iOS Development Guide

Tools & Languages: IDEs

2010-11-15

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures, Tables, and Listings

Appendix A **Hello, World! Source Code 87**

Introduction

To develop iOS applications, you use Xcode, Apple's first-class integrated development environment (IDE). Xcode provides all the tools you need to design your application's user interface and write the code that brings it to life. As you develop your application, you run it on your computer, an iPhone, an iPad, or an iPod touch.

This document describes the iOS application development process. It also provides information about becoming a member of the iOS Developer Program, which is required to run applications on devices for testing.

After you finish developing your iOS application, you submit it to the App Store, the secure marketplace where iOS users obtain their applications. However, you should test your application on a small set of users before publishing it to cover a wide variety of usage patterns and get feedback about your product. This document describes how to create a group of testers for your application and how to distribute it to them.

To take advantage of this document, you should be familiar with the iOS application architecture, described in *iOS Application Programming Guide*. You should also be familiar with basic programming concepts.

After reading this document, you'll have a basic understanding of the iOS application development process. To enhance that knowledge, you should read the documents listed later in this introduction.

Software requirements: This document applies to the iOS SDK 4.2 (with Xcode 3.2.5) distribution on Mac OS X v10.6.4.

If you're interested in developing iOS web applications, visit http://developer.apple.com/devcenter/safari/library.

Organization of This Document

This document contains the following chapters:

- "iOS Development Quick Start" (page 13) provides an overview of the major development tasks you follow to design, build, and run an application using Xcode.
- "Configuring Applications" (page 27) describes how to configure your application's properties and entitlements, and how to adapt it so that it builds correctly in the iOS simulation and device environments.
- "Building and Running Applications" (page 33) describes each of the steps required to run or debug your iOS applications.
- "Using iOS Simulator" (page 45) describes the ways in which you use your computer's input devices to simulate the interaction between iOS users and their devices.

- "Managing Devices and Digital Identities" (page 49) shows how to configure your computer and your device for development; how to use the Xcode Organizer to view console logs or crash information, and take screen shots of applications running on your device; and how to safeguard the digital identifications required to install applications in development on devices.
- "Debugging Applications" (page 57) describes the Xcode debugging facilities.
- "Unit Testing Applications" (page 63) introduces unit testing and describes how you can take advantage of it in your projects.
- "Tuning Applications" (page 75) describes Instruments and Shark, the tools you use to measure and tune your application's performance.
- "Distributing Applications" (page 77) describes how to create an archive of an application using a distribution provisioning profile, and how to send it to application testers or submit it to iTunes Connect. It also contains testing instructions for application testers.
- "iOS Development FAQ" (page 85) lists common questions developers ask about iOS development.
- "Hello, World! Source Code" (page 87) contains the source code for the Hello, World! application described in "Tutorial: Hello, World!" (page 19).
- "Unit-Test Result Macro Reference" (page 89) describes the test-result macros you can use in your unit-test methods.

Installing the iOS SDK

To install the tools you need to develop iOS applications, including Xcode, iOS Simulator, and others, visit http://developer.apple.com/devcenter/ios.

Note: The iOS SDK requires an Intel-based Mac.

See Also

These documents describe the essential concepts you need to know about developing iOS applications:

- *iOS Technology Overview* introduces iOS and its technologies.
- *iOS Application Programming Guide* describes the architecture of an iOS application and shows the key customization points in UIKit and other key system frameworks.
- *Cocoa Fundamentals Guide* introduces the basic concepts, terminology, architectures, and design patterns of the Cocoa frameworks and development environment.
- *The Objective-C Programming Language* introduces object-oriented programming and describes the main programming language used for iOS development.

- *Dashcode User Guide*, which describes how to create webpages optimized for Safari on iOS. These web applications make use of web technologies such as HTML, CSS, and JavaScript.

Introduction

iOS Development Quick Start

Developing iOS applications is a pleasant and rewarding endeavor. To convert your ideas into products you use Xcode, the integrated development environment (IDE) used to develop iOS applications. With Xcode you organize and edit your source files, view documentation, build your application, debug your code, and optimize your application's performance.

Note: To develop iOS applications, you must be a registered Apple developer. To run applications on a device, you must be a member of the iOS Developer Program. For more information, see "Managing Devices and Digital Identities" (page 49).

This chapter provides an overview of the major development tasks you follow to design, build, and run an application using Xcode. It also includes a quick tutorial that shows how to develop the ubiquitous Hello, World! application for iOS.

Essential Development Tasks

The iOS-application development process is divided into these major steps:

1. Create your project.

   Xcode provides several project templates that get you started. You choose the template that implements the type of application you want to develop. See "Creating an iOS Application Project" (page 14) for details.

2. Design the user interface.

   The **Interface Builder application** lets you design your application's user interface graphically and save those designs as resource files that your application loads at runtime. If you do not want to use Interface Builder, you can layout your user interface programmatically. See "User Interface Design Considerations" in *iOS Application Programming Guide* for more information.

3. Write code.

   Xcode provides several features that help you write code fast, including class and data modeling, code completion, direct access to documentation, and refactoring. See "Editing Code" (page 15) for details.

4. Build and run your application.

   You build your application on your computer and run it in the iOS simulation environment or on your device. See "Building and Running Your Application" (page 18) for more information.

5. Measure and tune application performance.

After you have a running application, you should measure its performance to ensure that it uses a device's resources as efficiently as possible and that it provides adequate responses to the user's gestures. See "Measuring Application Performance" (page 19) for more information.

The rest of this section gives more details about these steps.

Creating an iOS Application Project

The iOS SDK provides several project templates to get you up and running developing your application. You can choose from these types of application:

- **Navigation-based Application.** An application that presents data hierarchically, using multiple screens. The Contacts application is an example of a navigation-based application.
- **OpenGL ES Application.** An application that uses an OpenGL ES–based view to present images or animation.
- **Split View–based Application.** An iPad application that displays more than one view onscreen at a time to, for example, present data in a master-detail or source list–style arrangement. The iPad Mail application is an example of a split-view–based application.
- **Tab Bar Application.** An application that presents a radio interface that lets the user choose from several screens. The Clock application is an example of a tab bar application.
- **Utility Application.** An application that implements a main view and lets the user access a flip-side view to perform simple customizations. The Stocks application is an example of a utility application.
- **View-based Application.** An application that uses a single view to implement its user interface.
- **Window-based Application.** This template serves as a starting point for any application, containing an application delegate and a window. Use this template when you want to implement your own view hierarchy.

If you need to develop a static library for use in an iOS application, you can add a static library target to your project by choosing Project > New Target and selecting the Static Library target template in the iOS/Cocoa Touch list.

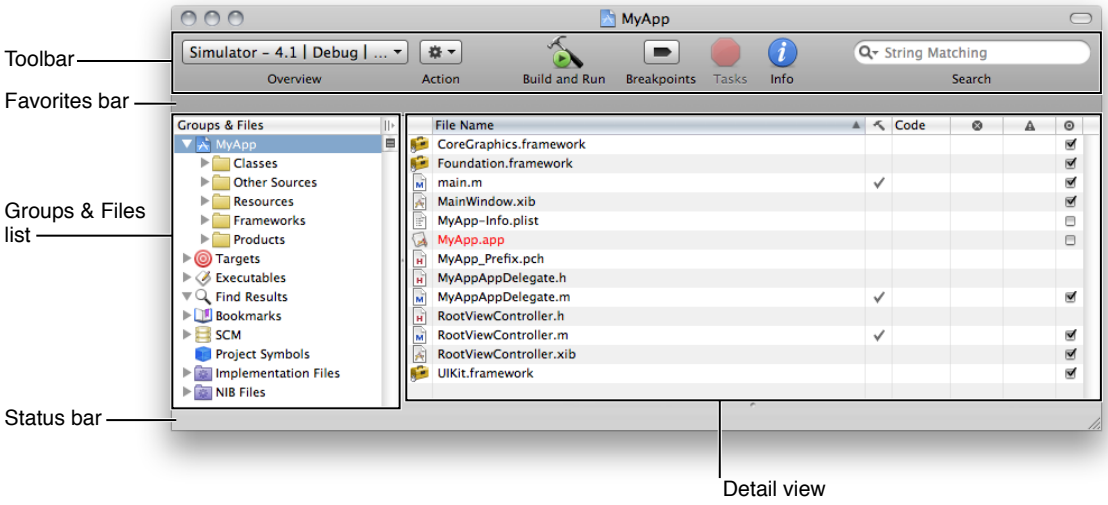
Static libraries used in iOS applications do not need to be code signed. Therefore, you should remove the Code Signing Identity build setting definition from the static library targets you create. To do so:

1. Open the static library target's Info window and display the Build pane.
2. In the Code Signing group, select the Any iOS conditional definition for the Code Signing Identity build setting.
3. Change the conditional definition's value from iPhone Developer to Don't Code Sign.

To learn more about the iOS application architecture, see *iOS Application Programming Guide*.

To develop an iOS application, you work on an Xcode project. And you do most of your work on projects through the **project window**, which displays and organizes your source files and other resources needed to build your application. This window allows you to access and edit all the pieces of your project. Figure 1-1 shows the project window.

Figure 1-1 Project window



The project window contains the following key areas for navigating your project:

- **Groups & Files list.** Provides an outline view of your project's contents. You can move files and folders around and organize your project contents in this list. The current selection in the Groups & Files list controls the contents displayed in the detail view.
- **Detail view.** Shows the item or items selected in the Groups & Files list. You can browse your project's contents in the detail view, search them using the search field, or sort them according to column. The detail view helps you rapidly find and access your project's contents.
- **Toolbar.** Provides quick access to the most common Xcode commands.
- **Favorites bar.** Lets you store and quickly return to commonly accessed locations in your project. The favorites bar is not displayed by default. To display the favorites bar, choose View > Layout > Show Favorites Bar.
- **Status bar.** Displays status messages for the project. During an operation—such as building or indexing—Xcode displays a progress indicator in the status bar to show the progress of the current task.

To learn more about creating projects, see "Creating Projects".

Editing Code

The main tool you use to write your code is the Xcode text editor. This advanced text editor provides several convenient features:

- **Header file lookup.** By Command–double-clicking a symbol, you can view the header file that declares the symbol.
- **API reference lookup.** By Option–double-clicking a symbol, you get access to API reference that provides information about the symbol's usage.
- **Code completion.** As you type code, you can have the editor help out by inserting text for you that completes the name of the symbol Xcode thinks you're going to enter. Xcode does this in an unobtrusive and overridable manner.
- **Code folding.** With code folding, you can collapse code that you're not working on and display only the code that requires your attention.

For details about these and other text editor features, see "The Text Editor".

Using Code Completion

The text editor helps you type code faster with code completion. When code completion is active, Xcode uses both text you have typed and the context into which you have typed it to provide suggestions for completing the token it thinks you intend to type. Code completion is not active by default.

To activate code completion:

1. Open the Xcode Preferences window.

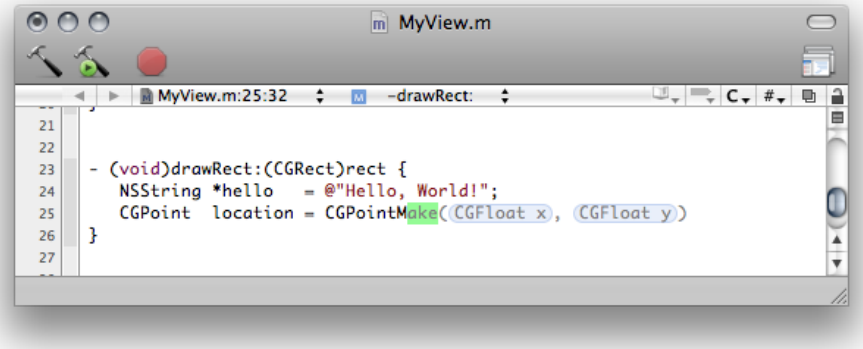
   Choose Xcode > Preferences.

2. In the Code Completion section of the Code Sense pane, choose Immediate from the Automatically Suggest pop-up menu.

3. Click OK.

As you type the name of a symbol, Xcode recognizes that symbol and offers a suggestion, as shown in Figure 1-2. You can accept suggestions by pressing Tab or Return. You may also display a list of completions by pressing Escape.

Figure 1-2 Using code completion



To learn more about code completion, see “Completing Code” in *Xcode Workspace Guide*.

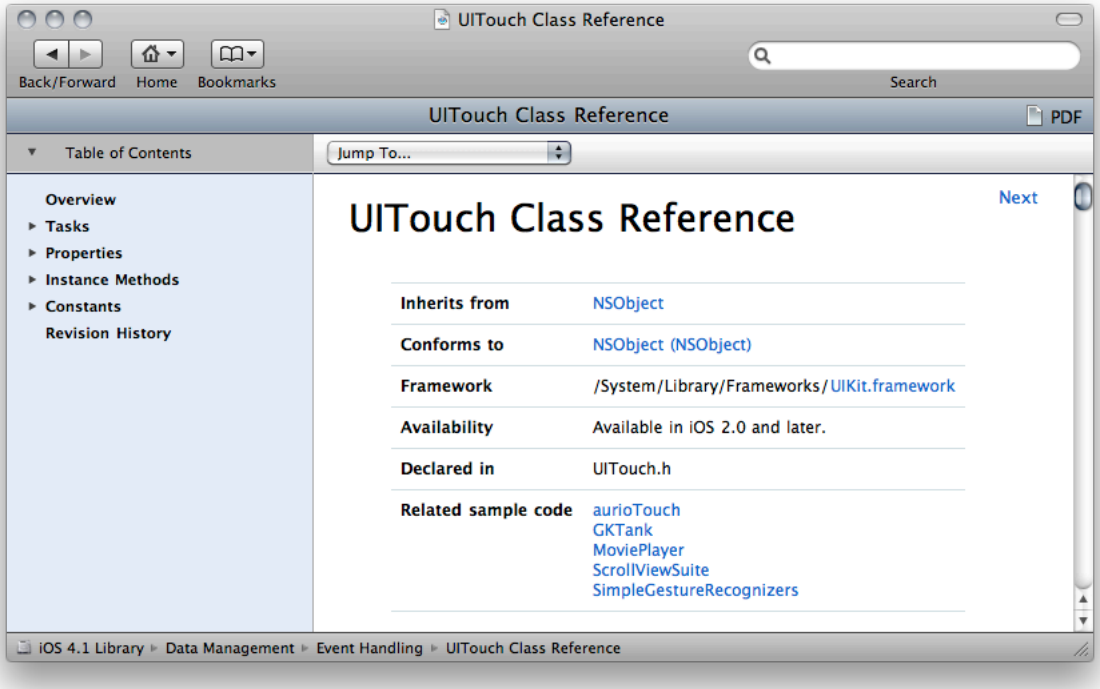
Accessing Documentation

During development, you may need fast access to reference for a particular symbol or high-level documentation about API usage or an iOS technology. Xcode gives you easy access to such resources through the Quick Help and Documentation windows.

Quick Help is a lightweight window, shown in Figure 1-4 (page 18), that provides a condensed view of the API reference for the selected item, without taking your focus away from the editor in which the item is located. This window provides an unobtrusive way to consult API reference. However, when you need to dig deeper into the reference, the Documentation window is just a click away.

The **Documentation window** (Figure 1-3) lets you browse and search the developer documentation (which includes API reference, guides, and articles about particular tools or technologies) installed on your computer. It provides access to a wider and more detailed view of the documentation than the Quick Help window, for the times when you need additional details.

Figure 1-3 Viewing API reference in the Documentation window



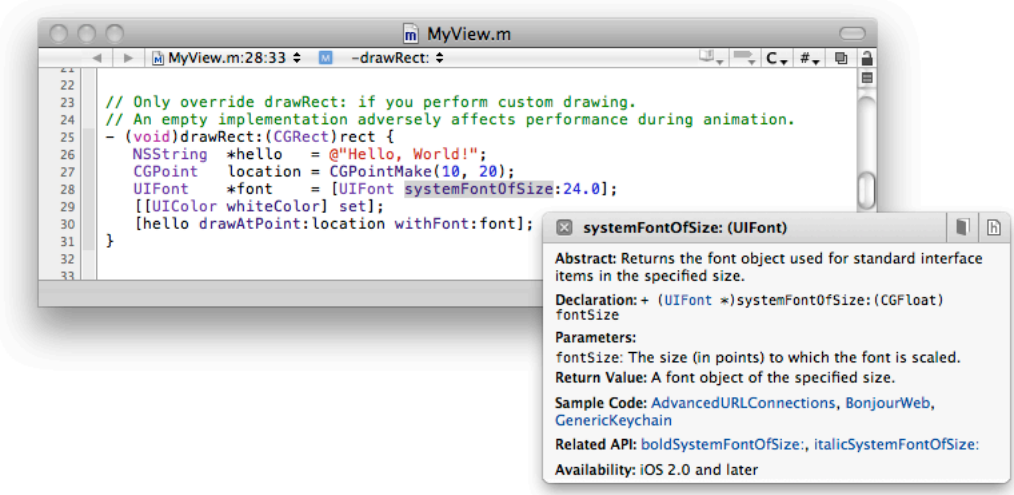
To display the API reference for a symbol in a source file, select the symbol in the text editor and choose Help > Find Selected Text in API Reference (you can also Option–double-click the symbol name). This command searches for the selected symbol in the API reference for your project’s SDK and displays it in the Documentation window. For example, if you select the `UIFont` class name in a source file and execute the Find Selected Text in API Reference command, Xcode opens the Documentation window and displays the API reference for the `UIFont` class.

While the Documentation window is a great tool to browse the iOS library, sometimes you may not want to take your focus away from the text editor while you write code, but need basic information about a symbol in a condensed way. The Quick Help window provides such information in a small and unobtrusive window.

The Quick Help window actively follows you as you move the cursor around a source file. When it recognizes a symbol for which it finds API reference, the Quick Help window displays that reference, as shown in Figure 1-4. All you have to do is glance at the Quick Help window to get essential details about the symbol.

To display the Quick Help window, choose Help > Quick Help.

Figure 1-4 Viewing API reference in the Quick Help window



From the Quick Help window you can quickly jump to more comprehensive reference for the symbol, or even view the header that declares it.

For more information about accessing documentation in Xcode, see Documentation Access.

Building and Running Your Application

iOS Simulator implements the iOS API, providing an environment that closely resembles the environment devices provide. It allows you to run your applications in Mac OS X, letting you quickly test application functionality when you don't have a device available. However, running applications in iOS Simulator is not the same as running them in actual devices. iOS Simulator does not emulate device performance: It doesn't implement the memory constraints or processor performance of an actual device. First, the simulator uses Mac OS X versions of the low-level system frameworks instead of the versions that run on the devices. Secondly, there may be hardware-based functionality that's unavailable on the simulator. But, in general, the simulator is a great tool to perform initial testing of your applications.

To get an accurate idea of how your application performs on a user's device, you must run the application on a device and gather performance data using Instruments and other performance-measuring tools.

To compile and debug your code, Xcode relies on open-source tools, such as GCC, LLVM-GCC, and GDB. Xcode also supports team-based development with source control systems, such as Subversion, CVS, and Perforce.

