.

Because the build configuration inspector overrides the configuration file, be careful when you open the build configuration inspector. You don't want to accidentally override any of your configuration file's build settings.

## Compiling Your Program

After writing the source code for your program, you must compile the program to create the program and make sure everything works. Xcode calls the process building. When building your project, Xcode compiles your source code files, links them with the frameworks in your project, and creates the final product, such as an application or a library.

### Precompiled Headers

Xcode supplies a prefix file for Cocoa application projects. The prefix file contains a list of header files. Xcode precompiles the header files in the prefix file. By precompiling these header files, Xcode has less work to do when it compiles the files that include the precompiled header files. If you have a lot of source code files that include a header file, adding the header file to the prefix file speeds up the building of your project.

What header files should you put in a prefix file? Header files that multiple source files include and header files that don't change often are the best files to add to a prefix file. The header files from Apple's frameworks are great files to put in a prefix file. If you're writing a Cocoa program, most of your files are going to include the Cocoa header file `Cocoa.h`, and you're not going to make changes to `Cocoa.h`.

Why do you want to place header files that rarely change in a prefix file? Xcode recompiles the precompiled header file when the prefix file changes or the header files in the prefix file change. If you add a header file that you're constantly changing, Xcode has to recompile the precompiled header every time you change the header file. Recompiling the precompiled header is slower than not using prefix files at all.

To add header files to a prefix file, open the prefix file. The prefix file of a Cocoa application project has the name `ProjectName_Prefix.pch` and includes the Cocoa header file. Use the `#import` (Objective-C) or `#include` (C and C++) statement to add other header files.

After adding files to your prefix file, make sure Xcode is set to precompile the headers in the prefix file.

1. Select the target name from the Groups and Files list.
2. Click the Info button to open the target's inspector.
3. Click the Build tab in the inspector.
4. Make sure the Precompile Prefix Header build setting's checkbox is selected.
5. Make sure the Prefix Header build setting shows the name of your prefix file.

The Precompile Prefix Header and Prefix Header build settings are part of the Language collection.

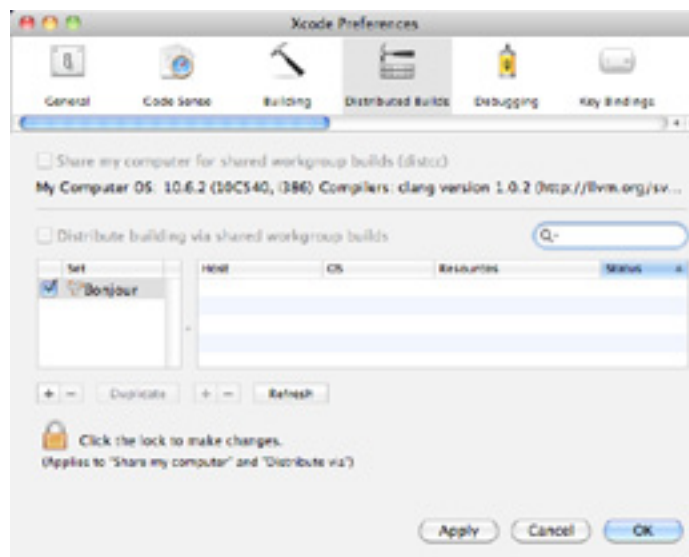
## Distributed Builds

One of the most common complaints about Xcode is its speed compiling programs. For those of you programming in C, C++, or Objective-C on a network with multiple Macs, distributed builds can reduce the time Xcode takes to compile programs. Distributed builds use other computers on the network to compile source code files. The more computers you have on the network, the faster the compilation goes. To distribute builds to another computer, that computer must be running the same operating system version and compiler version as your Mac. If you're building your program with Xcode 3.2 on Mac OS X 10.6 and the other computer is running Mac OS X 10.5 and Xcode 3.1, you won't be able to distribute your build to the other computer. To distribute builds to other computers:

1. Choose Xcode > Preferences to open Xcode's preferences panel, which you can see in Figure 4.2.
2. Click the Distributed builds button in the preferences panel's toolbar.
3. Select the Distribute via checkbox.

If the Distribute via checkbox is disabled, click the lock button at the bottom of the panel. You'll be asked for your password. After entering the password, the Distribute via checkbox should be enabled.

When you select the Distribute via checkbox, the Bonjour build set checkbox gets selected too. By selecting the Bonjour checkbox, Xcode automatically lists the computers you're connected to. For each connected computer, Xcode tells you the computer's name, the operating system it's running, the compilers installed on that computer, and its status. You can distribute builds only to computers with Sharing status. Click the Apply button to have your builds distributed to every computer with Sharing status.



**Figure 4.2**

Distributed builds panel



Having your builds distributed to every available computer on the network may not be what you want. You can create custom build sets that contain the computers you want to distribute builds to. Click the Duplicate button to duplicate an existing build set or click the + button to add a build set. Use the + and minus buttons under the host list to customize your build set. Select your build set from the build set list to distribute builds to the computers you want.

You can also choose to share your computer so other programmers can distribute their builds to your Mac. There is a checkbox at the top of the preferences panel to share your Mac for shared workgroup builds. If the checkbox is disabled, click the lock button to enable it.

## **Cleaning Targets**

Cleaning a target removes everything you previously built for a target. When you build a project with a clean target, you must recompile all your source code files so you don't want to clean a target unless it's necessary. Changing compiler settings in your project requires a clean target for the changes to take effect. Suppose you're compiling your files with the Fast optimization level and you change the optimization level to Faster to measure the speed difference between the two levels. To increase the optimization level you must recompile all the files, which requires you to clean the target.

Choosing Build > Clean cleans the active target. To clean all the targets in your project, choose Build > Clean All Targets.

## **Building Your Project**

Xcode provides five methods to build your programs.

- Compile a single file.
- Build a project.
- Build a project and run the program.
- Build a project and debug the program.
- Build and archive.

To compile a single file, select the file from the project window. Choose Build > Compile.

Choosing Build > Build builds the project. It compiles any source code files that changed since the last time you built the project. If you haven't built your project before or you cleaned the target, Xcode compiles all the source code files. After compiling the files, Xcode performs any additional steps required to create the target.



Choose Build > Build and Run to build and run your program. Building and running builds your project and runs it from Xcode. To be able to run your program, it must build successfully with no errors.

Choose Build > Build and Debug to build and debug your program. Building and debugging builds your project and runs it from Xcode's debugger. I cover debugging next chapter.

The project window toolbar also has a button that lets you build and run or build and debug your project. The Breakpoints button in the project window toolbar toggles running and debugging your program.

Build and archive is available only for iPhone application projects whose active SDK is set to the device, not the simulator. Xcode builds the project and creates an .ipa archive for you to submit to Apple for approval.

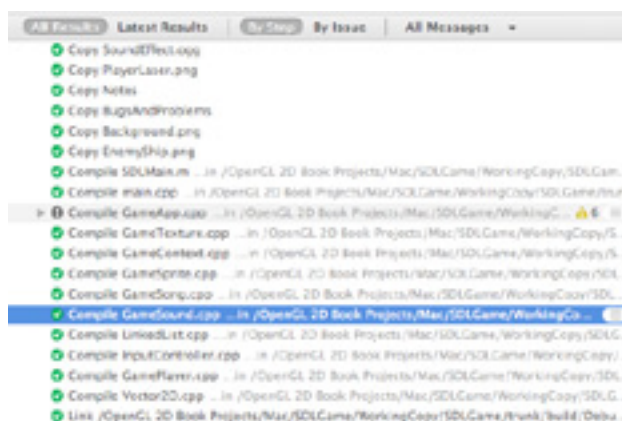
After you build your project, you may be wondering where to find what Xcode built. You can find the build product (application, framework, or library) inside your project's build folder.

```
build/Debug (debug build)
build/Release (release build)
```

If you change the Build Locations build setting, the application resides in the location you specify instead of the build folder.

## Seeing More Build Details

When you build your program, the only thing Xcode tells you is whether the build succeeded or failed. If the build failed, you want to know what caused the build to fail. To see more details of a build, open the build results window, shown in Figure 4.3, by choosing Build > Build Results.