Building your application involves the following steps:

- Compiling your source files and generating your application binary.
- Placing the binary in iOS Simulator or on your device.

Xcode performs these tasks for you when you execute the Build command. See "Building and Running Applications" (page 33) for details.

Measuring Application Performance

After you have tested your application's functionality, you must ensure that it performs well on a device. This means that the application uses the device's resources as efficiently as possible. For example, memory is a scarce resource; therefore, your application should maintain a small memory footprint so that it doesn't impair the performance of iOS. Your application should also use efficient algorithms to consume as little power as possible not to reduce battery life. Xcode provides two major tools to measure and tune application performance: Instruments and Shark.

The **Instruments application** is a dynamic performance analysis tool that lets you peer into your code as it's running and gather important metrics about what it is doing. You can view and analyze the data Instruments collects in real time, or you can save that data and analyze it later. You can collect data about your application's use of the CPU, memory, the file system, and the network, among other resources.

The **Shark application** is another tool that helps you find performance bottlenecks in your code. It produces profiles of hardware and software performance events, and shows how your code works as a whole and how it interacts with iOS.

See "Tuning Applications" (page 75) for more information.

Further Exploration

To learn more about the Xcode development process, see *A Tour of Xcode*.

Tutorial: Hello, World!

This tutorial guides you through the creation of a simple iPhone application that prints text on the screen.

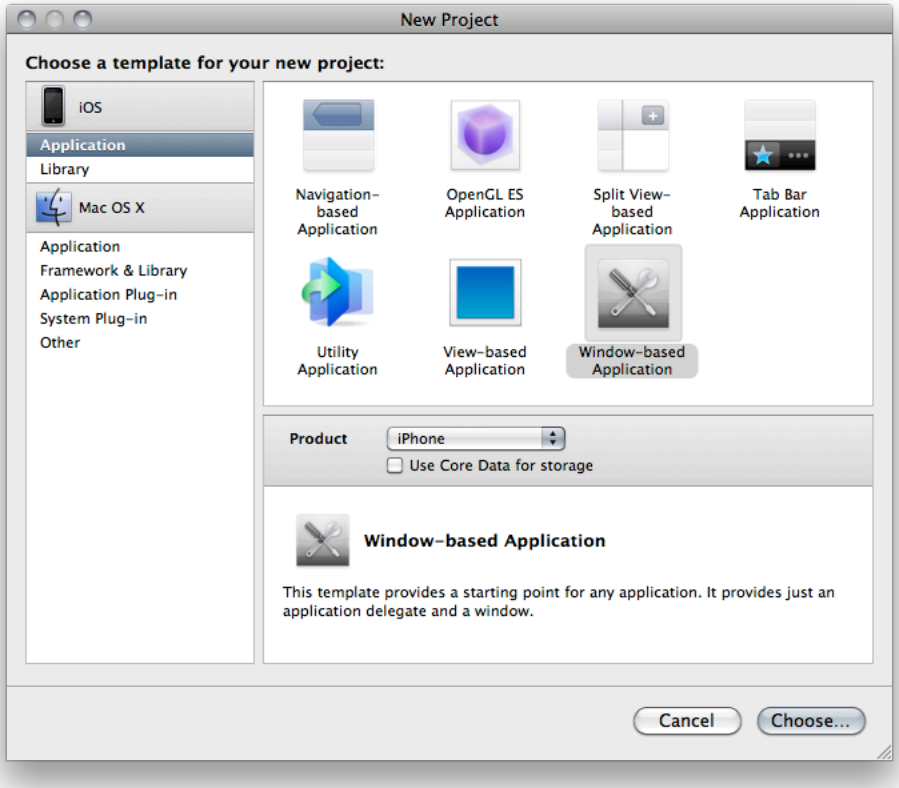
Create the Project

To create the Hello World project, follow these steps:

1. Launch the Xcode application, located in `<Xcode>/Applications`.

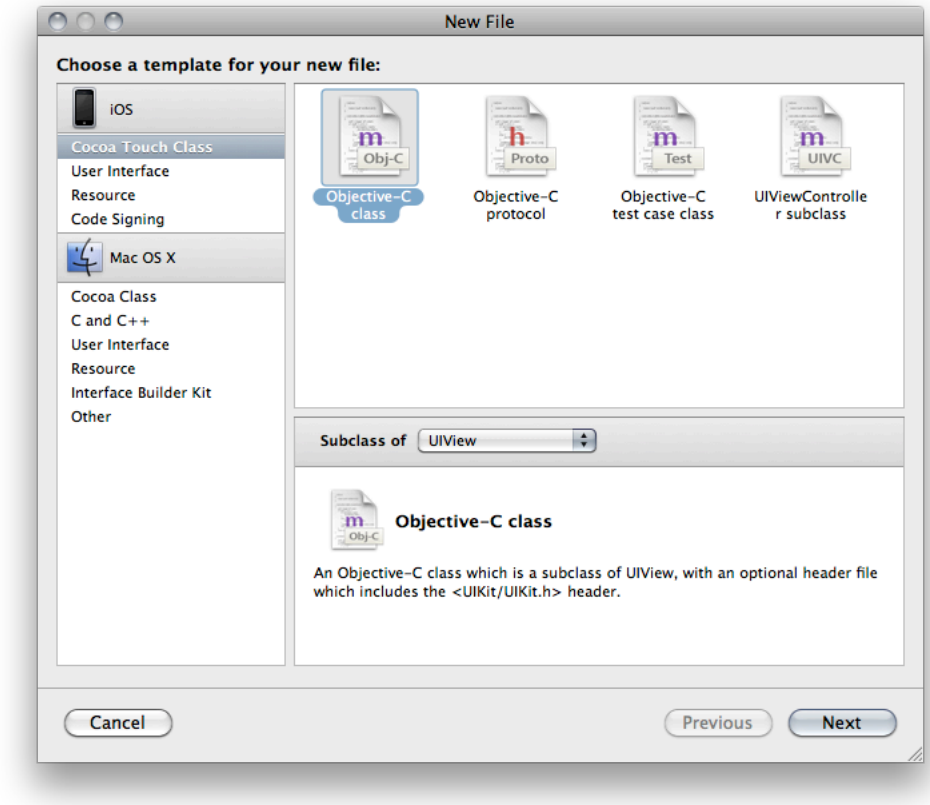
   `<Xcode>` represents the directory in which you installed the Xcode toolset. See "Xcode Installation Details" for more information.

2. Choose File > New Project.
3. Select the iOS/Application/Window-Based Application template, choose iPhone from the Product pop-up menu, and click Choose.



4. Name the project `HelloWorld` and choose a location for it in your file system.
5. Add the `MyView` class to the project.
    a. Choose File > New File.

b. Select the Cocoa Touch UIView subclass template and click Next.



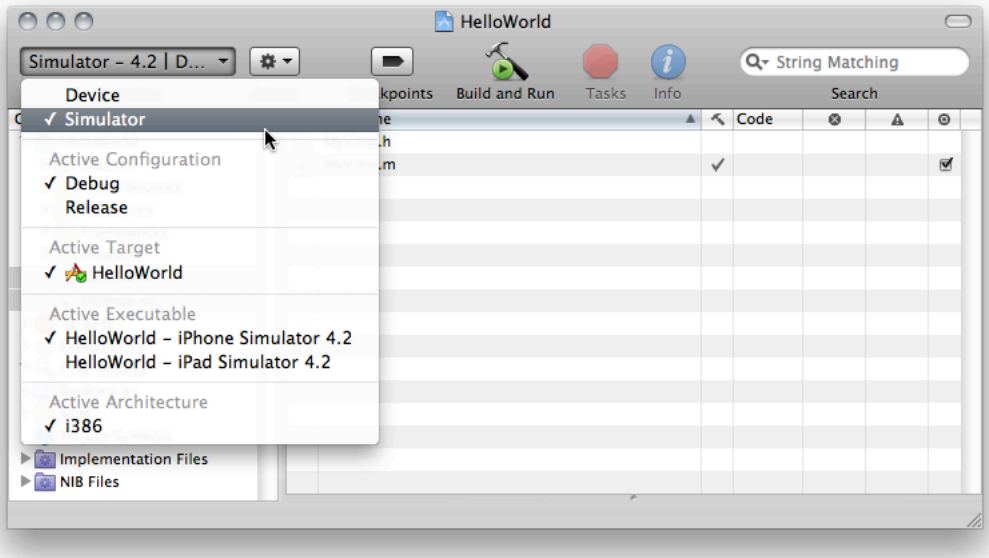
c. In the File Name text field, enter `MyView.m`.

d. Select the “Also create "MyView.h"” option and click Finish.

6. Choose the destination for the application.

The product destination specifies the type of environment for which Xcode builds your application and where to install it. If you have a development device plugged in at the time you create the project, Xcode sets the product destination to Device. Otherwise, it sets the destination to Simulator.

To set the product destination, choose between Device and Simulator from the Overview toolbar menu in the project window. If you have more than one device connected, specify the device onto which to install your application by choosing it from the Active Executable section of the Overview toolbar menu.



Write the Code

The Xcode text editor is where you spend most of your time. You can write code, build your application, and debug your code. Let's see how Xcode assists you in writing code.

To experience the Xcode source code editing features, you should perform the following instructions to enter the application's source code. For your convenience, "Hello, World! Source Code" (page 87) includes the final source code.

First, modify the `HelloWorldAppDelegate` class to use the `MyView` class:

1. In the Groups & Files list, select the HelloWorld project.
2. In the detail view, double-click `HelloWorldAppDelegate.m`.
3. In the `HelloWorldAppDelegate` editor window:
    a. Add the following code line below the existing `#import` line.

    ```
    #import "MyView.h"
    ```

    b. Add the following code lines to the `application:didFinishLaunchingWithOptions:` method, below the override-point comment.

    ```
    MyView *view = [[MyView alloc] initWithFrame:[window frame]];
    [window addSubview:view];
    [view release];
    ```

After making these changes, the code in the `HelloWorldAppDelegate.m` file should look similar to this:

```
#import "HelloWorldAppDelegate.h"
#import "MyView.h"

@implementation HelloWorldAppDelegate

@synthesize window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // Override point for customization after application launch
    MyView *view = [[MyView alloc] initWithFrame:[window frame]];
    [window addSubview:view];
    [view release];

    [window makeKeyAndVisible];
    return YES;
}

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

@end
```

Listing 1-1 shows the code that draws "Hello, World!" in the window. Add the code in the listing the implementation section of the `MyView.m` file.

Listing 1-1 Method to draw "Hello, World!" in a view

```
- (void)drawRect:(CGRect) rect {
    NSString *hello   = @"Hello, World!";
    CGPoint  location = CGPointMake(10, 20);
    UIFont   *font    = [UIFont systemFontOfSize:24.0];
    [[UIColor whiteColor] set];
    [hello drawAtPoint:location withFont:font];
}
```

If you turned on code completion (as described in "Using Code Completion" (page 16)), as you type symbol names the text editor suggests completions for the symbol names it recognizes. For example, as you type `CGPointM`, the text editor suggests the completion shown in Figure 1-2 (page 16). You can take advantage of code completion here by accepting the suggested completion and jumping to the parameter placeholders:

1. Jump to the first parameter by choosing Edit > Select Next Placeholder, and type `10`.

   The Select Next Placeholder command moves you among the arguments in function or method calls that the text editor suggests as completions to the text you're typing.

2. Jump to the second parameter and type `20`.

3. Enter the semicolon (`;`) at the end of the line and press Return.

Run the Application

To build and run the Hello World application, choose Build > Build and Run (or click the Build and Run toolbar item in the project window). If there are no build errors, Xcode installs the application in iOS Simulator or your device (depending on the product-destination setting).

Troubleshooting Hello, World! Build Errors

This section contains possible build errors for the Hello, World! project and their cause.

```
Building ... — 2 errors
   Compiling <project_directory>/Classes/HelloWorldAppDelegate.m (2 errors)
      error: 'MyView' undeclared (first use in this function)
      error: 'view' undeclared (first use in this function)
```

Fix this build error by adding the line

```
#import "MyView.h"
```

to the `HelloWorldAppDelegate.m` file.

To learn about other possible build errors, see "Solving Build Errors" (page 40).

Further Exploration

Now that you learned how to write the standard Hello, World! application for iOS, you can experiment with *HelloWorld*, the Cocoa Touch version of this ubiquitous application.

For a step-by-step tutorial in developing a more complex application, see *Your First iOS Application*.

To learn more about Objective-C, see *Learning Objective-C: A Primer*.

To learn more about developing iOS applications, see *iOS Application Programming Guide*.

Configuring Applications

This chapter describes how to set up your application's properties to customize its runtime environment, configure its entitlements to take advantage of iOS security features, and how to adapt its build process for different SDKs and architectures.

This chapter also shows how to upgrade a target that builds an iPhone application into either a target that builds an application optimized for iPhone and iPad, or two targets, one to build an iPhone application and one to build an iPad application.

To learn how to set up a device for development, see "Managing Devices and Digital Identities" (page 49).

Editing Property-List Files

Property-list files are XML files that organize data into named values and lists of values using simple data types. These data types let you create, transport, and store structured data in an accessible and efficient way. Xcode uses two main types of property-list file to store runtime-configuration information for your application:

- **Information-property list.** These files, commonly referred to as info-plist files, contain essential information used by your application and iOS. See "The Information Property List" in *iOS Application Programming Guide* for information about the application properties defined in info-plist files.
- **Entitlements.** These files define properties that provide your application access to iOS features (such as push notifications) and secure data (such as the user's keychain).

To learn more about property-list files, see *Property List Programming Guide*.

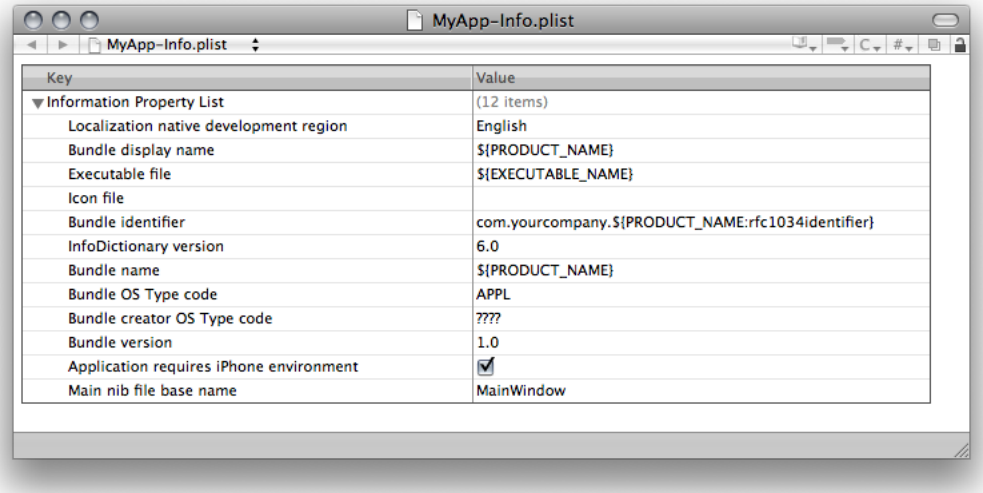
To edit a property-list file, perform one of these actions:

- To edit it in the editor pane of the Project window, select the file in the Groups & Files list or in the detail view.
- To edit it in a dedicated editor window, double-click the file in the Groups & Files list or in the detail view.

For more information about the Groups & Files list, see "Project Window Components" in *Xcode Workspace Guide*.

Figure 2-1 shows a property-list editor window with an info-plist file.

Figure 2-1 Property-list editor window with an info-plist file

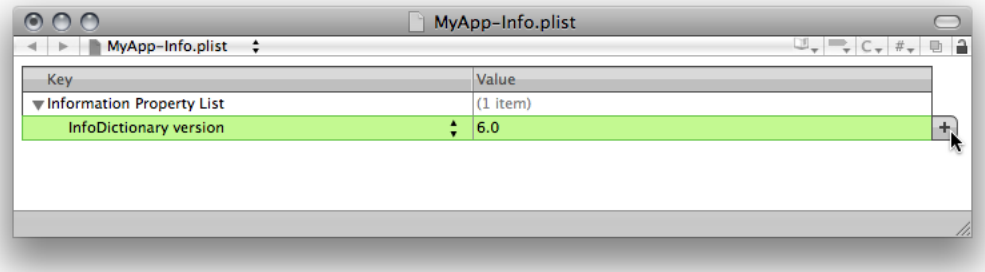


Each row in the file specifies a property definition (or key/value pair). Key cells specify property names (and their data types). Value cells specify property values.

These are the property-list editing actions you can perform with the property-list editor:

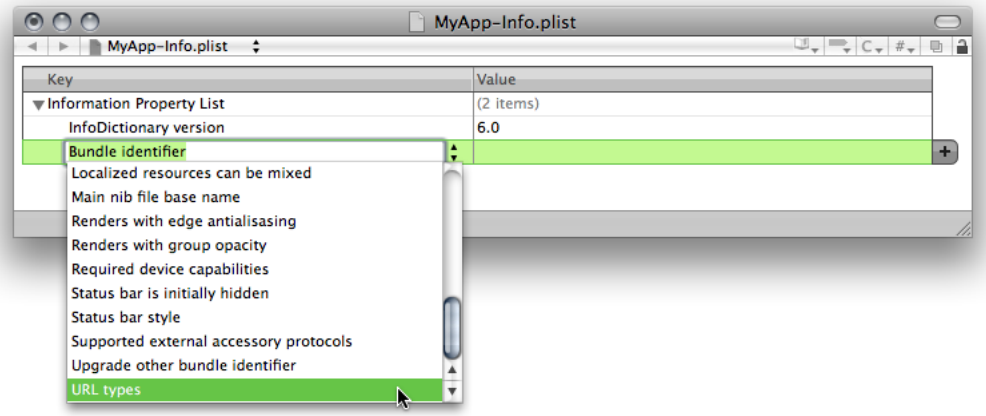
- **Add a sibling property.** With a row selected and its disclosure triangle closed, click the Add Sibling button (shown in Figure 2-2) or press Return to add a sibling to the property.

  Figure 2-2 Adding a sibling property in the property-list editor

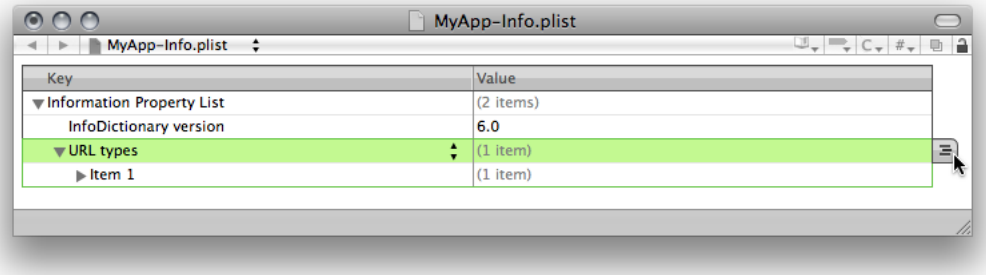
  After adding the sibling, choose the property type from the property-type menu in the key cell, as shown in Figure 2-3.

Figure 2-3 Specifying a property's type in the property-list editor



- **Add a child property.** With a row with a multi-value property—a property whose value is a list of values—selected and its disclosure triangle open, click the Add Child button (shown in Figure 2-4) or press Return to add a child to the property.

Figure 2-4 Adding a child property in the property-list editor



- **Delete property.** With a row selected, press Delete to delete the row.

To make editing property-list files convenient and to ensure the file's structure is correct, the property-list editor uses property-list–file schemas to display property names in the Key column, and formatted values (such as check boxes for Boolean values) in the Value column. However, you may want to see the properties' key names and values in their XML (or "raw") form. To toggle between viewing the formatted keys and values and their raw form, toggle the Show Raw Keys/Values option of the property-list editor shortcut menu.

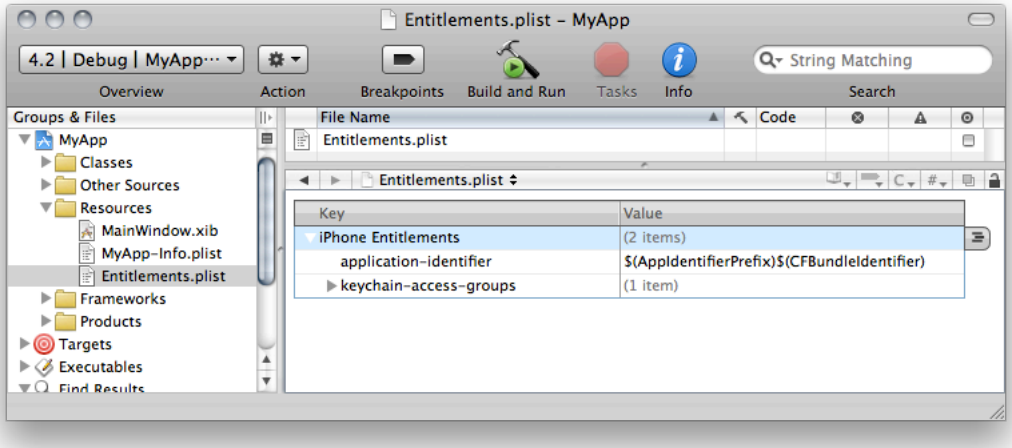
Managing Application Entitlements

iOS provides access to special resources and capabilities—such as whether your application can be debugged—through properties called **entitlements**. To specify entitlement information in your application, you add an entitlement property-list file containing entitlement definitions (key/value pairs) to your project. When you build your application, Xcode copies the file to the generated application bundle.

To add an entitlements property-list file to your project:

1. In the Groups & Files list, select the Resources group.
2. Choose File > New File.
3. Choose the iOS/Code Signing/Entitlements template.
4. Name the file `Entitlements.plist`. (You can use any name; just ensure it matches the value of the Code Signing Entitlements build setting, as explained later in this section.)
5. Click Finish.
6. Set the type of the property-list file to iPhone Entitlements.

   With the file selected in the text editor, choose View > Property List Type > iPhone Entitlements plist.



7. Add your entitlement entries to the file.

   For each entitlement property you need to define:

   a. Click the Add Child or Add Sibling button to the right of the selected row. The Add Child button has three lines depicting a hierarchy, the Add Sibling button has a plus (+) sign on it.

   b. Choose the entitlement property from the pop-up menu that appears.

      If the entitlement you need to add doesn't appear in the menu, choose View > Property List Type > Default for File Type. Then enter the entitlement-key name and type.

c. Enter the value for the property.

Conditional Compilation and Linking

The two iOS runtime environments are the simulation environment and the device environment. You use the former to test your application on your Mac, and the latter to test it on a device. These environments are fundamentally different; therefore, when using technology that's implemented differently in the two environments you need to tweak your code so that some of it runs in iOS Simulator application but not on a device.

This section shows how to target code to iOS Simulator or a device and which frameworks (or libraries) to link, depending on whether the active SDK belongs to the iOS Simulator SDK family or the iOS Device SDK family.

Compiling Source Code Conditionally for iOS Applications

There may be times when you need to run code on the simulator but not on a device, and the other way around. On those occasions, you can use the preprocessor macros `TARGET_OS_IPHONE` and `TARGET_IPHONE_SIMULATOR` to conditionally compile code.

Listing 2-1 shows how to use the `TARGET_IPHONE_SIMULATOR` macro to determine whether code meant for iOS is being compiled for the simulator or devices.

Listing 2-1 Determining whether you're compiling for the simulator

```
// Set hello to "Hello, <device or simulator>"!
#if TARGET_IPHONE_SIMULATOR
   NSString *hello = @"Hello, iOS Simulator!";
#else
   NSString *hello = @"Hello, iOS device!";
#endif
```

Listing 2-2 shows how to use the `TARGET_OS_IPHONE` macro in a source file to be shared between Mac OS X and iOS.

Listing 2-2 Determining whether you're compiling for iOS

```
#if TARGET_OS_IPHONE
   #import <UIKit/UIKit.h>
#else
   #import <Cocoa/Cocoa.h>
#endif
```

The `TARGET_OS_IPHONE` and `TARGET_IPHONE_SIMULATOR` macros are defined in the `TargetConditionals.h` header file.

Linking Frameworks Conditionally for iOS Applications

There may be occasions when you need to configure your application target so that it links against one framework to run on the simulator and a different framework to run on a device.

To link a framework only when using a particular SDK, set the Other Linker Flags build setting in all configurations for the SDK you want the definition to apply to `-framework <framework_name>`.

If you need to, you can add another condition to the Other Linker Flags build setting to specify a different SDK and framework.

See "Editing Conditional Build Settings" in *Xcode Project Management Guide* for details about defining build settings for particular SDKs.

Upgrading a Target from iPhone to iPad

If you have an iPhone application that you want to upgrade to run on iPad devices you need to upgrade the target that builds your iPhone application into a target that can build both an iPhone and an iPad application, or add a target to your project for building the iPad application.

To upgrade an iPhone target for iPad development, select the target in the Groups & Files list and choose Project > Upgrade Current Target for iPad.

Building and Running Applications

When you're ready to run or debug your application, you build it using the Xcode build system. If there are no build errors, you can run it in iOS Simulator or on a device.

Note: After testing your application in iOS Simulator, you must test it on an iOS-based device to measure and tune its performance. To be able to run your application on a device, you must be a member of the iOS Developer Program; see "Becoming a Member of the iOS Developer Program" (page 49) for details.

The iOS SDK comprises two SDK families: The iOS Simulator SDK and the iOS Device SDK.

- **iOS Simulator SDK:** These SDKs build applications that run in iOS Simulator.
- **iOS Device SDK:** These SDKs build applications that run in a device.

These are the steps you follow to build and run applications:

1. Specify the build-time environment.
2. Specify the runtime environment.
3. Specify the application's destination (where to run it: the simulator or a device).
4. Build the application.
5. Run the application.

This chapter describes each of the steps required to run or debug your application. Start with "Running Sample Applications" (page 33) if you're interested in seeing applications that showcase iOS features.

Running Sample Applications

The iOS Dev Center provides several resources that help you learn about the iOS application development process. One of these resource types is sample code. You can download sample code to your computer and run it in iOS Simulator. If you're an iOS Developer Program member, you can also run sample code on your device. See "Becoming a Member of the iOS Developer Program" (page 49) for details.

To run sample code:

1. Download the ZIP archive (the file with the `.zip` suffix) containing the sample code you want to use. The archive is named after the application name; for example, `HelloWorld.zip`.
2. If you download the sample-code projects from the Xcode Documentation window, Xcode expands the archive and opens the project automatically. Otherwise, continue to the next step.

3. Expand the archive by double-clicking it.
4. Navigate to the sample-code project directory, also named after the example application; for example, `HelloWorld`.
5. Double-click the project package, a file with the `.xcodeproj` suffix. For example, `HelloWorld.xcodeproj`. This action opens the project in Xcode.

   You can also drag the project package to the Xcode icon in the Dock to open the project.

> **Troubleshooting: Xcode doesn't launch:** If Xcode doesn't launch, you need to download it and install it on your computer. To download Xcode, visit iOS Dev Center.

With the sample-code project open in Xcode, follow the instructions in the following sections to build and run the application.

The Build-and-Run Workflow

This section describes each of the steps in the build-and-run workflow.

Specifying the Build-time Environment

When you build your application, Xcode uses a build environment made up of frameworks, libraries, applications, command-line tools, and other resources. Each release of the iOS SDK makes improvements to this environment to, for example, add user-interface features or improve compilers and resource-processing tools. In addition to these resources, you can specify whether you want to build your application to debug it or to distribute it to customers. This section describes how to set your build-time environment.

Setting the Base SDK

One of the main factors that determine how Xcode builds your application is the SDK used to build it.

You specify the SDK Xcode uses to build your application with the Base SDK build setting, which you can set in the General pane of the Project Info window or in the Build pane of the Project or Target Info window.

> **Note:** To ensure minimal reconfiguration of your projects as you adopt new SDK releases, instead of a specific SDK release, set the base SDK for your projects to Latest iOS. This way your project always uses the latest available SDK in the toolset.

Base SDK Missing

If your project has its Base SDK setting set to a particular iOS SDK release, when you open that project with a later iOS SDK distribution in which that SDK release is not available, the Base SDK setting has no valid value. In this case, the Overview toolbar menu displays the message "Base SDK Missing."

To fix this:

1. Set the base SDK for the project to an available SDK release or to Latest iOS.
2. Ensure that the target doesn't set a different value for the Base SDK build setting.
3. Close and reopen the project

Setting Your Code Signing Identity

When you build your application to run it on a device, Xcode signs it with a development certificate (also known as a code signing identity) stored on your keychain. To learn how to obtain and install development certificates, see "Preparing Your Mac for iOS Development" (page 49).

The Code Signing Identity build setting specifies the code signing identity Xcode uses to sign your binary. Xcode looks for code signing identities in your default keychain. Figure 3-1 shows an example set of options for the Code Signing Identity build setting obtained from the login (default) keychain shown in Figure 3-2 (the name of the default keychain appears bold in the Keychains list).

Figure 3-1 Code Signing Identity build setting options

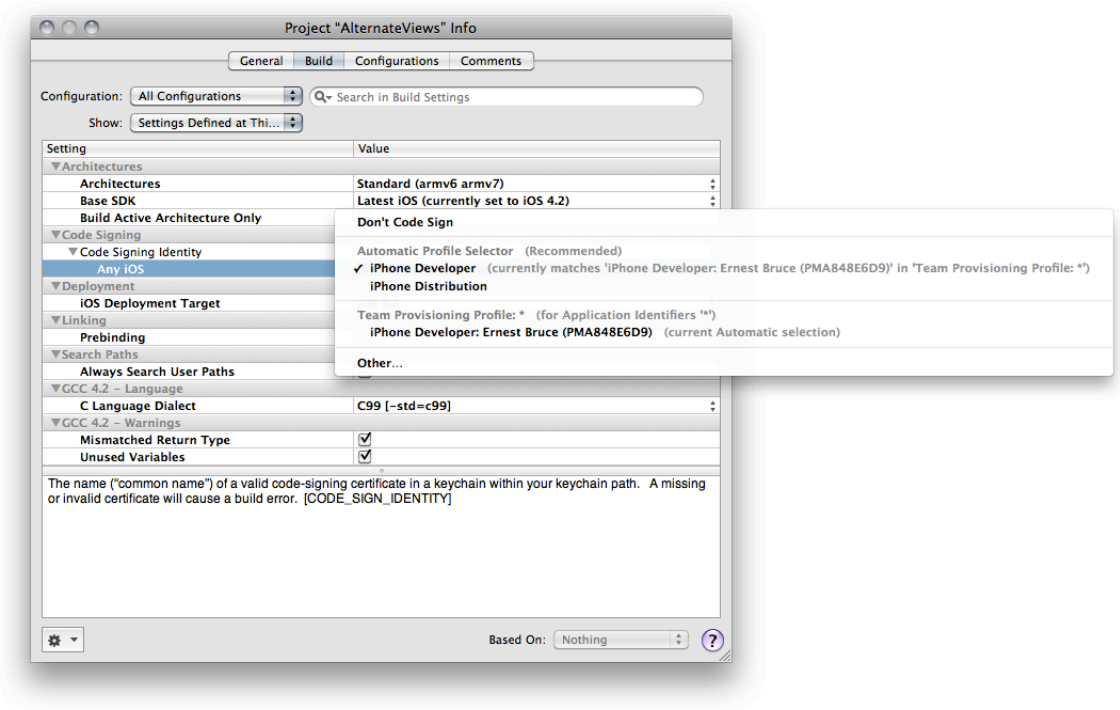


Figure 3-2 A keychain with a developer certificate

These are the possible values for the Code Signing Identity build setting:

- **Don't Code Sign.** Choose this option if you don't want to sign your binary. Note that with this value, when you build your applications with an iOS device as the destination, the build fails.
- **Automatic Profile Selectors.** These choices look for an identity whose name starts with "iPhone Developer" or "iPhone Distribution."
- **A code signing identity.** A specific code signing identity. The code signing identities in your default keychain are listed under the name of the provisioning profile under which you downloaded them from the iOS Provisioning Portal. Expired or otherwise invalid identities are dimmed and cannot be chosen.

Xcode project templates are configured to use the iPhone Developer automatic selector.

Important: If need to use different code signing identities with the same name, you must use a separate Mac OS X user account for each identity.

Setting the Architecture

An iOS device uses one of a set of architectures, which include `armv6` and `armv7`. The Architectures build setting identifies the architectures for which your application is built. You have two options for specifying the value of this setting:

- **Standard.** Produces an application binary with a common architecture, compatible with all supported iOS devices. This option generates the smallest application, but it may not be optimized to run at the best possible speed for all devices.
- **Optimized.** Produces an application binary optimized for each supported iOS device. However, the build time is longer than when using the Standard option, and the application is also larger because multiple instruction sets are bundled into it.

Choose the value for the Architecture build setting in the Build pane of the Target or Project Info window.

If you need to build your application so that it contains executable code for a different set of architectures than these predefined values offer, you can choose Other from the Architecture build-setting value list and enter the desired iOS-device architecture names.

Setting the Active Build Configuration

A **build configuration** tells Xcode the purpose of the built product. Xcode project templates are configured with Debug and Release configurations, which let you build your application for debugging or for release to customers. Debug builds include information and features that aid in debugging your application. Release builds produce smaller and faster binaries. During early and midlevel development, you should use the Debug configuration because it provides the best debugging experience. You should use the Release configuration in the last stages of development to measure and analyze your application's performance.

When you start a build task, Xcode uses the **active build configuration** to build your application. There are two places you can set the active build configuration:

- In the Set Active Build Configuration submenu in the Project menu
- In the Overview pop-up menu in the toolbar

To learn more about the Release and Debug configurations, see "Building Products".

Setting the Device Family

The **device family** identifies the type of devices you want the application to run on. There are two device families: iPhone, and iPad. The iPhone device family includes iPhone and iPod touch devices. The iPad device family includes iPad devices.

To specify the device families on which you want your application to be able to run:

1. Open the Project Info or Target Info window.
2. In the Build pane, locate the Targeted Device Family build setting and choose an appropriate value for it.

Table 3-1 lists the values you can specify for the Targeted Device Family build setting. It also indicates the device family the built application is optimized for.

Table 3-1 Values for the Targeted Device Family build setting

Value	Application is optimized for
iPhone	iPhone, and iPod touch
iPad	iPad
iPhone/iPad	iPhone, iPod touch, and iPad (universal application)

Specifying the Runtime Environment

Each release of iOS (and its corresponding SDK) includes features and capabilities not present in earlier releases. As new releases of iOS are published, some users may upgrade immediately while other users may wait before moving to the latest release. You can take one of two strategies, depending on the needs of your application and your users:

- **Target the latest iOS release.** Targeting the latest release allows you to take advantage of all the features available in the latest version of iOS. However, this approach may offer a smaller set of users capable of installing your application on their devices because your application cannot run on iOS releases that are earlier than the target release.
- **Target an earlier iOS release.** Targeting an earlier release lets you publish your application to a larger set of users (because your application runs on the target OS release and later releases), but may limit the iOS features your application can use.

You specify the earliest iOS release on which you want your application to run with the iOS Deployment Target build setting. By default, this build setting is set to the iOS release that corresponds to the Base SDK build-setting value. For example, when you set Base SDK to iOS 4.2, the value of the iOS Deployment Target build setting is iOS 4.2.

To build your application using the iOS 4.2 SDK and to allow it to run on iOS 4.0, set Base SDK to iOS 4.2 and iOS Deployment Target to iOS 4.0.

When you build your application, your deployment target selection is reflected in the `MinimumOSVersion` entry in the application's `Info.plist` file. When you publish your application to the App Store, the store indicates the iOS release on which your application can run based on the value of this property.

Note: If the SDK you're using to build the application is more recent than the application's target iOS release (for example, the active SDK is iOS 4.2 and iOS Deployment Target is iPhone OS 4.0), Xcode displays build warnings when it detects that your application is using a feature that's not available in the target OS release. See "Specifying the Runtime Environment" (page 38) for more information.

You must also ensure that the symbols you use are available in the application's runtime environment using SDK-compatibility development techniques. These techniques are described in *SDK Compatibility Guide*.

Specifying Where to Place Your Application

During development you may want to switch from running your application in iOS Simulator to running it on a device to, for example, test the device performance of your application. To switch between running your application on a device or in the simulator use the Overview toolbar menu in the Project-window toolbar, shown in Figure 3-3 (page 39), as described in these steps:

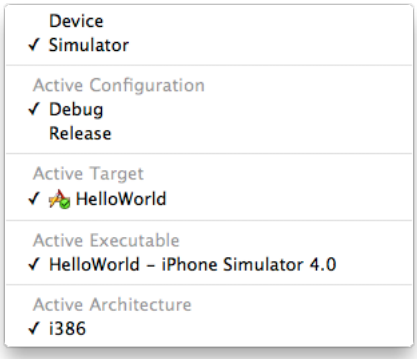
1. Choose whether you want to run your application on a device or in the simulation environment.

   The first two options in the Overview toolbar menu let you choose between these alternatives.

2. Choose the device or the iOS Simulator version on which you want to run the application.

   The Active Executable section of the Overview toolbar menu lets you specify a device or iOS Simulator version.

Figure 3-3 The Overview pop-up menu in the Project-window toolbar



Building Your Application

To start the build process, choose Build > Build.

The status bar in the project window indicates that the build was successful or that there are build errors or warnings. You can view build errors and warnings in the text editor or the Project window.

Important: When building for the simulation environment, the generated binary runs only on the targeted iOS Simulator release. It doesn't run on earlier or later releases of the simulator.

iOS devices support two instruction sets, ARM and Thumb. Xcode uses Thumb instructions by default because using Thumb typically reduces code size by about 35 percent relative to ARM. Applications that have extensive floating-point code might perform better if they use ARM instructions rather than Thumb. You can turn off Thumb for your application by turning off the Compile for Thumb build setting.

If the build completes successfully, you can proceed to run your application as described in "Running Your Application" (page 41).

To learn more about the build process, see "Building Products".