**Figure 4.3**

Build results window

The build results area gives you a play-by-play account of the build, listing each step taken during the build. If the step was successful, it will have a green checkmark next to it. Warnings have a yellow icon next to them, and errors have a red X next to them. Errors and warnings will also have a message detailing what went wrong. Static analyzer issues have a blue icon next to them with a message detailing a problem. For more information on static analyzer issues, refer to the “Static Analysis” section later in this chapter.

If you have an issue (error, warning, or analyzer issue) during your build, select the issue from the build results window. Xcode opens the file containing the issue in the editor and takes you to the line of code with the issue. From there you can locate the cause of the issue and correct it. If you don’t see an editor, drag the splitter bar at the bottom of the build results window to show the editor. Double-clicking a step opens the file in a separate window.

## **Message Bubbles**

When there is an error, warning, or analyzer issue in your code, a message bubble appears in the line of code in the editor. The line of code is highlighted. In the editor is an icon that identifies the issue: error, warning, or analyzer issue. Clicking the icon hides the message bubble; click the icon again to show the message bubble. On the right side is the error message. Clicking the message takes you back to the build results window. If there are multiple errors in a line of code, a number appears to the right of the error message. Click the number to see all the errors for that line of code.

Xcode initially shows message bubbles for all issues. Choose View > Message Bubbles to customize the display of message bubbles. If you want message bubbles to appear for errors only, choose View > Message Bubbles > Errors Only.

## **Showing the Build Transcript**

Each build step executes commands from the command line. Xcode shields you from the details of executing commands, but the build transcript shows the executed commands. When you compile a file, the build results area tells you that Xcode compiled the file. The build transcript shows the commands Xcode called to compile the file.

Selecting a build step from the build results makes the build transcript button appear. Clicking the build transcript button shows the build transcript for that step. If the step has an error or warning, the build transcript button will appear without you having to select the step. Right-clicking a step and choosing Open These Results as Transcript Text File opens an editor window and displays the entire build log.

## Filtering

At the top of the build results window is a scope bar with three sets of controls that lets you filter what appears in the build results window. The first set determines whether to show all results or the latest results. All results shows the results of each file in the project while the latest results shows only the files that were compiled during the most recent build.

The second set determines the order Xcode displays the build results: by step or by issue. When ordering by step, the build results window lists each step in the order it was taken. When ordering by issue, the build results window groups the errors together, the warnings together, and the analyzer issues together.

The third set determines what messages get displayed in the build results window. You can display all messages, errors only, analyzer results only, errors and warnings, or issues only. Issues displays any problems: errors, warnings, and issues with the analyzer.

## Automatically Opening the Build Results Window

To automatically open the build results window, open Xcode's building preferences. Use the Open during builds pop-up menu to tell Xcode when to open the build results window. It is initially set to never, but you can tell it to open every time you build your project, to open when an error occurs in the build, and to open when an issue occurs, which means the build results window opens when there's an error, a warning, or an issue with the static analyzer.

## Tips for Correcting Build Errors

One of the most frustrating aspects of writing Mac software for people new to Mac development is getting the program to compile. Compilers can be picky, spitting out error messages on code that worked on another compiler. In this section I provide some tips to fixing the errors you get when trying to build your program.

## Add All Necessary Frameworks

Missing frameworks are the leading cause of linker errors and cause compiler errors if your program calls functions from the missing frameworks. Make sure you add the frameworks to your project.

If you're using libraries and frameworks in your project that Apple did not supply, Xcode may be unable to find them, even though you did add the libraries and frameworks to your project. You will have to add search paths for your libraries and frameworks. Refer to the "Search Paths" section earlier in this chapter for more information on adding search paths to your project.

### **Include Necessary Header Files**

Adding a framework isn't the only step you must take if you want to call the framework's functions in your program. You must remember to include the framework's header files as well. Mac OS X and iPhone OS have a slightly different way of including system headers than other operating systems. The method Mac OS X and iPhone OS uses to include system headers causes build errors if you're unaware of the method. Suppose you want to play QuickTime movies in your program. You add the QuickTime framework to your project and include the header file `Movies.h`.

```
#include <Movies.h>
```

This statement generates an error during building. Mac OS X requires you to enter the framework the header file belongs to before the header file when including a system header. The following line demonstrates the proper way to include the `Movies.h` header file:

```
#include <QuickTime/Movies.h>
```

### **The Error May Not Be Where Xcode Says It Is**

When Xcode finds a syntax error in your code, it reports the file where the error occurs along with the line number. You go to the line of code and can't find anything wrong. In this situation the error may have occurred in an earlier line of code. Check the lines of code above the line where Xcode reported the error.