Solving Build Errors

This section lists build errors you may experience while building your application and provides suggestions for solving them.

Provisioning Profile Errors

When building for a device, if Xcode has trouble installing your application onto your device due to a problem with your provisioning profile, ensure that your provisioning profile is properly configured in the iOS Developer Program Portal (see "Becoming a Member of the iOS Developer Program" (page 49)). If necessary reinstall it on your computer and device, as described in "Preparing Your Mac for iOS Development" (page 49).

Code Signing Errors

When the code signing identity you're using to sign your binary has expired or is otherwise invalid, you may get the following build-error message:

```
Code Signing Identity 'iPhone Developer' does not match any valid, non-expired,
 code-signing certificate in your keychain.
```

To solve the error, choose a valid code signing identity, as described in "Setting Your Code Signing Identity" (page 35).

Application ID Errors

Application ID build errors may be produced due to a conflict between the application ID set in your provisioning profile (obtained through the Program Portal) and the application ID specified by the `CFBundleIdentifier` property of your application. To avoid such errors, ensure that the application ID in the profile is set to `com.<organization_name>.*` and your application's `CFBundleIdentifier` property is set to `com.<organization_name>.<application_name>`. That is, if the application ID in your provisioning profile specifies a domain set, the application ID specified by the `CFBundleIdentifier` property of your application must not redefine the domain set, it may only reduce it. Table 3-2 and Table 3-3 identify valid and invalid pairings of these items.

Table 3-2 Valid App ID/`CFBundleIdentifier` property pair

Provisioning profile	App ID	`com.mycompany.*`

Application bundle	`CFBundleIdentifier`	`com.mycompany.MyApp`

Table 3-3 Invalid App ID/`CFBundleIdentifier` property pair

Provisioning profile	App ID	`com.mycompany.MyApp.*`
Application bundle	`CFBundleIdentifier`	`com.mycompany.MyApp`

To learn about the structure of iOS application binaries, including details about the `CFBundleIdentifier` property, see "The Application Bundle" in *iOS Application Programming Guide*.

To learn about solving other build errors, use the techniques described in "Viewing Errors and Warnings" in *Xcode Project Management Guide* to fix them.

Running Your Application

When you run your application, Xcode installs it in iOS Simulator or on a device, and launches it.

Once running, you can ensure that your application performs as you intend using all the capabilities of your device. You should especially ensure that your application uses the device's resources—CPU, memory, battery, and so on—as efficiently as possible. See "Tuning Applications" (page 75) for more information.

To run your application, choose Run > Run or Run > Debug.

Troubleshooting: If you get the "Failed to start remote debug server" error message while trying to debug your application on your device, your device may not be running the iOS release that corresponds to your iOS SDK. For more information, see "Installing iOS" (page 54).

Streamlining the Build-and-Run Workflow

In addition to the Build, Run, and Debug commands, Xcode provides convenience commands that perform these operations as a single task. These commands are Build and Run and Build and Debug.

Managing Application Data

As you develop your application, you might need to rely on user settings and application data to remain on iOS Simulator or your development device between builds. Xcode doesn't remove any user settings or application data as you build your application and install it on its host. But you may need to erase that information as part of testing your application the way users will use it. To do so, remove the application from the Home screen. See "Uninstalling Applications" (page 47) for details.

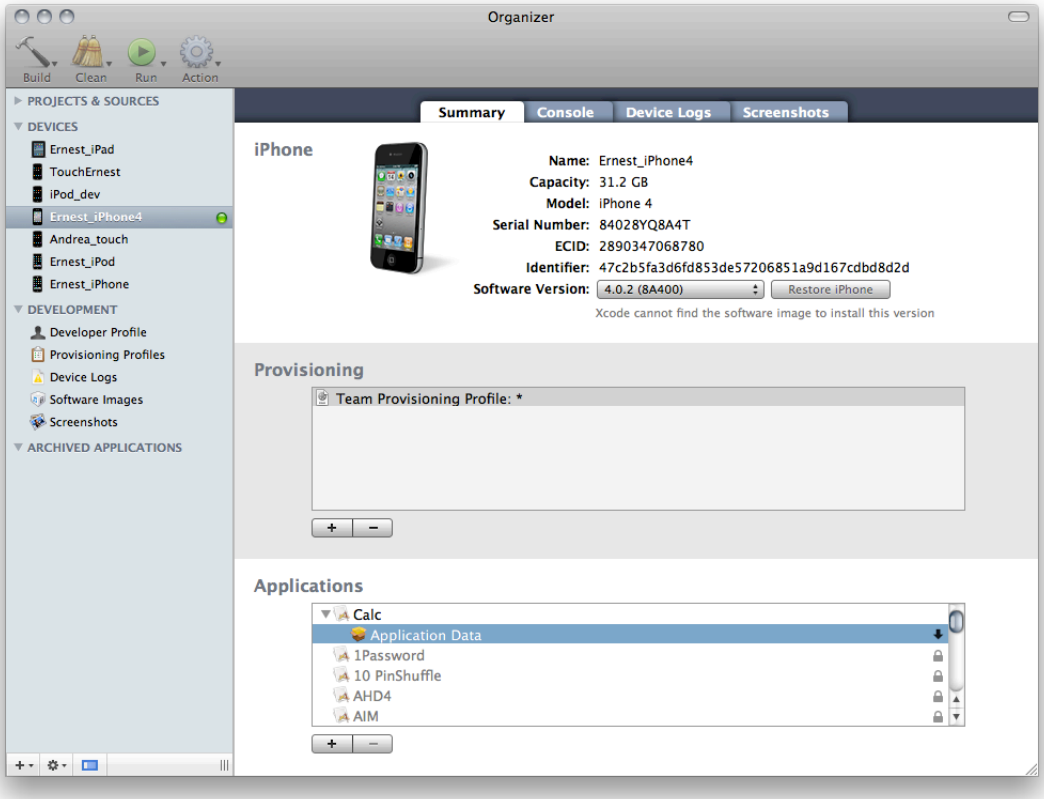
As described in "File and Data Management" in *iOS Application Programming Guide*, iOS applications can access only files that reside in the application's local file system. You can view your application's local file system in your device and in iOS Simulator.

To view your application's iOS Simulator–based file system, navigate to the `~/Library/Application Support/iOS Simulator/<sdk_version>/Applications` directory in the Finder. Then open each directory in the `Applications` directory to find your application's binary file. Alongside the binary file are the directories that make up your application's local file system.

To make a copy of your application's device–based file system to your Mac:

1. In Xcode, choose Window > Organizer.
2. In the Organizer, select your device in the Devices list.
3. In the Summary pane, click the disclosure triangle next to your application.
4. Click the download button (the down-pointing arrow to the right of the Application Data package), as shown in Figure 3-4.
5. In the dialog that appears, choose a location for the file-system copy.

Figure 3-4 Downloading an application's device-based local file system



To restore your application's file system to a backup on your Mac:

1. In Xcode, open the Organizer.
2. In the Finder, locate the directory containing the backup.

3. Drag the backup to your application in the Summary pane of the Organizer.

Further Exploration

To learn more about using Xcode to build and run applications, see *Xcode Project Management Guide*.

Using iOS Simulator

The iOS simulation environment lets you build and run your iPhone or iPad application on your computer. You use the simulation environment to:

- Find and fix major problems in your application during design and early testing.
- Learn about the Xcode development experience and the iOS development environment before becoming a member of the iOS Developer Program.
- Lay out and test your application's user interface.
- Measure your application's memory usage before carrying out detailed performance analysis on iOS devices.

A major part of the iOS simulation environment is the **iOS Simulator application**. This application presents the iPhone or iPad user interface in a window on your computer. The application provides several ways of interacting with it using your keyboard and mouse to simulate taps and device rotation, among other gestures.

Important: iOS 4.0 and later use the same Objective-C runtime as Mac OS X v10.6. iOS 3.2 and earlier use the Mac OS X v10.5 Objective-C runtime. Because of this change, binaries generated with an iOS SDK distribution earlier than 4.0 do not run in the simulator that's part of the iOS SDK 4.0 and later distributions. After moving from iOS SDK 3.2 and earlier distributions to a 4.0 or later distribution, you must rebuild your iOS Simulator binaries to run them in the simulator. If you use licensed static libraries in your application, you must obtain versions of them generated with an iOS SDK 4.0 or later distribution. For more information about the Objective-C runtime, see *Objective-C Runtime Reference*.

The following sections describe the ways in which you use your computer's input devices to simulate the interaction between users and their devices. They also show how to uninstall applications from the simulator and how to reset the contents of the simulator.

Setting the Simulation-Environment Device Family and iOS Version

iOS Simulator can simulate two device families (iPhone and iPad) and more than one iOS release.

To specify the device family you want to simulate, choose Hardware > Device, and choose the device family.

To set the iOS release used in the simulation environment, choose Hardware > Version, and choose the version you want to test on.

Manipulating the Hardware

iOS Simulator lets you simulate most of the actions a user performs on her device. When you're running your application in iOS Simulator, you can carry out these hardware interactions through the Hardware menu:

- **Rotate Left.** Rotates the simulator to the left.
- **Rotate Right.** Rotates the simulator to the right.
- **Shake Gesture.** Shakes the simulator.
- **Home.** Takes the simulator to the Home screen.
- **Lock.** Locks the simulator.
- **Simulate Memory Warning.** Sends the frontmost application low-memory warnings. For information on how to handle low-memory situations, see "Observing Low-Memory Warnings" in *iOS Application Programming Guide*.
- **Toggle In-Call Status Bar.** Toggles the status bar between its normal state and its in-call state. The status bar is taller in its in-call state than in its normal state. This command shows how your application's user interface looks when the user launches your application while a phone call is in progress.
- **Simulate Hardware Keyboard.** Toggles the software keyboard on iPad simulation. Turn off the software keyboard to simulate using a keyboard dock or wireless keyboard with an iPad device.

Performing Gestures

Table 4-1 lists gestures you can perform on the simulator (see *iOS Human Interface Guidelines* for gesture information).

Table 4-1 Performing gestures in iOS Simulator

Gesture	Desktop action
Tap	Click.
Touch and hold	Hold down the mouse button.
Double tap	Double click.
Swipe	1. Place the pointer at the start position. 2. Hold the mouse button. 3. Move the pointer in the swipe direction and release the mouse button.
Flick	1. Place the pointer at the start position. 2. Hold the mouse button. 3. Move the pointer quickly in the flick direction and release the mouse button.

Gesture	Desktop action
Drag	1. Place the pointer at the start position. 2. Hold down the mouse button. 3. Move the pointer in the drag direction.
Pinch	1. Hold down the Option key. 2. Move the circles that represent finger touches to the start position. 3. Move the center of the pinch target by holding down the Shift key, moving the circles to the desired center position, and releasing the Shift key. 4. Hold down the mouse button, move the circles to the end position, and release the Option key.

Installing Applications

Xcode installs applications in iOS Simulator automatically when you build your application targeting the simulation environment. See "Building and Running Applications" (page 33) for details.

In iOS Simulator, you can install only applications targeted at the simulation environment. You cannot install applications from the App Store in the simulator.

Uninstalling Applications

To uninstall applications you have installed on the simulator use the same method used to uninstall applications from devices:

1. Place the pointer over the icon of the application you want to uninstall and hold down the mouse button until the icon starts to wiggle.
2. Click the icon's close button.
3. Click the Home button to stop the icon wiggling.

Resetting Content and Settings

To set the user content and settings of the simulator to their factory state and remove the applications you have installed, choose iOS Simulator > Reset Content and Settings.

Core Location Functionality

The relocation reported by the CoreLocation framework in the simulator is fixed at the following coordinates (accuracy 100 meters), which correspond to 1 Infinite Loop, Cupertino, CA 95014.

- Latitude: 37.3317 North
- Longitude: 122.0307 West

Viewing iOS Simulator Console Logs

To learn how to view your application's console logs when it runs in iOS Simulator, see "Viewing Console Output and Device Logs" (page 59).

iOS Simulator File System on Your Mac

iOS Simulator stores the file systems for the iOS releases it supports in your home directory at:

```
~/Library/Application Support/iOS Simulator
```

That directory contains one subdirectory per iOS release supported by iOS Simulator. For example, it may contain `4.1` and `4.2` directories, corresponding to the file systems for the iOS 4.1 and the iOS 4.2 simulation environments.

Within each iOS-release directory, iOS Simulator stores system application preferences files in `Library/Preferences` and third-party–application preferences files in `Applications/<app_UUID>Library/Preferences`.

Hardware Simulation Support

iOS Simulator doesn't simulate accelerometer or camera hardware.

Managing Devices and Digital Identities

With iOS Simulator you can start developing iOS applications without using iOS-based devices. This way you can familiarize yourself with the API and development workflows used to develop applications. However, you must always test your applications on actual devices before publishing them to ensure that they run as intended and to tune them for performance on actual hardware.

As a registered Apple developer you can log in to the iOS Dev Center, which provides access to iOS developer documentation and lets you build iOS applications that run in iOS Simulator. (To become a registered Apple developer, visit http://developer.apple.com/programs/register/.) Being a registered Apple developer, however, doesn't allow you to run applications on iOS-based devices. To do so you must be a member of the iOS Developer Program. See "Becoming a Member of the iOS Developer Program" (page 49) for more information.

This chapter shows how to configure your computer and devices for iOS development. It also shows how to view your application's console logs and crash information, or to take screen shots of your application as it runs. The chapter also describes how to safeguard the digital identifications required to install applications in development in devices.

Becoming a Member of the iOS Developer Program

The **iOS Developer Program** provides the tools and resources you need to run applications on your development devices and distribute them to other iOS users. To become a member of the iOS Developer Program, visit http://developer.apple.com/programs/ios.

After becoming an iOS Developer Program member, you have access to the iOS Provisioning Portal in the iOS Dev Center. The **iOS Provisioning Portal** (or Portal) is a restricted-access area of the iOS Dev Center that stores information about your development devices. It also manages other resources, including development certificates and provisioning profiles, used in the development and distribution of iOS applications.

Important: If you are not a member of the iOS Developer Program, you do not have access to the iOS Provisioning Portal.

Preparing Your Mac for iOS Development

To run applications on a device, you must configure your computer and device for iOS development. This section presents an overview of the process, which the Xcode Organizer can manage for you.

Note: Configuring an iOS-based device for development does not hinder its normal operation.

In preparing your device for development, you create or obtain the following digital assets:

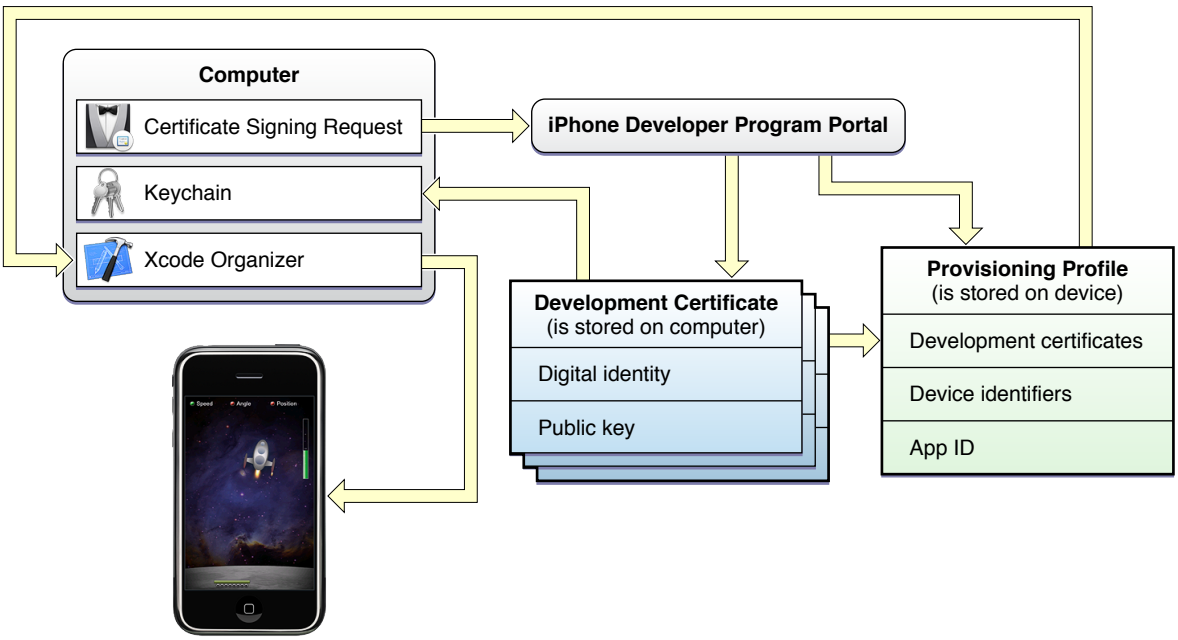
- **Certificate signing request.** A certificate signing request (CSR) contains personal information used to generate your development certificate. You submit this request to the iOS Provisioning Portal.
- **Development certificate.** A development certificate identifies an iOS application developer. After the CSR is approved, you download your developer certificate from the portal and add it to your keychain.

  When you build your iOS application with Xcode, it looks for your development certificate in your keychain; if it finds the certificate, Xcode signs your application, otherwise, it reports a build error.
- **Provisioning profile.** A provisioning profile associates one or more development certificates, devices, and an app ID (iOS application ID).

  To be able to install iOS applications signed with your development certificate on a device, you must install at least one provisioning profile on the device. This provisioning profile must identify you (through your development certificate) and your device (by listing its unique device identifier). If you're part of an iOS developer team, other members of your team, with appropriately defined provisioning profiles, may run applications you build on their devices.

Figure 5-1 illustrates the relationship between these digital assets.

Figure 5-1 Preparing computers and devices for iOS development



Provisioning a Device for Development

To run applications on a device, you must set up the device for development. This process involves two main tasks:

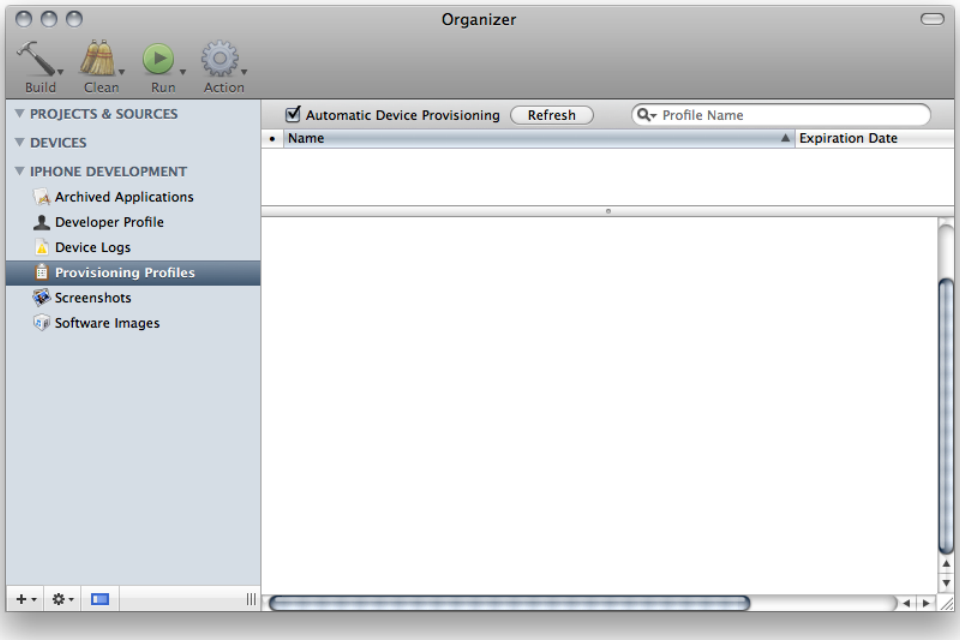
1. Creating a provisioning profile that identifies your device on the iOS Provisioning Portal.
2. Installing that provisioning profile in your device.