### **One Error Can Cause Multiple Syntax Errors**

Sometimes one error can trigger multiple syntax errors. It can be overwhelming to look at the build transcript and see hundreds of error messages. Fix one error at a time, rebuilding the project after each fix. You may discover you have fewer mistakes in your program than you thought.

## Look for Typographical Errors

Typographical errors cause a high percentage of syntax errors. Make sure you're not missing semicolons. Make sure each left brace has a matching right brace. Make sure each left parenthesis has a matching right one. Check for misspelled variable and function names.

## Check Function Arguments

Improper function arguments cause many syntax errors. Check the arguments in your function calls. Make sure the number of arguments match, the arguments are in the right order, and the arguments have the proper data type. Pointer variables as arguments cause a lot of syntax errors. If you pass a non-pointer variable improperly, you get a syntax error.

Let's use an example to demonstrate passing a non-pointer variable improperly. The Core Foundation function `CFURLGetFSRef()` creates a file system reference from a file location. You would use the file system reference to open the file. The second argument to `CFURLGetFSRef()` is a pointer to a file system reference. If you have the following code:

```
CFURLRef theFile;
FSRef fileRef;
Boolean success;

// Code to retrieve the file location has been omitted.

success = CFURLGetFSRef(theFile, fileRef);
```

You get an error because the function `CFURLGetFSRef()` takes a pointer to `FSRef` as an argument, and you passed a `FSRef`. You must pass the address of the variable `fileRef`.

```
success = CFURLGetFSRef(theFile, &fileRef);
```

## Building for Unsupported Languages

Xcode natively supports projects written in AppleScript, C, C++, Java, and Objective-C. It can also build projects written in languages Xcode doesn't natively support. The Xcode Tools include Perl, PHP, Python and Ruby. Other popular languages are Fortran, Lua, Pascal, and Smalltalk, but you have to download a compiler to use those languages. You can even create your own programming language and build programs written in that language from Xcode. All you need is a program to do the building.

To use a language that Xcode doesn't natively support, create an external build system project. You must tell Xcode to use your language's build tool to build the project.

1. Double-click the target's name in the Groups and Files list. Doing so opens the target's settings window.
2. Select Build Tool Configuration to set the build tool, which you can see in Figure 4.4.
3. In the Build Tool text field, type the path to the program you're going to use to build the program. Initially for an external build project, the build tool is `make`, located at `/usr/bin/make`.
4. In the Arguments text field, enter any arguments you want to use to build the program. Normally this involves compiler flags you want to use when building the program.

When you build your project, Xcode uses the tool you specified to do the building.

## Static Analysis

Xcode 3.2 comes with the Clang static analyzer. The static analyzer goes through your code and looks for bugs. It can find mistakes such as memory leaks, bad pointer references, dead code, and uninitialized variables. These are problems that can be difficult to find on your own.

To run the static analyzer, choose Build > Build and Analyze. Activating the Run Static Analyzer build setting tells Xcode to run the analyzer every time it builds your project. Static analysis takes longer than compiling so you may not want to automatically run the analyzer on large projects.

The status bar will tell you if the analyzer found any problems. Open the build results window (choose Build > Build Results) to examine the issues the analyzer uncovered. The static analyzer provides more detailed information than a compiler error message. A compiler error message tells you the error and the line of code where the error occurred. The static analyzer shows the complete path of problems in your code that led to the issue. Click the disclosure triangle next to an issue to see the steps in the path.



**Figure 4.4**

Build tool for external target

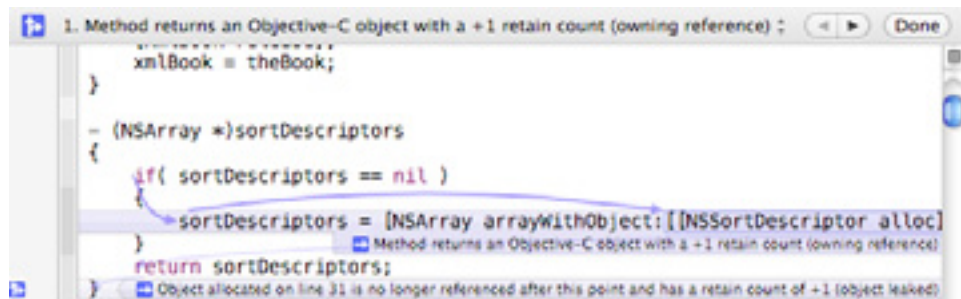
When you select an analyzer issue from the build results window, Xcode opens the file containing the issue in the editor and takes you to the line of code with the issue. From there you can locate the cause of the issue and correct it. If you don't see an editor, drag the splitter bar at the bottom of the build results window to show the editor. Double-clicking an issue opens the file in a separate window.

Each analyzer issue has a message bubble in the editor. In the gutter is an icon. Clicking the icon hides the message bubble; click the icon a second time to show the message bubble. The line of code is highlighted in the editor. To the right of the code is the analyzer message. If a line of code has additional steps in its path of problems, its icon contains two arrows: one facing up and one facing right. Click the analyzer icon to the left of the message to see the additional steps, which you can see in Figure 4.5. You'll notice the icon changes to a right-facing arrow.

When a line of code has additional steps in its path of problems, a scope bar appears under the editor's navigation bar. Use the menu on the left side to pick a step to analyze, or use the left and right arrow buttons to navigate the various steps in the path.

## Creating Universal Binaries

Universal binaries are binaries that support multiple architectures. A Mac universal binary supports both Intel and PowerPC Macs. An iPhone universal binary supports both iPhones and iPads. Xcode takes away much of the pain involved in configuring projects to create universal binaries.



**Figure 4.5**

Static analysis expansion of a multi-step issue



## Creating Mac Universal Binaries

Creating a Mac universal binary that runs on both Intel and PowerPC Macs is simple. Use the Release build configuration. The Release build configuration is configured to build a three-way universal binary: 32-bit Intel, 64-bit Intel, and 32-bit PowerPC.

## Creating iPhone Universal Binaries

Creating a universal binary that works on both iPhones and iPads isn't too difficult. Either create a window-based application project or take an iPhone project and upgrade it to a universal binary.

### Creating a New iPhone Project

The easiest way to create a universal iPhone application is to create a window-based application. Choose Universal from the Product pop-up menu, and you're done.

If you choose one of the other iPhone application templates, there's more work because there is no Universal item in the Product pop-up menu. Your only choices are iPhone and iPad. Choose iPhone and upgrade the project to add iPad support.

### Upgrading an Existing iPhone Project

Open the project and select the target from the Groups and Files list. Choose Project > Upgrade Current Target for iPad. You have the option of creating one universal application or two device-specific applications: one for iPhone and one for iPad. If you choose to create a universal application, Xcode adds an iPad-specific version of the xib file for the main window, `MainWindow-iPad.xib`, to the project.

If you choose to create two device-specific applications, Xcode creates a second target for the iPad application. Xcode also creates a copy of each source code file (except `main.m`) and xib file for the second target. These files have the same name as their iPhone counterparts, which makes distinguishing them in the project window difficult.

### Universal Application Build Settings

When upgrading the iPhone application to a universal application, Xcode should have modified the build settings to support a universal application, but you should still check the following build settings:



- Architectures
- Targeted Device Family
- iPhone OS Deployment Target

Remember to check the build settings for the target, not the project. Target build settings override project build settings.

The Architectures build setting should be set to Optimized. The Targeted Device Family build setting should be set to iPhone/iPad. The iPhone OS Deployment Target should be set to something earlier than iPhone OS 3.2. iPhone OS 3.2 runs only on iPads. If you have different values for any of these build settings, change them.

If you choose to create two device-specific applications, you don't have to worry about these build settings. One target is set to build an iPhone application, and the second target is set to build an iPad application.