You can have Xcode perform these tasks automatically when you designate new devices for development, as described in "Provisioning a Device for Generic Development" (page 51). However, if your application requires a specialized provisioning profile (if it uses push notifications or in-app purchases), you need to follow the steps in "Provisioning a Device for Specialized Development" (page 53).

Provisioning a Device for Generic Development

To provision a new device for generic development:

1. Open the Xcode Organizer.
2. In the DEVELOPMENT group, select Provisioning Profiles.
3. Select the Automatic Device Provisioning option.



4. Plug in your device.

5. In the DEVICES group, select your device and click Use for Development.

6. In the dialog that appears, enter your iOS Developer Program credentials.

   Xcode logs in to the Portal and adds the new device to it. Xcode also adds a provisioning profile called "Team Provisioning Profile: *" to your device. This provisioning profile is shared among the members of your team who provision their devices automatically.

7. If Xcode offers to request a development certificate in your behalf, click Submit Request.

To ensure that your device is provisioned correctly, you can download a sample project, build it using a device SDK, and run the application it produces on your device. See, "Running Sample Applications" (page 33) for details.

Provisioning a Device for Specialized Development

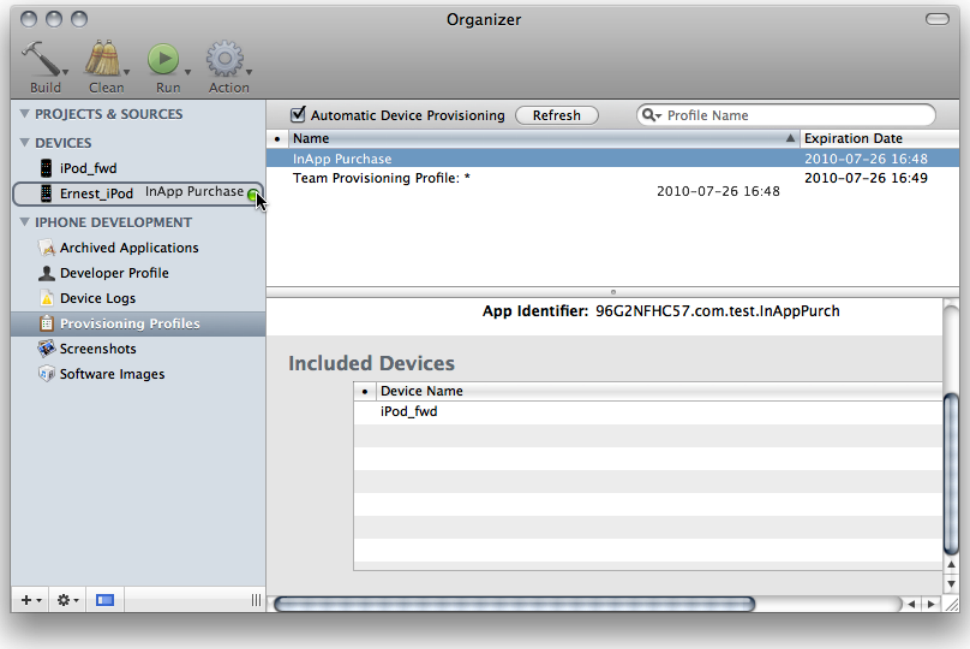
When you use specialized provisioning profiles (required to run applications that use push notifications or in-app purchases, for example), you must create those provisioning profiles in the Portal and install them on your development device.

First, follow the steps in "Provisioning a Device for Generic Development" (page 51) to add your device to the Portal.

To add a specialized provisioning profile to your device:

1. In the Portal, add your device to the specialized provisioning profile.
2. In the Xcode Organizer, select Provisioning Profiles in the DEVELOPMENT group.

   If you don't see the provisioning profile you want to add to your device in the provisioning profile list, click Refresh.
3. Drag the specialized provisioning profile to your device in the DEVICES group.



Installing iOS

When you develop applications using the iOS SDK, you should test those applications on devices running the iOS version the SDK targets. You can download the latest release of the iOS SDK from the iOS Dev Center. Use iTunes to install the latest iOS release onto your device.

Note: During seed periods, you can download seed releases of the iOS SDK and iOS from the iOS Dev Center.

To restore a device:

1. Launch Xcode and open the Organizer window.
2. Plug the device into your computer.
3. Select the device in the Devices list.
4. From the Software Version pop-up menu, choose the version of iOS you want to place on the device.

   If you're using a seed release of the iOS SDK and the version of iOS you want to install is not listed in the Software Version pop-up menu:

   a. Download the iOS seed that corresponds to your iOS SDK seed from http://developer.apple.com.

   Important: You must be a member of the iOS Developer Program to be able to download seed releases of iOS.

   b. From the Software Version pop-up menu, choose Other Version.

   c. Navigate to the disk image containing the iOS developer software and click Open.

      Xcode extracts the iOS software from the disk image. You can dispose of the disk image you downloaded.

   d. From the Software Version pop-up menu, choose the newly downloaded iOS version.

   During non-seed periods, you can install iOS only using iTunes.
5. Click Restore iPhone or Restore iPod, depending on your device's type.
6. Use iTunes to name your device.

Running Applications on a Device

After following the instructions in "Preparing Your Mac for iOS Development" (page 49) and "Installing iOS" (page 54) (if necessary) you can run your application on your development device.

You tell Xcode which iOS SDK to use to build your application by setting the active SDK. Specify that you want to run your application on a device by choosing Device from the Overview toolbar menu. See "Specifying Where to Place Your Application" (page 39) for details.

Capturing Screen Shots

Screen shots help to document your application. This is also how you create your application's default image, which iOS displays when the user taps your application's icon. You can capture screen shots of your device's screen from the Organizer or directly on your device.

To capture a screen shot from the Organizer:

1. Configure your application's screen for the screen shot.

   Depending on your application's workflow, you may need to place breakpoint in your code and run your application until it reaches that point.

2. Open the Organizer window, select your device, and click Screenshots.

3. Click Capture.

   To make that screen shot your application's default image, click Save As Default Image.

   To get a PNG file of the screen shot, drag it to the Desktop.

If you have iPhoto installed on your computer, you may capture screen shots directly on your device and import them into your iPhoto library.

To capture a screen shot on your device, press the the Lock and Home buttons simultaneously. Your screen shot is saved in the Saved Photos album in the Photos application.

Note: Although the default image includes the status bar as it looked when the screen shot was captured, iOS replaces it with the current status bar when your application launches.

Managing Your Digital Identities

When you request a certificate from the iOS Provisioning Portal, a public/private key pair is generated. The public key is included in your certificate. The private key is stored in your keychain. With these items, Xcode code-signs the applications you build with it. If you need to use another computer to develop iOS applications, you must transfer these digital-identification items to the other computer. You can do this in the Xcode Organizer.

To export your digital-identification items to a secure file, follow these steps:

1. Open the Xcode Organizer.

2. In the DEVELOPMENT group, select Developer Profile.

3. Click Export Developer Profile.
4. Name the file, select a location for it, enter a password to secure the file, and click Save.

Now, when you need to develop iOS applications on another computer, import your digital-identification items into it by performing these steps:

1. Copy the developer-profile archive to the second computer.
2. On the second computer, launch Xcode.
3. Open the Organizer.
4. In the DEVELOPMENT group, select Developer Profile.
5. Click Import Developer Profile.
6. Locate the archive, enter the password used to secure it, and click Open.

Debugging Applications

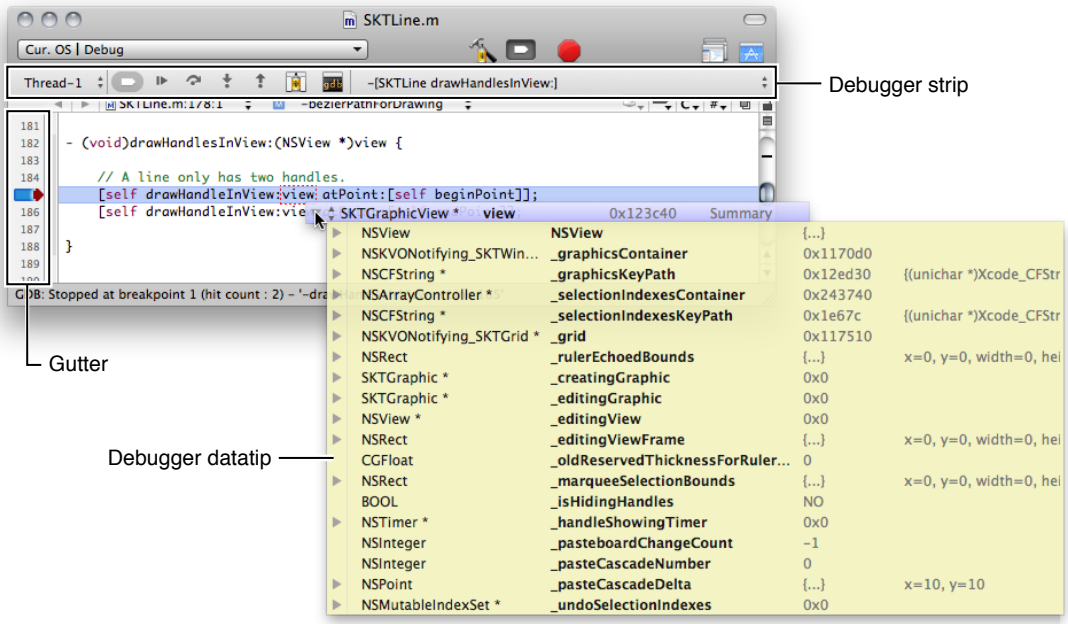
This chapter describes the Xcode debugging facilities.

Part of your debugging workflow may require to view or manipulate data your application writes in its file system. For example, you may need to edit the data the application has stored to recreate a particular condition you want to test. See "Managing Application Data" (page 41) for details about manipulating your application's data.

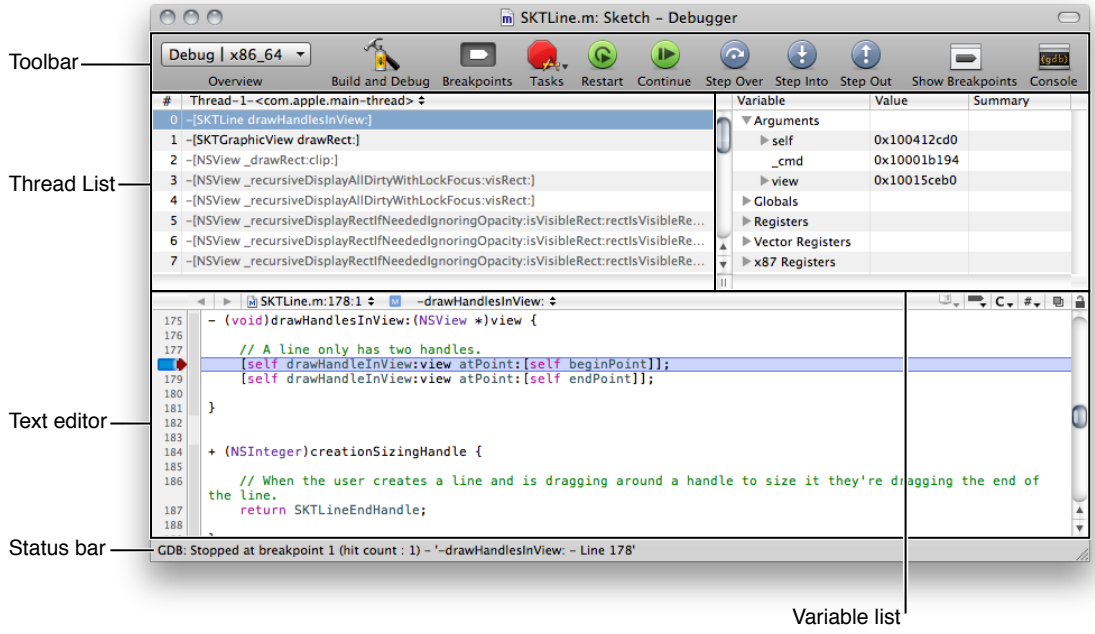
Debug Facilities Overview

Xcode provides several debugging environments you can use to find and squash bugs in your code:

- **The text editor.** The text editor allows you to debug your code right *in* your code. It provides most of the debugging features you need. You can:
  - Add and set breakpoints
  - View your call stack per thread
  - View the value of variables by hovering the mouse pointer over them
  - Execute a single line of code
  - Step in to, out of, or over function or method calls



- **The Debugger window.** When you need to perform more focused debugging, the Debugger window provides all the debugging features the text editor provides using a traditional interface. This window provides lists that allow you to see your call stack and the variables in scope at a glance.



- **The GDB console.** A GDB console window is available for text-based debugging.

Important: To debug successfully on a device, you must install on your computer the iOS SDK corresponding to the iOS release installed on the device. That is, you cannot debug an application on a device running iOS 4.2 if you do not have the iOS 4.2 SDK installed on your computer.

For more information about the Xcode debugging facilities, see *Xcode Debugging Guide*.

Viewing Console Output and Device Logs

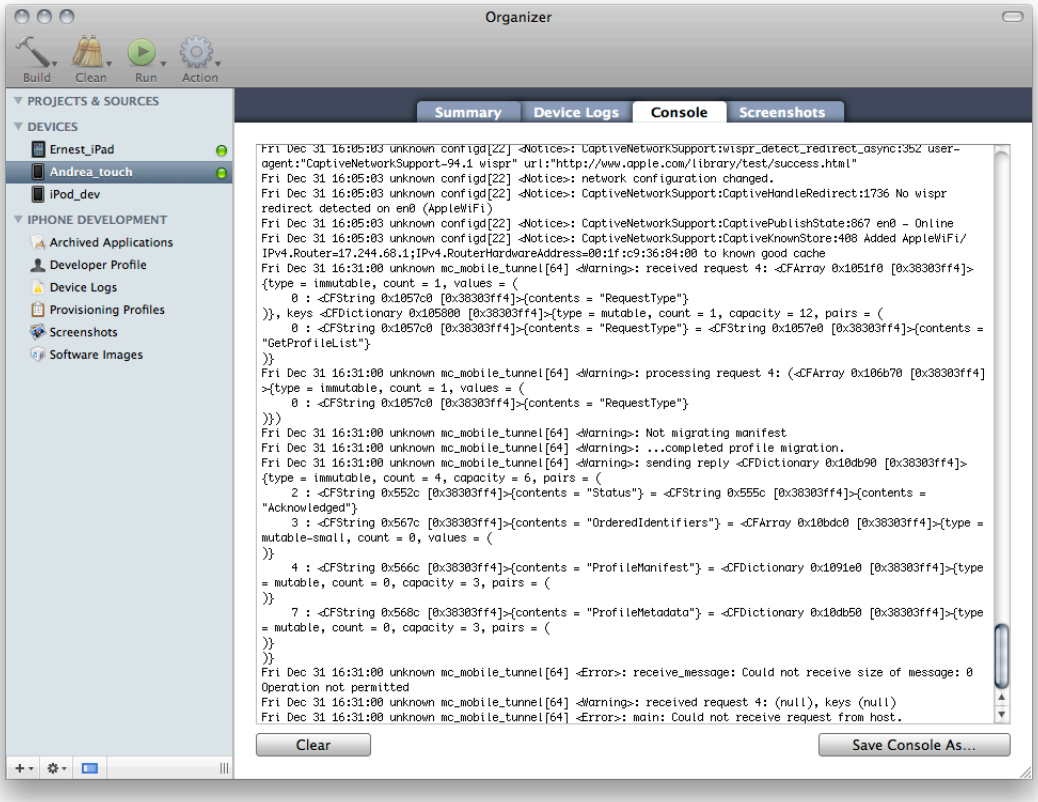
The iOS frameworks, such as UIKit, produce log entries to the console to indicate, among other things, when an unexpected event occurs. You can produce console messages in your iOS applications, too. One way to produce console logs is to use the `NSLog` function. In addition to the Xcode debugger, console logs may help you analyze your application's logic and track down bugs.

When running your application on iOS Simulator, you can access its console logs in the Console application (located in `/Applications/Utilities`). When you run the application on your development device, log entries from your application appear in the Xcode Organizer.

To view a device's console output:

1. Open the Organizer window.
2. Select the device whose console output you want to view.

3. Click Console.



You can save the console output to a file by clicking Save Console As.

The Device Logs pane in the Organizer contains information about application crashes. You may have to unplug your device and plug it in again to refresh the crash list.

For more information about crash logs, see Understanding and Analyzing iPhone OS Application Crash Reports.

Finding Memory Leaks

If you fix a memory leak and your program starts crashing, your code is probably trying to use an already-freed object or memory buffer. To learn more about memory leaks, see “Finding Memory Leaks”.

You can use the NSZombieEnabled facility to find the code that accesses freed objects. When you turn on NSZombieEnabled, your application logs accesses to deallocated memory, as shown here:

```
2008-10-03 18:10:39.933 HelloWorld[1026:20b] *** -[GSFont ascender]: message sent to
deallocated instance 0x126550
```

To activate the NSZombieEnabled facility in your application:

1. Choose Project > Edit Active Executable to open the executable Info window.

2. Click Arguments.
3. Click the add (+) button in the “Variables to be set in the environment” section.
4. Enter `NSZombieEnabled` in the Name column and `YES` in the Value column.
5. Make sure that the checkmark for the `NSZombieEnabled` entry is selected.

For more information about configuring executable environments, see “Configuring Executable Environments” in *Xcode Project Management Guide*.

You can use the Leaks instrument to easily find memory leaks. For more information, see "Memory Instruments" in *Instruments User Guide*.

Unit Testing Applications

Unit tests help you write robust and secure code. Xcode provides an easy-to-use flexible unit-testing environment that you can use to ensure your code works as designed as it goes through changes.

This chapter introduces unit testing and describes how you can take advantage of it in your projects.

For a case study in unit-testing adoption and usage, see:

http://www.macdevcenter.com/pub/a/mac/2004/04/23/ocunit.html

Xcode is already configured for unit testing. You don't need to install additional software.

Unit Testing Overview

Unit testing lets you specify behavior that your code must exhibit to ensure that its functionality remains unchanged as you modify it to, for example, make performance improvements or fix bugs. A **test case** exercises your code in a specific way; if the results vary from the expected results, the test case fails. A **test suite** is made up of a set of test cases. You can develop one or more test suites to test different aspects of your code.