## Developing for Multiple Versions of Mac OS X and iPhone OS

Xcode projects are set to create applications for the version of Mac OS X you're running and the version of the iPhone SDK you've installed on your Mac. If you're writing a Mac application for personal use, Xcode's behavior isn't a problem, but if you want another person to use your application and he or she is running an older version of Mac OS X, your application will not run on his or her Mac. If you're writing an iPhone application, you want as many people as possible to download it. How do you get your application to run on older versions of Mac OS X and iPhone OS?

The solution is to set the deployment target. Xcode has two deployment target build settings: Mac OS X Deployment Target and iPhone OS Deployment Target. Both of these settings are in the Deployment build settings collection. The Mac OS X deployment target is the earliest version of Mac OS X that can run your application. The iPhone OS deployment target is the earliest version of iPhone OS that can run your application.

Keep in mind when you set a deployment target is that it assumes your application uses no features Apple introduced in a later version of Mac OS X or iPhone OS. If you're writing a Cocoa application with Objective-C 2.0, your application will not run on anything earlier than Mac OS X 10.5. Setting the deployment target to Mac OS X 10.4 isn't going to make your application run on 10.4. If you're running an iPhone Core Data application, it won't run on anything earlier than iPhone OS 3.0.

What should the deployment target be? It should be the earliest version of Mac OS X or iPhone OS you can reasonably support. For Mac applications a general rule is to support the latest version of Mac OS X plus the previous 1-2 versions. Supporting older versions

of iPhone OS is not currently as important as supporting older versions of Mac OS X. All iPhones can run iPhone OS 3.1 (iPhone OS 3.1.3 is the latest version that runs on all iPhones), and there is currently no reason why an iPhone owner can't run iPhone OS 3.1 so iPhone OS 3.1 is a suitable deployment target for iPhone applications. But there's no harm in supporting older versions of iPhone OS. If you're writing an application that runs only on iPads, set the deployment target to iPhone OS 3.2. iPhoneOS 3.2 is the earliest version with iPad support.

Suppose you're writing a Core Data application. Reasonable deployment targets would be Mac OS X 10.4 and iPhone OS 3.0. Maintaining a second codebase that doesn't use Core Data to support older versions of Mac OS X and iPhone OS wouldn't make much sense. If an Apple technology shaves months off your development time or adds something essential to your application, don't be afraid to use it and limit your application to people running newer versions of Mac OS X and iPhone OS.

## **Compiler Version**

Mac applications that use the LLVM compiler or GCC 4.2 to build them will not run on anything earlier than Mac OS X 10.5. If you want to support older versions of Mac OS X, change the compiler to GCC 4.0. iPhone applications should not need to change the compiler version.

## **SDKs**

An SDK contains everything you need to build an application for a particular version of Mac OS X or iPhone OS. The SDK for a particular version of Mac OS X or iPhone OS contains the latest technologies your application can use. Xcode projects initially use the latest SDK. In most cases there is no need to change SDKs. The SDK has no effect on what versions of Mac OS X or iPhone OS can run your program. The deployment target determines the versions of Mac OS X or iPhone OS that can run your application.

Normally the best course of action is to use the latest SDK while using the earliest deployment target you want to support. By using the latest SDK your application can take advantage of any bug fixes Apple made.

## **When to Use an Earlier SDK?**

Last section I said there was rarely a need to change SDKs. When would you want to use an earlier SDK? I can think of three reasons why you might want to use an earlier SDK. The first reason is if you want to build a version of your application for a specific version of Mac

OS X. Suppose you had two versions of your application: one for Snow Leopard and one for Leopard. You might want to use the 10.6 SDK for the Snow Leopard version and the 10.5 SDK for the Leopard version.

The second reason is for testing purposes. Suppose you're supporting Snow Leopard, Leopard, and Tiger and you want to make sure you didn't use any Snow Leopard or Leopard-specific function calls. You would temporarily use the 10.4 SDK so you could discover any compatibility errors when you build the project. Once your code compiled without any problems with the 10.4 SDK, you could switch to the 10.6 SDK. The Base SDK build setting lets you temporarily change the SDK. You could use the 10.4 SDK for the Debug build configuration and the 10.6 SDK for the Release build configuration.

The third reason to use an older SDK is if you have some old code. A newer SDK may force you to update some of the old code to get it to build. You may not want to spend the time to update your code so you would use an older SDK.