Unit tests are the basis of test-driven development, which is a style of writing code in which you write test cases before writing the code to be tested. This development approach lets you codify requirements and edge cases for your code before you get down to writing it. After writing the test cases, you develop your algorithms with the aim of passing your test cases. After your code passes the test cases, you have a foundation upon which you can make improvements to your code, with confidence that any changes to the expected behavior (which would result in bugs in your product) are identified the next time you run the tests.

Even when not using test-driven development, unit tests can help reduce the introduction of bugs in your code. You can incorporate unit testing in a working application to ensure that future source-code changes don't modify the application's behavior. As you fix bugs, you can add test cases that confirm the bugs are fixed. However, adding unit tests to a project that's not designed with unit testing in mind may require redesigning or refactoring parts of the code to make them testable.

The Xcode unit-testing environment is based on the open-source SenTestingKit framework. This framework provides a set of classes and command-line tools that let you design test suites and run them on your code.

Xcode offers two types of unit tests: *logic tests* and *application tests*.

- **Logic tests.** These tests check the correct functionality of your code in a clean-room environment; that is, your code is not run inside an application. Logic tests let you put together very specific test cases to exercise your code at a very granular level (a single method in class) or as part of a workflow (several methods in one or more classes). You can use logic tests to perform stress-testing of your code to ensure that it behaves correctly in extreme situations that are unlikely in a running application. These tests help

you produce robust code that works correctly when used in ways that you did not anticipate. Logic tests are iOS Simulator SDK–based; however, the application is not run in iOS Simulator: The code being tested is run during the corresponding target's build phase.

- **Application tests.** These tests check the functionality of your code in a running application. You can use application tests to ensure that the connections of your user-interface controls (outlets and actions) remain in place, and that your controls and controller objects work correctly with your object model as you work on your application. Because application tests run only on a device, you can also use these tests to perform hardware testing, such as getting the location of the device.

Setting Up Testing

Logic unit tests (introduced in "Unit Testing Overview" (page 63)) allow you to perform exhaustive, highly tailored testing of your code. To perform logic tests, you build a unit-test bundle using the iOS Simulator SDK. When you build the unit-test bundle, Xcode runs the test suites that are part of the bundle. You incorporate test suites into a test bundle by adding `SenTestCase` subclasses to the bundle. These classes contain test-case methods that call the API to exercise it and report whether the calls produced the expected results. Xcode reports whether the tests passed or failed in text editor windows and the Build Results window.

Application unit tests let you test your code within an application running on an iOS device, with access to the resources available in the Cocoa Touch framework.

This section describes how to set up a project for each type of unit test.

Setting Up Logic Testing

To set up a logic unit-test bundle in a project:

1. Add an iOS unit-test bundle target to the project. Name the target `LogicTests` (you can use any name you like for your unit-test bundles, but you should include the suffix *Tests* to identify them as such).

   For details about adding targets to a project, see "Creating Targets" in *Xcode Build System Guide*.

2. Set the LogicTests target as the active target.

3. Add a group called Tests to the Group & Files list, and select that group in the list.

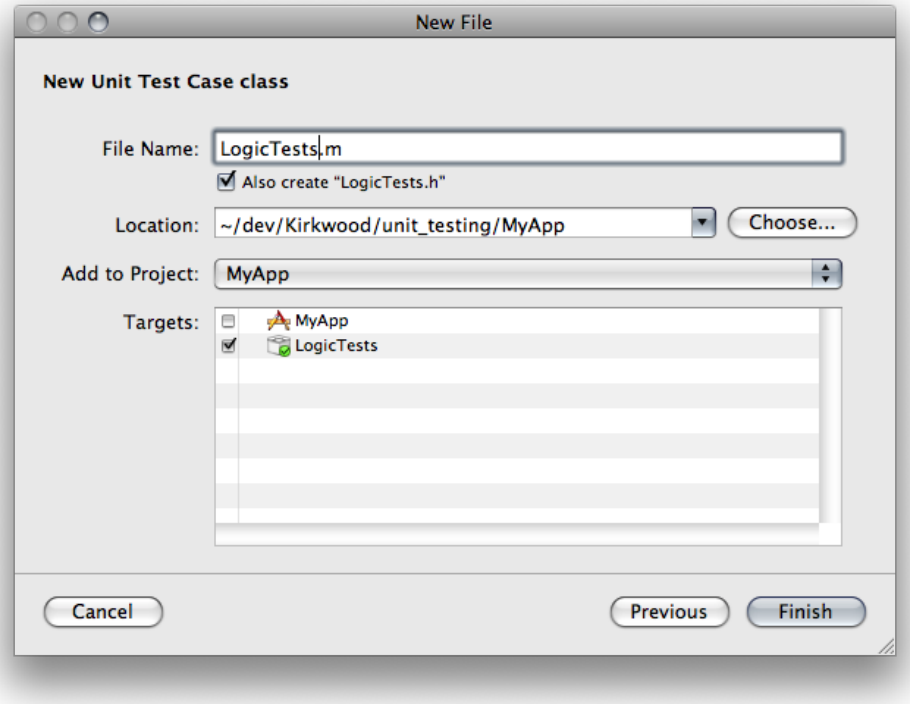
4. Add a unit-test class to the unit-test-bundle target. Each unit-test class in the bundle makes up a test suite.

   a. Choose File > New File, select the iOS > Cocoa Touch Class > "Objective-C test case class" class template, and click Next.

   b. Name the class `LogicTests` (you can use any name here, too).

   c. Select the option to create the header file.

d. Ensure that LogicTests is the only target selected in the target list.



5. Change `LogicTests.h` so that it looks like this:

```
#import <SenTestingKit/SenTestingKit.h>
#import <UIKit/UIKit.h>

@interface LogicTests : SenTestCase {
}
@end
```

6. Change `LogicTests.m` so that it looks like this:

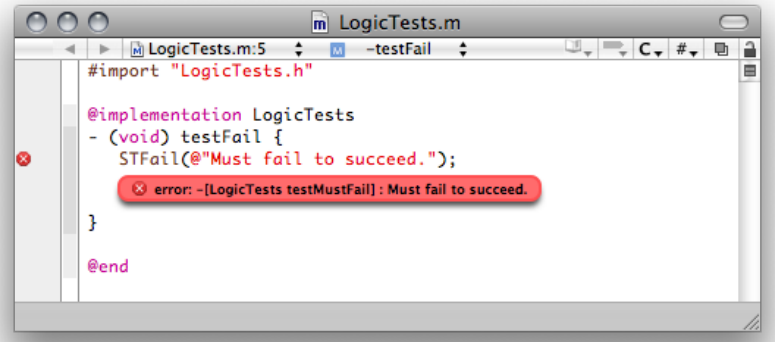
```
#import "LogicTests.h"

@implementation LogicTests
- (void) testFail {
      STFail(@"Must fail to succeed.");
}
@end
```

7. Set the base SDK for the project to Latest iOS and choose Simulator from the Overview menu.

   For more information about the Base SDK build setting, see "Setting the Base SDK" (page 34).

8. Set the active target to LogicTests and choose Build > Build. If the unit-test bundle is configured correctly, the build fails and Xcode displays an error message in the text editor.



9. Now make the test case pass by changing the highlighted lines in `LogicTest.m`:

```
#import "LogicTests.h"

@implementation LogicTests
- (void) testPass {
      STAssertTrue(TRUE, @"");
}
@end
```

At this point you have a correctly configured logic–unit-test bundle. See "Writing Tests" (page 71) to learn how to add test cases to it.

Setting Up Application Testing

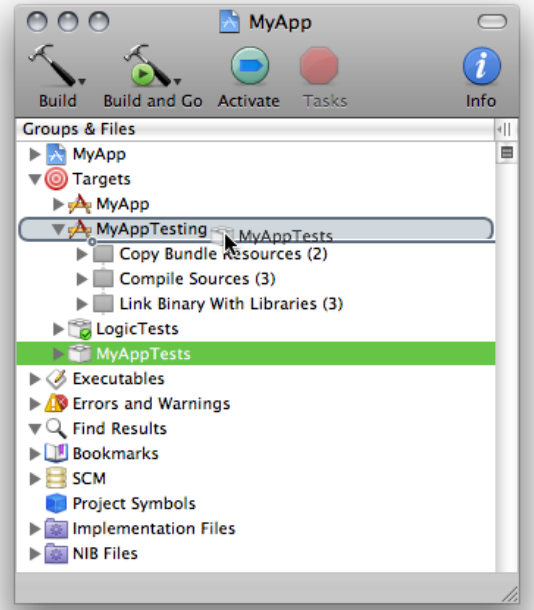
To set up an application–unit-test bundle in a project:

1. Make a copy of the target that builds the application to test by choosing Duplicate from its shortcut menu, and name it `<application_name>Testing` (for example, MyAppTesting). (You can use any name for this target, but you should include the suffix *Testing* to identify it as a target used to run application unit tests.) The only purpose of this target is to run application unit tests.

2. Add an iOS unit-test bundle target to the project. Name the target `<application_name>Tests` (for example, MyAppTests).

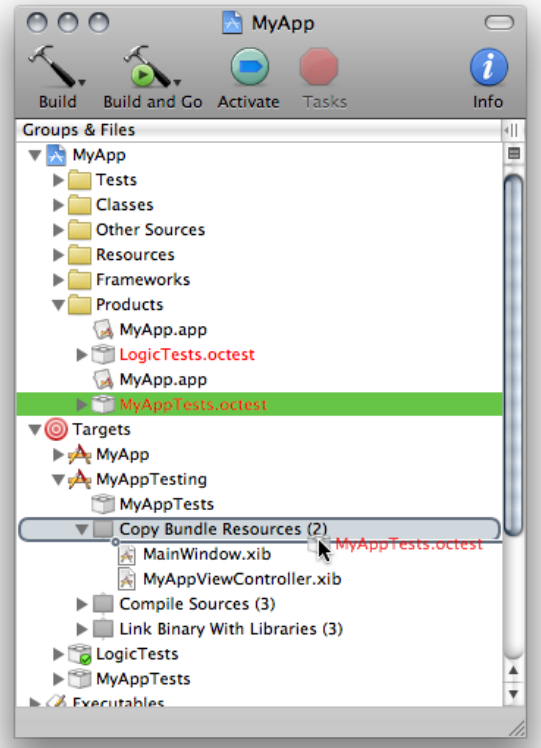
   For details about adding targets to a project, see "Creating Targets" in *Xcode Build System Guide*.

3. Make the MyAppTesting target dependent on the MyAppTests target by dragging MyAppTests to MyAppTesting.



Making the MyAppTesting target dependent on the MyAppTests target ensures that when you build MyAppTesting, MyAppTests gets built first (if you have modified MyAppTests since it was last built).

4. Embed the MyAppTests bundle into the MyAppTesting bundle by dragging the `MyAppTests.octest` product to the MyAppTesting target Copy Bundle Resources build phase.



5. Add a group called Tests to the Groups & Files list (if it doesn't already exist), and select that group in the list.

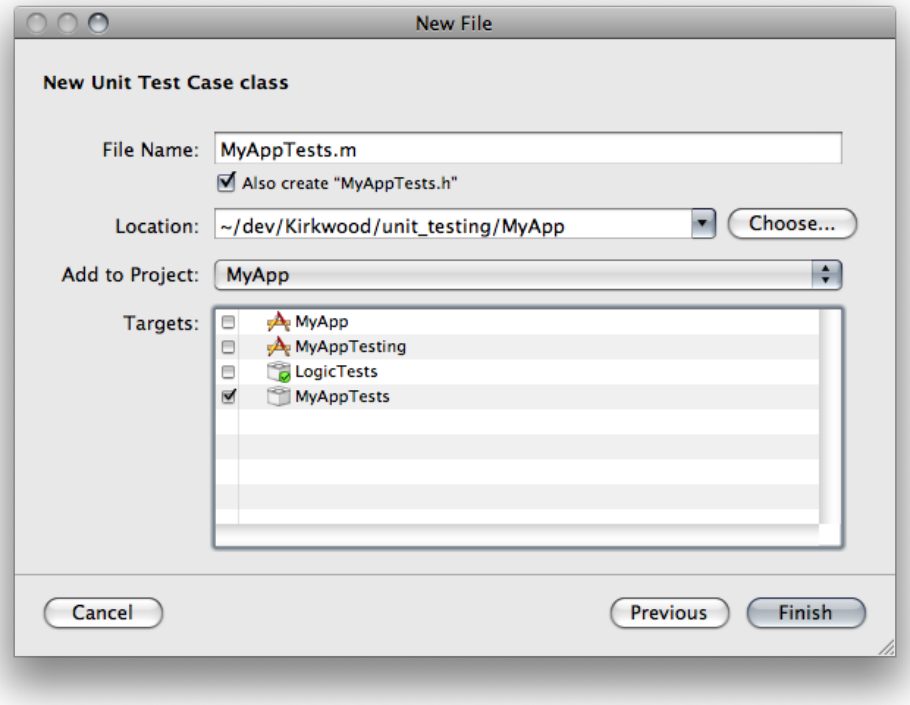
6. Add a unit-test class to the MyAppTests target. Each unit-test class in the bundle makes up a test suite.

    a. Choose File > New File, select the iOS > Cocoa Touch Class > "Objective-C test case class" template, and click Next.

    b. Name the class `MyAppTests` (you can use any name for this class, but it should have the suffix *Tests* for easy identification).

    c. Select the option to create the header file.

d. Ensure that MyAppTests is the only target selected in the target list, and click Finish.



7. Change `MyAppTests.h` so that it looks like this:

```
#import <SenTestingKit/SenTestingKit.h>
#import <UIKit/UIKit.h>

@interface MyAppTests : SenTestCase {
}
@end
```

8. Change `MyAppTests.m` so that it looks like this:

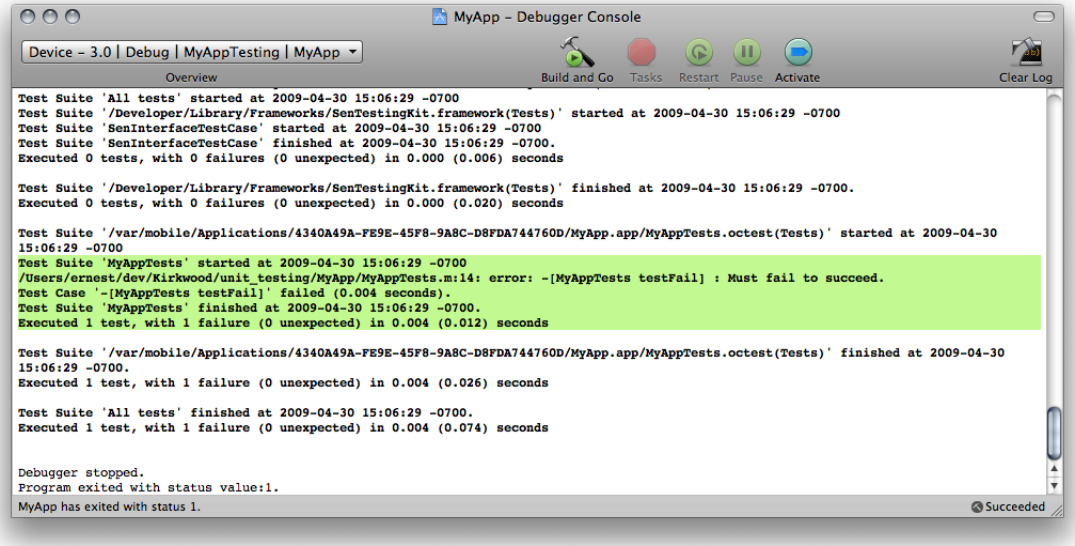
```
#import "MyAppTests.h"

@implementation MyAppTests
- (void) testFail {
      STFail(@"Must fail to succeed.");
}
@end
```

9. Set the Base SDK for the project to Latest iOS and choose Device from the Overview menu.

   For more information about the Base SDK build setting, see "Setting the Base SDK" (page 34).

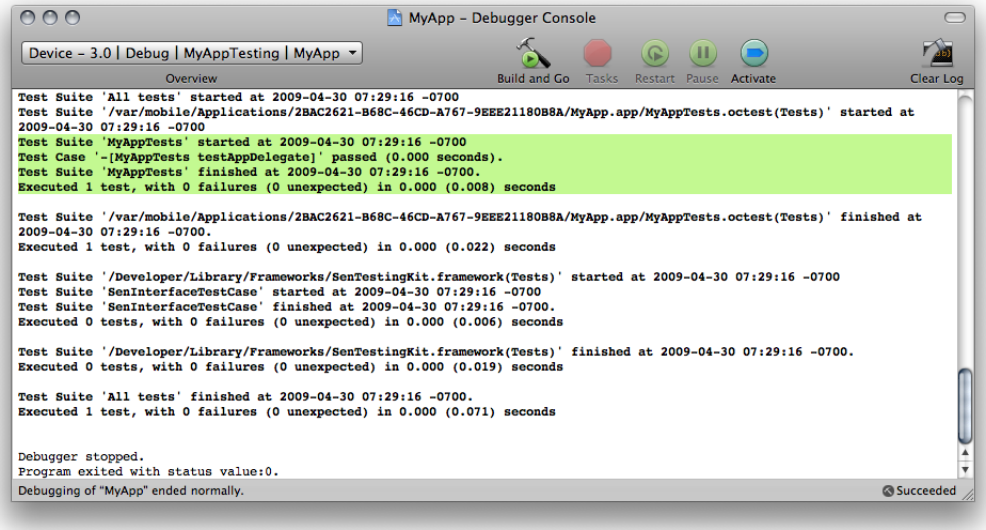
10. Set the active target to MyAppTesting and choose Build > Build and Run. Xcode builds your application, installs and launches it on your device, and runs the test suite, which fails. You can see the test results in the console. This outcome confirms that application unit testing is set up correctly.



11. Now make the test suite pass by changing the highlighted lines in `MyAppTests.m`:

```
#import "MyAppTests.h"

@implementation LogicTests
- (void) testAppDelegate {
   id app_delegate = [[UIApplication sharedApplication] delegate];
   STAssertNotNil(app_delegate, @"Cannot find the application delegate.");
}
@end
```



Now that you have a correctly configured application unit-test bundle, see "Writing Tests" (page 71) to learn how to add test cases to it.

Writing Tests

After configuring a unit-test bundle with a unit-test class (which implements a test suite), you add cases to the suite by adding test-case methods to the class. A **test-case method** is an instance method of a unit-test class that's named `test...`, with no parameters, and whose return type is `void`. Test-case methods call the API they test and report whether the API performed as expected—for example, whether it returns the anticipated return or whether it raises an exception. Test-case methods use a set of macros to check for the expected conditions and report their findings. "Unit-Test Result Macro Reference" (page 89) describes these macros.

For a test case to access the subject API, you may have to add the appropriate implementation files to the unit-test bundle and import the corresponding header files into your unit-test class. For an example of a project that uses unit tests, see the *iPhoneUnitTests* sample-code project.

> **Note:** iPhone unit-test–cases are written in Objective-C.

This is the structure of a test-case method:

```
- (void) test<test_case_name> {
    ...     // Set up, call test-case subject API.
    ST...   // Report pass/fail to testing framework.
    ...     // Tear down.
}
```

Each test-case method is invoked independently. Therefore, each method must set up and tear down any auxiliary variables, structures, and objects it needs to interact with the subject API. Conveniently, you can add a pair of methods to a unit-test class that are called before and after each test-case method is invoked: `setUp` and `tearDown`. Just like test-case methods, the type of both methods is `void` and they take no arguments.

This is an example of a `setUp/tearDown` method pair:

```
- (void) setUp {
    test_subject = [[[MyClass alloc] init] retain];
    STAssertNotNil(test_subject, @"Could not create test subject.");
}

- (void) tearDown {
    [test_subject release];
}
```

Note: When there's a reported failure in a `setUp` or `tearDown` call, the failure is ultimately reported in the test-case method that originated the call.

Running Tests

To ensure that changes you make to your code don't modify its correct behavior, you should run your test suites periodically, especially after making significant changes. This section shows how to run logic and application tests. It also shows how to set up an application target so that Xcode runs logic tests every time you build it.

Running Logic Tests

To run your logic tests, all you need to do is build the appropriate logic-test target. You can build such a target in two ways:

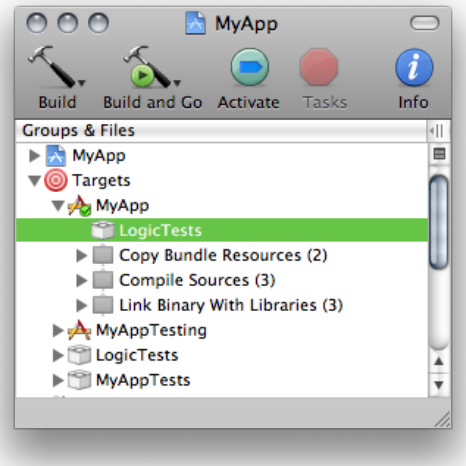
- Making the the target active and choosing Build > Build
- Choosing Build from the target's shortcut menu

The Build Results window shows the results of your tests.

You can make Xcode run your logic tests every time you build your application. This way you don't have to remember to run those tests.

Follow these steps to set up your application target so that building it runs your logic tests:

1. Drag your logic-test target into your application target. This action makes the application target depend on the logic-test target.



When you build the application target, Xcode builds the logic target first, reporting test failures in the process.

> **Note:** Xcode runs your logic tests only when building for the simulation environment.

2. In the LogicTests Info window, display the build pane.
3. Choose All Configurations from the Configuration pop-up menu.
4. Choose All Settings from the Show pop-up menu.
5. In the Architectures group of the build-setting list, set Base SDK to Latest iOS, and deselect Build Active Architecture Only.

Running Application Tests

To run your application tests, you must build, and run or debug the application-testing target. Remember that you can use such targets only to run or debug test cases; you cannot use them to run your application on your device interactively.

The console displays the results of your application tests.

Writing Testable Code

The Xcode integrated support for unit testing makes it possible for you to build test suites to support your development efforts in any way you want. You can use it to detect potential regressions in your code or to validate the behavior of your application. These capabilities can add tremendous value to your projects. In particular, they can improve the stability of your code greatly by ensuring individual APIs behave in the expected ways.

Of course, the level of stability you receive from unit testing is highly dependent on the quality of the test cases you write. These are some guidelines to think about as you write code to ensure that it's easily testable:

- **Define API requirements.** You should define requirements and outcomes for each method or function that you add to your program. These requirements should include input and output ranges, exceptions thrown and the conditions under which they are raised, and the type of returned values (especially if they are objects). Specifying requirements and making sure that requirements are met in your code help you write robust, secure code.

  See the *iPhoneUnitTests* sample-code project for an example of using exceptions to identify and report incorrect API usage by a client.

- **Write test cases as you write code.** As you write each API, write one or more test cases that ensure the API's requirements are met. It's harder to write unit tests for existing code than for code you haven't written yourself or have written recently.

- **Check boundary conditions.** If the parameter of a method expects values in a specific range, your tests should pass values that include the lowest and highest values of the range. For example, if an integer parameter can have values between 0 and 100, inclusive, three variants of your test may pass the values 0, 50, and 100, respectively.

- **Use negative tests.** Negative tests ensure your code responds to error conditions appropriately. Verify that your code behaves correctly when it receives invalid or unexpected input values. Also verify that it returns error codes or raises exceptions when it should. For example, if an integer parameter can accept values in the range 0 to 100, inclusive, you should create test cases that pass the values -1 and 101 to ensure that the API raises an exception or returns an error code.

- **Write comprehensive test cases.** Comprehensive tests combine different code modules to implement some of the more complex behavior of your API. While simple, isolated tests provide value, stacked tests exercise complex behaviors and tend to catch many more problems. These kinds of test mimic the behavior of your code under more realistic conditions. For example, in addition to adding objects to an array, you could create the array, add several objects to it, remove a few of them using different methods, and then ensure the set and number of remaining objects is correct.

- **Cover your bug fixes with test cases.** Whenever you fix a bug, write one or more tests cases that verify the fix.

Tuning Applications

Optimizing your application's performance is an important part of the development process, more so in iOS-based devices, which, although powerful computing devices, do not have the memory or CPU power that desktop or portable computers possess. You also have to pay attention to your application's battery use, as it directly impacts your customer's battery-life experience.

This chapter describes Instruments and Shark, the tools you use to measure and tune your application's performance.

Prerequisites: Follow the instructions in "Building and Running Applications" (page 33) before trying the application-tuning techniques described in this chapter on your application.

For general performance guidelines, see *iOS Application Programming Guide*.

The Instruments Application

The Instruments application lets you gather a variety of application performance metrics, such as memory and network use. You can gather data from iOS applications running in iOS Simulator or on your development devices.

It is important that your iOS applications use the resources of iOS-based devices as efficiently as possible to provide a satisfactory experience for you customers. For example, your application should not use resources in a way that makes the application feel sluggish to users or drains their batteries too quickly. Applications that use too much memory run slowly. Applications that rely on the network for their operation must use it as sparingly as possible because powering up the radios for network communications is a significant drag on the battery.

The Instruments application provides an advanced data gathering interface that lets you know exactly how your application uses resources, such as the CPU, memory, file system, and so on.

Instruments uses software-based data-gathering tools, known as instruments, to collect performance data. An **instrument** collects a specific type of data, such as network activity or memory usage. You find which instruments are available for iOS in the Instruments Library.

Although most iOS applications run in iOS Simulator and you can test most design decisions there, the simulator does not emulate a device, in particular it doesn't attempt to replicate a device's performance characteristics such as CPU speed or memory throughput. To effectively measure your application's performance as users may use it on their devices, you must use an actual device. That's because only on a device can you get an accurate representation of the runtime environment (in terms of processor speed, memory limitations, specialized hardware, and the like).

These are some limitations of iOS Simulator:

- **Maximum of two fingers.** If your application's user interface can respond to touch events involving more than two fingers, you can test that capability only on devices.
- **Accelerometer.** Although you can access your computer's accelerometer (if it has one) through the UIKit framework, its readings differ from the accelerometer readings on a device. This discrepancy stems largely from the different positioning of the screen in relation to the rest of the hardware between computers and iOS-based devices.
- **OpenGL ES.** OpenGL ES uses renderers on devices that are slightly different from those it uses in iOS Simulator. For this reason, a scene on the simulator and the same scene on a device may not be identical at the pixel level. See "Drawing with OpenGL ES" in *iOS Application Programming Guide* for details.

To measure your application's performance on a device:

1. Build and run your application on the device as described in "Building and Running Applications" (page 33).
2. Stop the application.
3. Launch Instruments.

   The Instruments application is located at `<Xcode>/Applications`. (`<Xcode>` refers to the installation location of the Xcode toolset.)
4. Choose a template, such as Activity Monitor, to create the trace document.

   A **trace document** contains one or more instruments that collect data about a process.
5. From the Target pop-up menu in the toolbar, choose the iOS-based device containing the application from which you want to collect performance data.
6. Add or remove instruments from the trace document to collect the desired data.
7. Use the Target pop-up menu to select the application to launch (the same application you ran in step 1).
8. Click Record to start collecting data and use your application, exercising the areas you want to examine.

To learn more about measuring and analyzing application performance, see *Instruments User Guide*.

The Shark Application

To complement the performance data Instruments collects, the Shark application lets you view system-level events, such as system calls, thread-scheduling decisions, interrupts, and virtual memory faults. You can see how your code's threads interact with each other and how your application interacts with iOS.

When performance problems in your code are more related to the interaction between your code, iOS, and the hardware architecture of the device, you can use Shark to get information about those interactions and find performance bottlenecks.

For information about using Shark with your iOS applications, see *Shark User Guide*.

Distributing Applications

When you're ready to distribute your application for testing or for general distribution through the App Store, you need to create an archive of the application using a distribution provisioning profile, and send it to application testers or submit it to iTunes Connect. This chapter shows how to perform these tasks.

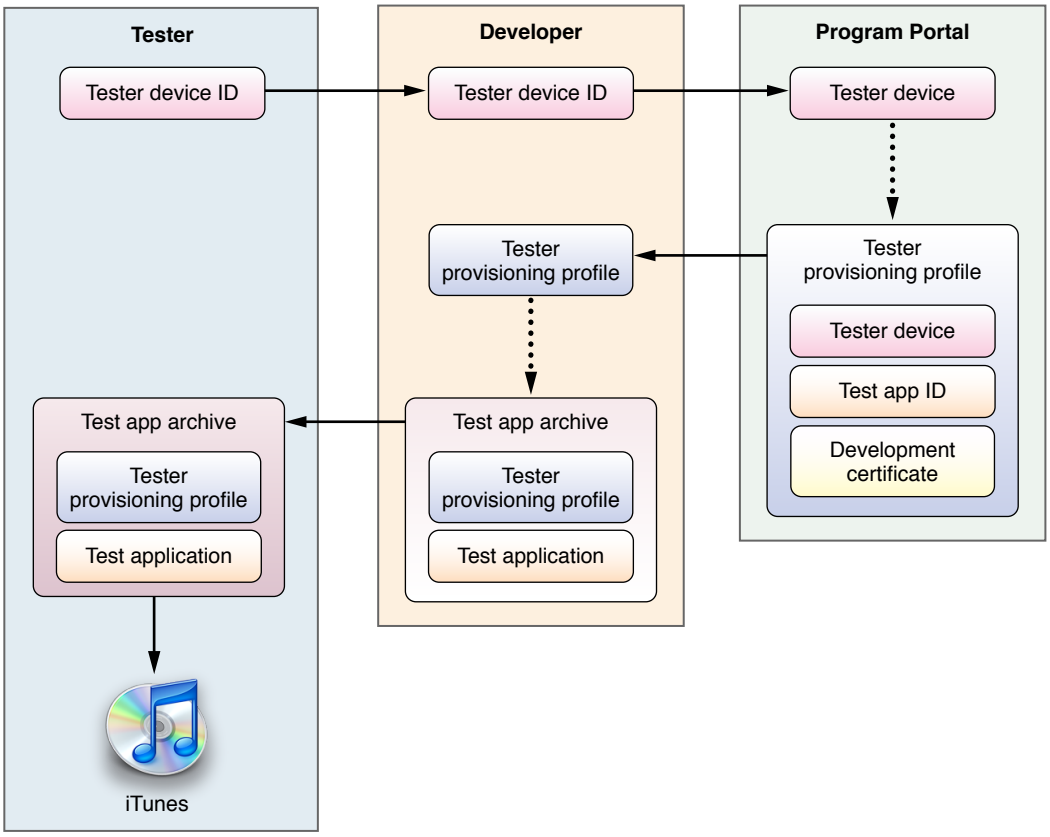
Publishing Your Application for Testing

After testing and tuning your application yourself or with the assistance of your teammates, it's always a good idea to perform wider testing with a representative sample of your application's potential users. Such testing may reveal issues that surface only with particular usage patterns. Incorporating a few non-developer users in your testing strategy lets you expose your application to a variety of usage styles, and, if such usage produces crashes in your application, allows you to collect the crash reports (also known as crash logs) from those users, to help you resolve those execution problems.

An iOS application in development can run only on devices with provisioning profiles generated by the application developer. As iOS Developer Program members, you and your fellow team members install these files on your devices as part of your development process (as described in "Managing Devices and Digital Identities"). To include users that are not part of your development team in your testing strategy, you must add them as part of your team in the iOS Provisioning Portal and issue them **test provisioning profiles** (also known as ad-hoc provisioning profiles), which allow them to install on their devices applications that have not been published to the App Store.

Figure 9-1 illustrates the process of adding users as testers and delivering your test application to them.

Figure 9-1 Adding testers to your team



Prerequisites: Test applications must be built and signed correctly to work properly. To ensure this:

- Before sending your application to testers, review the information in "Building and Running Applications" (page 33).
- To add testers to your team you must have a distribution certificate in the Portal.

To help testers obtain the information you need to add them to your testing program and to show them how to send you crash logs, you can send them the information in "Instructions for Application Testers" (page 81).

Important: To add testers to your team, you must be a member of the iOS Developer Program. See "Becoming a Member of the iOS Developer Program" (page 49) for details.

The remainder of this section describes the steps you need to perform to add testers to your team and shows how to import the crash logs they send you into the Organizer.

Adding Application Testers to Your Team

To add iOS users to your team as testers:

1. Obtain the testers' device IDs.

   The easiest way to obtain this information is through email. Have your tester follow the instructions for sending their device ID to you in "Sending Your Device ID to a Developer" (page 81).

2. Add the testers' device IDs to the Portal.

   Tip: Use the testers' email addresses as the device name.

3. If you already have a testing provisioning profile for your application in the Portal, add the testers' device IDs to it. Otherwise, create the profile with these characteristics:

Distribution method	Ad Hoc
Profile name	<Application_Name> Testing Profile
App ID	Appropriate application ID for the application being tested
Devices	Testers' device IDs

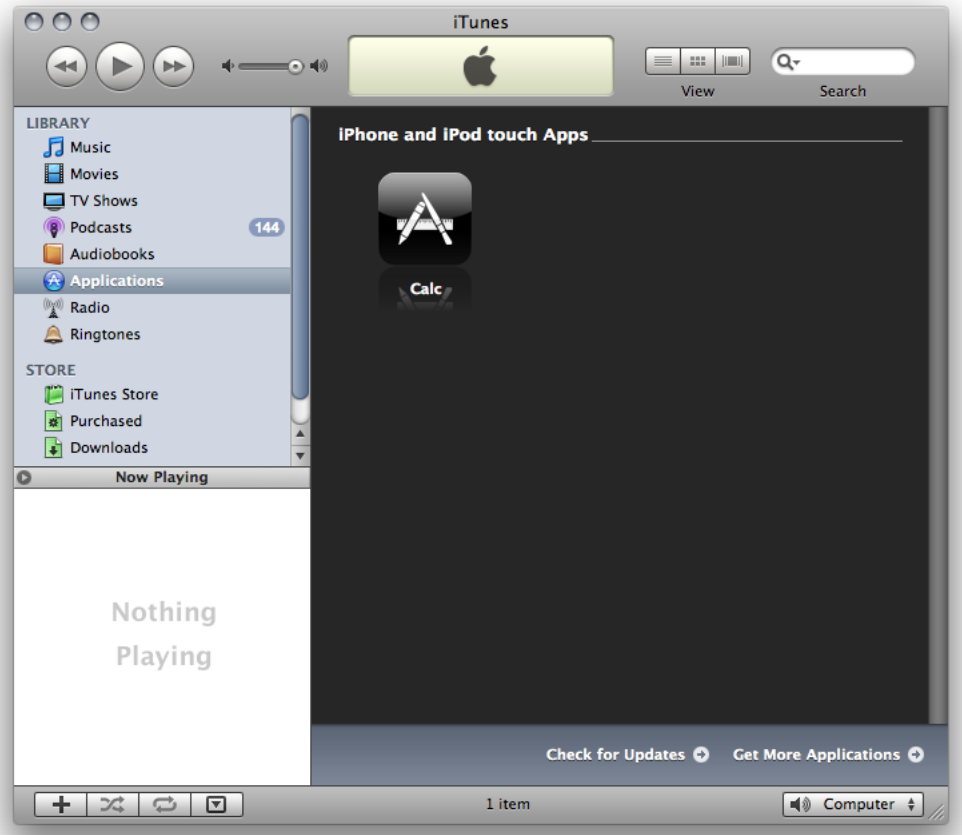
4. Download the testing profile and install it in the Xcode Organizer.

   Drag the `<Profile_Name>.mobileprovision` file to the Provisioning Profiles list under DEVELOPMENT.

Adding the iTunes Artwork to Your Application

Test versions of your application should contain artwork iTunes uses to identify your application. Otherwise, when users add your application to their iTunes library, iTunes uses generic artwork for it, as shown in Figure 9-2.

Figure 9-2 Generic iTunes artwork for test applications



The iTunes artwork your testers see should be your application's icon. This artwork must be a 512 x 512 JPEG or PNG file named `iTunesArtwork`. Note that the file must not have an extension.

After generating the file of your application's icon, follow these steps to add it to your application:

1. Open your project in Xcode.
2. In the Groups & Files list, select the Resources group.
3. Choose Project > Add to Project, navigate to your `iTunesArtwork` file, and click Add.
4. In the dialog that appears, select the "Copy items" option and click Add.

Archiving Your Application for Testing

To archive your application for distribution to your testers:

1. Set the Code Signing Identity build setting to a test provisioning profile.

2. In the Project window Overview toolbar menu, set Active Executable to a device.
3. Choose Build > Build and Archive.

Note: To streamline this process, create a distribution configuration (by duplicating the Release configuration), and change the Code Signing Identity build setting in that configuration. That way, you can keep your development and distribution configurations separate. For more information about build configurations, see "Build Configurations".

Sending Your Application to Testers

To send your application to testers:

1. In the Archived Applications list in the Organizer, select the application archive you want to send to testers, and click Share Application.
2. Click E-Mail.

   In the email message, provide your testers information they need to test your application.

Importing Crash Logs from Testers

To add tester crash logs to the Organizer to view their symbol information, drag the crash logs to the Crash Logs group under the DEVELOPMENT section.

Important: For Xcode to symbolize crash logs (to add information about the API used to the crash log), the volume containing your archived applications and their corresponding dSYM files must be indexed by Spotlight.

Instructions for Application Testers

This section provides instructions to testers about the procedures to follow to test your iOS applications on their devices. An application tester is a potential user of your application who is willing to test it before it's released through the App Store.

You may send these instructions, along with any special tasks needed to test your application, to customers interested in testing your application.

Sending Your Device ID to a Developer

Before a developer can send you an application for testing, they must register your device with Apple under their application-testing program.

To send your device ID to a developer for test-program registration:

1. Launch iTunes.

2. Connect your device to your computer.
3. Select the device in the Devices list.
4. In the Summary pane, click the Serial Number label. It changes to Identifier.
5. Choose Edit > Copy.
6. Email your device identifier to the developer. Be sure to include your name and device name in the email.

Installing an Application for Testing

After being registered in a developer's testing program, the developer sends you an archive of the test application. You need to install the archive into iTunes to run the application on your device.

To install the test application on your device:

1. In the Finder, double-click the application archive, `<Application_Name>.ipa`.

   The application appears in the iTunes Applications list.
2. Sync your device.

   If the version of iOS on your device is earlier than the test application can run on, you need to update your device with the current release of iOS.

Sending Crash Reports to a Developer

When the application you're testing crashes, iOS creates a record of that event. The next time you connect your device to iTunes, iTunes downloads those records (known as crash logs) to your computer. To help get the problem fixed, you should send crash logs of the application you're testing to its developer.

Sending Crash Reports from Macs

To send crash logs to developers:

1. In the Finder, open a new window.
2. Choose Go > Go to Folder.
3. Enter `~/Library/Logs/CrashReporter/MobileDevice`.
4. Open the folder named after your device's name.
5. Select the crash logs named after the application you're testing.
6. Choose Finder > Services > Mail > Send File.
7. In the New Message window, enter the developer's email address in the To field and `<application_name> crash logs from <your_name>` (for example, `MyTestApp crash logs from Anna Haro`) in the Subject field.
8. Choose Message > Send.

9. In the Finder, you may delete the crash logs you sent to avoid sending duplicate reports later.

Sending Crash Reports from Windows

To send crash logs to developers, enter the crash log directory (Listing 9-1 and Listing 9-2) in the Windows search field, replacing `<user_name>` with your Windows user name.

Listing 9-1 Crash log storage on Windows Vista

```
C:\Users\<user_name>\AppData\Roaming\Apple
computer\Logs\CrashReporter/MobileDevice
```

Listing 9-2 Crash log storage on Windows XP

```
C:\Documents and Settings\<user_name>\Application Data\Apple
computer\Logs\CrashReporter
```

Open the folder named after your device's name and send the crash logs for the application you're testing in an email message using the subject-text format `<application_name> crash logs from <your_name>` (for example, `MyTestApp crash logs from Anna Haro`) to the application's developer.

Publishing Your Application for Distribution

When you're ready to publish your application for general distribution through the App Store, you submit it to iTunes Connect. This section describes how to prepare your application for submission and how to submit it to iTunes Connect.

Creating a Distribution Profile for Your Application

To create a distribution profile for your application:

1. Create a distribution provisioning profile in the Portal with these characteristics:

Distribution method	App Store
Profile name	<Application_Name> Distribution Profile
App ID	The appropriate application ID for your application.

2. Download the distribution profile and install it in the Xcode Organizer.

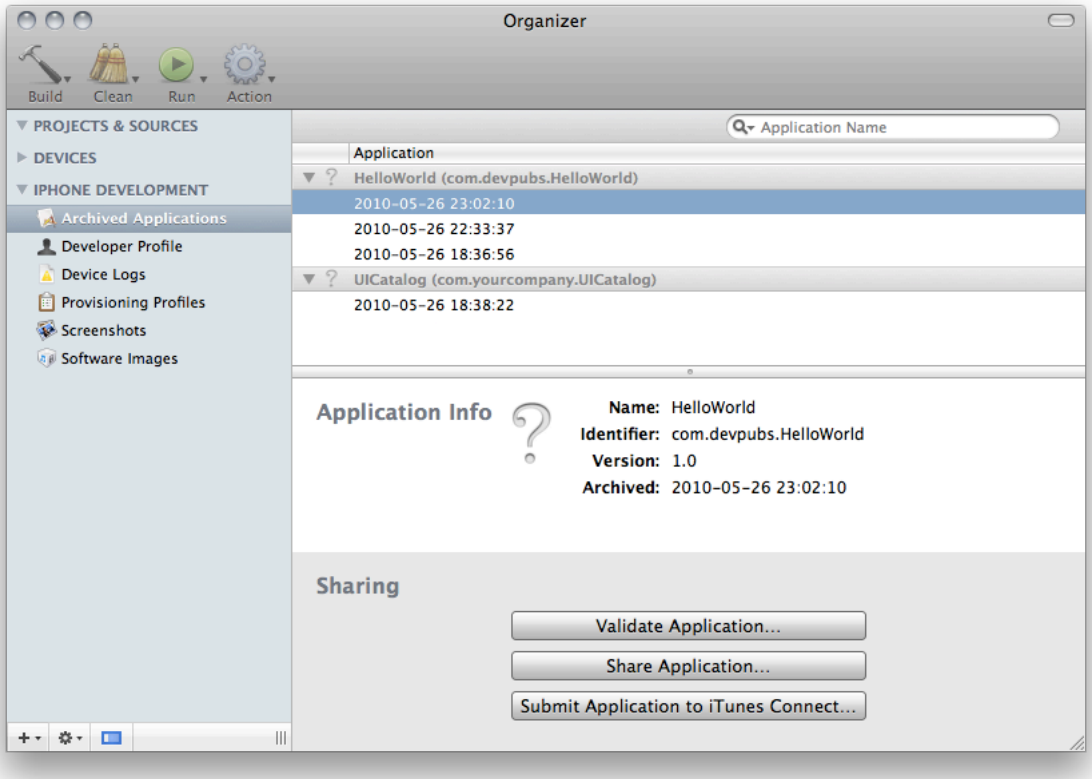
   Drag the `<Profile_Name>.mobileprovision` file to the Provisioning Profiles list under DEVELOPMENT in the Organizer.

Archiving Your Application for Submission to iTunes Connect

To create a distribution archive of your application:

1. Set the Code Signing Identity build setting to the distribution profile.
2. In the Overview toolbar menu in the Project window, set Active Executable to a device.
3. Choose Build > Build and Archive.

   Your new application archive appears in the Archived Applications list in the Organizer. Each archive is identified with the date and time it was created.



Submitting Your Application to iTunes Connect

To submit your application to iTunes Connect:

1. In the Archived Applications list in the Organizer, select the application archive you want to submit, and click Submit Application to iTunes Connect.
2. In the dialog that appears, enter your iTunes Connect credentials, and click Submit.

iOS Development FAQ

Here are some common questions developers ask about iOS development.

Frequently Asked Questions

- How do I fix a "missing SDK" problem?

  The reason Xcode says that an SDK is missing is that a particular iOS SDK release is not part of the iOS SDK distribution you are using.

  Set the Base SDK build setting for the project or target to Latest iOS, as described in "Setting the Base SDK" (page 34).

- How do I submit applications to iTunes Connect for distribution through the App Store.

  Use the Build and Archive command. See "Distributing Applications" (page 77).

- Does the iOS Simulator application run on network home directories?

  No.

- Do Objective-C properties need to be backed up by instance variables or accessor methods for them to work?

  Yes.

- Do static libraries need to be code-signed before being used in an iOS application?

  No.

- Why is my application having problems processing PNG files?

  The code that is trying to use your PNG files may not understand compressed PNG files.

  Turn off the Compress PNG Files build setting. For information about build settings, see "Editing Build Settings" in *Xcode Project Management Guide*.

- Can I develop iOS applications on Windows?

  No. iOS applications can be developed only on Mac OS X.

- How do I link all the Objective-C classes in a static library?

  Set the Other Linker Flags build setting to `-ObjC`. If that doesn't bring in all the classes, set it to `-all_load`.

- When should I replace deprecated API?

Update as soon as you can, considering the iOS versions you want your application to run on. See *SDK Compatibility Guide* for details.

- Can iOS Simulator use my computer's camera?

  No.

- What are the minimum hardware requirements for iOS development?

  A Mac with an Intel processor.

Hello, World! Source Code

This appendix contains the source code for the Hello, World! application described in "Tutorial: Hello, World!" (page 19).

Listing A-1 main.m

```
// main.m
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {

   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int retVal = UIApplicationMain(argc, argv, nil, nil);
   [pool release];
   return retVal;
}
```

Listing A-2 HelloWorldAppDelegate.h

```
// HelloWorldAppDelegate.h
#import <UIKit/UIKit.h>

@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@end
```

Listing A-3 HelloWorldAppDelegate.m

```
// HelloWorldAppDelegate.m
#import "HelloWorldAppDelegate.h"
#import "MyView.h"

@implementation HelloWorldAppDelegate

@synthesize window;

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

   // Override point for customization after application launch.
   MyView *view = [[MyView alloc] initWithFrame:[window frame]];
   [window addSubview:view];
   [view release];
   [window makeKeyAndVisible];
   return YES;
}
```

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

@end
```

Listing A-4 MyView.h

```
// MyView.h
#import <UIKit/UIKit.h>

@interface MyView : UIView {
}

@end
```

Listing A-5 MyView.m

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

@implementation MyView

- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
        // Initialization code
    }
    return self;
}

- (void)drawRect:(CGRect)rect {
   NSString *hello   = @"Hello, World!";
   CGPoint  location = CGPointMake(10, 20);
   UIFont   *font    = [UIFont systemFontOfSize:24];
   [[UIColor whiteColor] set];
   [hello drawAtPoint:location withFont:font];
}

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

@end
```

Unit-Test Result Macro Reference

The SenTestingKit framework defines a set of test-case result macros that allow you to report the test-case results to the framework. When a test fails, the framework emits a test-failure message, which Xcode displays in text-editor, Build Results, or console windows, depending on the type of unit tests being performed.

The following sections describe the test-result macros you can use in your test-case methods. These macros are declared in `SenTestCase.h`.

Important: When the expressions your test cases evaluate throw exceptions, they produce unknown errors. To test whether your code raises exceptions, create test cases that explicitly check for the presence or absence or exceptions; see "Exception Tests" (page 93).

Unconditional Failure

STFail

Fails the test case.

```
STFail(failure_description, ...)
```

Parameters

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Equality Tests

STAssertEqualObjects

Fails the test case when two objects are different.

```
STAssertEqualObjects(object_1, object_2, failure_description, ...)
```

Parameters

`object_1`

An object.

`object_2`

An object.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails when `[object_1 isEqualTo:object_2]` is false.

STAssertEquals

Fails the test case when two values are different.

```
STAssertEquals(value_1, value_2, failure_description, ...)
```

Parameters

`value_1`

A scalar, structure, or union.

`value_2`

A scalar, structure, or union.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails when `value_1` is not equal to `value_2`.

STAssertEqualsWithAccuracy

Fails the test case when the difference between two values is greater than a given value.

```
STAssertEqualsWithAccuracy(value_1, value_2, accuracy, failure_description, ...)
```

Parameters

`value_1`

An integer or a floating-point value.

`value_2`

An integer or a floating-point value.

`accuracy`

An integer or a floating-point value.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails when the difference between `value_1` and `value_2` is greater than `accuracy`.

Nil Tests

STAssertNil

Fails the test case when a given expression is not nil.

```
STAssertNil(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

STAssertNotNil

Fails the test case when a given expression is nil.

```
STAssertNotNil(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Boolean Tests

STAssertTrue

Fails the test case when a given expression is false.

```
STAssertTrue(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

STAssertFalse

Fails the test case when a given expression is true.

```
STAssertFalse(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Exception Tests

STAssertThrows

Fails the test case when an expression doesn't raise an exception.

```
STAssertThrows(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

STAssertThrowsSpecific

Fails the test case when an expression doesn't raise an exception of a particular class.

```
STAssertThrowsSpecific(expression, exception_class, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`exception_class`

An exception class.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails when `expression` doesn't raise an exception of the class `exception_class`.

STAssertThrowsSpecificNamed

Fails the test case when an expression doesn't raise an exception of a particular class with a given name.

```
STAssertThrowsSpecificNamed(expression, exception_class, exception_name,
failure_description, ...)
```

Parameters

`expression`

Expression to test.

`exception_class`

An exception class.

`exception_name`

A string with the name of an exception.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails when `expression` doesn't raise an exception of the class `exception_class` with the name `exception_name`.

STAssertNoThrow

Fails the test case when an expression raises an exception.

```
STAssertNoThrow(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

STAssertNoThrowSpecific

Fails the test case when an expression raises an exception of a particular class.

```
STAssertNoThrowSpecific(expression, exception_class, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`exception_class`

An exception class.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails `expression` raises an exception of the class `exception_class`.

STAssertNoThrowSpecificNamed

Fails the test case when an expression doesn't raise an exception of a particular class with a given name.

```
STAssertNoThrowSpecificNamed(expression, exception_class, exception_name,
failure_description, ...)
```

Parameters

`expression`

Expression to test.

`exception_class`

An exception class.

`exception_name`

A string with the name of an exception.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Detail

The test fails when the `expression` raises an exception of the class `exception_class` with the name `exception_name`.

STAssertTrueNoThrow

Fails the test case when an expression is false or raises an exception.

```
STAssertTrueNoThrow(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

STAssertFalseNoThrow

Fails the test case when an expression is true or raises an exception.

```
STAssertFalseNoThrow(expression, failure_description, ...)
```

Parameters

`expression`

Expression to test.

`failure_description`

Format string specifying error message. Can be `nil`.

`...`

(Optional) A comma-separated list of arguments to substitute into `failure_description`.

Glossary

active build configuration The build configuration used to build a product. See also **build configuration**.

active SDK The SDK used to build a product and the runtime environment on which the product is to run. See also **SDK family**.

application ID A string that identifies an iOS application or a set of iOS applications from one vendor. They are similar to bundle identifiers. This is an example application ID: `GFWOTNXFIY.com.mycompany.MyApp`.

base SDK Project setting that specifies the default SDK to use when building the project's targets. Targets can override this setting with the iOS OS Deployment Target build setting.

build configuration A named collection of build settings that build one or more products in a project in different for specific purposes—for example, for debugging or for release.

certificate signing request (CSR) File that contains personal information used to generate a development certificate. Certificate signing requests are created by the Keychain Access application.

code completion A shortcut that automatically suggests likely completions as you type an identifier or a keyword. The suggestions are based on the text you type and the surrounding context within the file.

development certificate File that identifies an iOS application developer. Xcode uses development certificates to sign application binaries.

device family Type of device in which iOS can run. There two device families: iPhone, and iPad.

entitlement A property that allows an application to access a protected iOS feature or capability.

instrument A data-gathering agent developed using the Instruments application. Instruments collect performance information about an application or an entire system.

Instruments application A performance analysis tool used to gather and mine application-performance data.

iOS Dev Center An Apple developer center that provides all the resources needed to develop iOS applications. Access to this developer center requires an ADC membership. See also **Apple Developer Connection**.

iOS Developer Program A program that allows you to develop iOS applications, test them on devices, and distribute them to your customers through the App Store.

iOS Provisioning Portal A restricted-access area of the iOS Dev Center that allows you to configure devices to test your iOS applications

iOS Simulator application An application that simulates the iOS runtime environment and user experience in Mac OS X for testing iOS applications in early stages of development.

project window A window that displays and organizes the files that make up an Xcode project.

provisioning profile A file that allows applications in development to be installed on an iOS-based device. It contains one or more development certificates, an application ID, and one or more device IDs

SDK family Group of SDK releases used to build software products for a particular Apple platform. The available SDK families are iOS Device SDK, iOS Simulator SDK, and Mac OS X SDK.

test case A piece of code that executes test-subject code to verify that it behaves as expected.

test-case method An instance method of a unit-test class named `test...` that exercises API to test and reports whether it produced the expected results.

test provisioning profile A provisioning profile issued to users not on an iOS application developer team. It allows them to install and test applications that have not been published to the App Store. Also known as **ad-hoc provisioning profile**.

test suite A set of test cases. See also **test case**.

Xcode A set of tools and resources used to develop Cocoa and Cocoa Touch applications.

Xcode application The main application of the Xcode integrated development environment (IDE). It manages the other applications that are part of Xcode and provides the main user interface used to develop software products.

Document Revision History

This table describes the changes to *iOS Development Guide*.

Date	Notes
2010-11-15	Documented changes to Base SDK build setting.
	Described Latest iOS value for Base SDK and added instructions for solving "Missing Base SDK" problem in "Setting the Base SDK" (page 34).
	Added instructions on moving the center of a pinch in iOS Simulator in "Performing Gestures" (page 46).
	Updated content for the workflows and requirements of the iOS SDK 4.2 distribution.
2010-08-26	Made minor corrections.
2010-07-02	Changed the title from iPhone Development Guide. Updated Hello, World! tutorial to iPhone SDK 4.0.
	Updated Hello, World! tutorial and source code ("Tutorial: Hello, World!" (page 19) and "Hello, World! Source Code" (page 87)) for iOS SDK 4.0 toolset.
2010-05-28	Added information about automatic provisioning profile management, application archiving, and application distribution.
	Updated "Building and Running Applications" (page 33) with details about using the Base SDK and iPhone OS Deployment Target build setting and the Overview toolbar menu in the Project window.
	Updated "Using iOS Simulator" (page 45) with information about how the Objective-C–runtime change in iOS 4.0 affects existing iOS Simulator binaries.
	Updated "Managing Devices and Digital Identities" (page 49) to describe automatic provisioning-profile management and how to manage developer profiles in the Xcode Organizer.
	Updated "Distributing Applications" (page 77) with details about the use of the Build and Archive command and recommended workflow for distributing applications for testing.
	Added hardware-simulation support information to "Using iOS Simulator" (page 45).
2010-03-19	Added iPad information.

Date	Notes
	Added "Setting the Device Family" (page 37) to describe how to specify the family of devices on which you want your application to run.
	Added "Upgrading a Target from iPhone to iPad" (page 32) to explain how to upgrade an iPhone target for building iPad applications.
	Updated "Using iOS Simulator" (page 45) with iPad information.
2010-01-20	Fixed typos and addressed feedback.
2009-08-06	Added information about editing property-list files, linking static libraries, and iOS Simulator versions. Made minor changes.
	Added "Editing Property-List Files" (page 27).
	Added important information about debugging applications on devices to "Debug Facilities Overview" (page 57).
	Added "Setting the Simulation-Environment Device Family and iOS Version" (page 45).
2009-05-28	Described how to set the architecture for which an application is built.
	Added "Setting the Architecture" (page 36) to describe how to choose the architectures an application can be built for.
2009-05-14	Added unit-testing information and streamlined build workflow.
	Added "Unit Testing Applications" (page 63).
	Added "iOS Simulator Frameworks and Libraries".
	Updated "Building and Running Applications" (page 33) with streamlined build workflow. Added information about restoring application data from a backup.
2009-04-08	Added information on creating entitlement property-list files. Made minor content reorganization.
	Added "Managing Application Entitlements" (page 30).
2009-03-04	Made minor content changes.
2009-02-04	Made minor content changes.
	Updated "Capturing Screen Shots" (page 55) to specify how to get PNG files of screen shots captured in the Organizer.
	Updated "Building Your Application" (page 39) with application ID build-error information.
2009-01-06	Made minor content additions.
	Explained that iOS Simulator binaries can be used on only one release of the simulator.

Date	Notes
2008-11-14	Added information about new iOS Simulator features.
	Added "Adding the iTunes Artwork to Your Application" (page 79).
	Added information about the Simulate Memory Warning and Toggle In-Call Status Bar commands to "Manipulating the Hardware" (page 46).
	Added "Core Location Functionality" (page 48).
	Added information about using static libraries in iOS applications to "Creating an iOS Application Project" (page 14).
2008-10-15	New document that describes how to develop iPhone applications using Xcode.
	Incorporates content previously published in *iPhone OS Programming Guide* and *iOS Simulator Programming Guide*.

Document Revision History