Preferences and Settings Programming Guide

Developer

Contents

Figures, Tables, and Listings

About Preferences and Settings

Preferences are pieces of information that you store persistently and use to configure your app. Apps often expose preferences to users so that they can customize the appearance and behavior of the app. Most preferences are stored locally using the Cocoa preferences system—known as the **user defaults system**. Apps can also store preferences in a user's iCloud account using the key-value store.

The user defaults system and key-value store are both designed for storing simple data types—strings, numbers, dates, Boolean values, URLs, data objects, and so forth—in a property list. The use of a property list also means you can organize your preference data using array and dictionary types. It is also possible to store other objects in a property list by encoding them into an `NSData` object first.

At a Glance

Apps integrate preferences in several ways, including programmatically at various points throughout your code and as part of the user interface. Preferences are supported in both iOS and Mac apps.

You Decide What Preferences You Want to Expose

Preferences are different for each app, and it is up to you to decide what parts of your app you want to make configurable. Configuration involves checking the value of a stored preference from your code and taking action based on that value. Thus, the preference value itself should always be simple and have a specific meaning that is then implemented by your app.

Relevant section "What Makes a Good Preference?" (page 8)

Apps Provide Their Own Preferences Interface

Because each app's preferences are different, the app itself is responsible for deciding how best to present those preferences to the user, if at all. Both iOS and OS X provide some standard places for you to incorporate a preferences interface, but you are still responsible for designing that interface and displaying it at the appropriate time.

Relevant section "Providing a Preference Interface" (page 8)

Apps Access Preferences Using the User Defaults Object

Apps access locally stored preferences using a user defaults object, which is either an `NSUserDefaults` object (iOS and OS X) or an `NSUserDefaultsController` object (OS X only). In addition to retrieving preference values, apps can use this object to register default values for preferences and manage other aspects of the preferences system.

Relevant chapter "Accessing Preference Values" (page 13)

iCloud Stores Shared Preference and Configuration Data

Apps that support iCloud can put some of their preference data in the user's iCloud account and make it available to instances of the app running on the user's other devices. You use this capability to supplement (not replace) your app's existing preferences data and provide a more coherent experience across the user's devices. For example, a magazine app might store information about the page number and issue last read by the user so that the app running on a different device can show that same page.

Relevant chapter "Storing Preferences in iCloud" (page 19)

Defaults Are Grouped into Domains in OS X

OS X preferences are grouped by domains so that system preferences can be differentiated from app preferences. Splitting preferences in this manner lets the user specify some preferences globally and then override one or more of those preferences inside an app.

Relevant section "The Organization of Preferences" (page 9)

A Settings Bundle Manages Preferences for iOS Apps

An iOS, apps can display preferences from the Settings app, which is a good place to put preferences that the user does not need to configure frequently. To display preferences in the Settings app, an app's bundle must include a special resource called a *Settings bundle* that defines the preferences to display, the proper way to display them, and the information needed to record the user's selections.

Note Apps are not required to use a Settings bundle to manage all preferences. For preferences that the user is likely to change frequently, the app can display its own custom interface for managing those preferences.

Relevant chapter "Implementing an iOS Settings Bundle" (page 24)

See Also

For information about property lists, see *Property List Programming Guide*.

For more advanced information about using Core Foundation to manage preferences, see *Preferences Programming Topics for Core Foundation*.

About the User Defaults System

The user defaults system manages the storage of preferences for each user. Most preferences are stored persistently and therefore do not change between subsequent launch cycles of your app. Apps use preferences to track user-initiated and program-initiated configuration changes.

What Makes a Good Preference?

When defining your app's preferences, it is better to use simple values and data types whenever possible. The preferences system is built around property-list data types such as strings, numbers, and dates. Although you can use an `NSData` object to store arbitrary objects in preferences, doing so is not recommended in most cases.

Storing objects persistently means that your app has to decode that object at some point. In the case of preferences, a stored object means decoding the object every time you access the preference. It also means that a newer version of your app has to ensure that it is able to decode objects created and written to disk using an earlier version of your app, which is potentially error prone.

A better approach for preferences is to store simple strings and values and use them to create the objects your app needs. Storing simple values means that your app can always access the value. The only thing that changes from release to release is the interpretation of the simple value and the objects your app creates in response.

Providing a Preference Interface

For user-facing preferences, Table 1-1 lists the options for displaying those preferences to the user. As you can see from this table, most options involve the creation of a custom user interface for managing and presenting preferences. If you are creating an iOS app, you can use a Settings bundle to present preferences, but you should do so only for settings the user changes infrequently.

Table 1-1 Options for displaying preferences to the user

Preference	iOS	OS X
Frequently changed preferences	Custom UI	Custom UI
Infrequently changed preferences	Settings bundle	Custom UI

> **Note** An example of preferences that might change frequently include things like the volume levels or control options of a game. An example of preferences that might change infrequently are the email address and server settings in the Mail app. For iOS apps, it is ultimately up to you to decide whether it is appropriate to expose preferences from the Settings app or from inside your app.

Preferences in Mac apps should be accessible from a Preferences menu item in the app menu. Cocoa apps created using the Xcode templates provide such a menu item for you automatically. It is your responsibility to present an appropriate user interface when the user chooses this menu item. You can provide that user interface by defining an action method in your app delegate that displays a custom preferences window and connecting that action method to the menu item in Interface Builder.

There is no standard way to display custom preferences from inside an iOS app. You can integrate preferences in many ways, including using a separate tab in a tab-bar interface or using a custom button from one of your app's screens. Preferences should generally be presented using a distinct view controller so that changes in preferences can be recorded when that view controller is dismissed by the user.

The Organization of Preferences

Preferences are grouped into domains, each of which has a name and a specific usage. For example, there's a domain for app-specific preferences and another for systemwide preferences that apply to all apps. All preferences are stored and accessed on a per-user basis. There is no support for sharing preferences between users.

Each preference has three components:

- The domain in which it is stored
- Its name (specified as an `NSString` object)
- Its value, which can be any property-list object (`NSData`, `NSString`, `NSNumber`, `NSDate`, `NSArray`, or `NSDictionary`)

The lifetime of a preference depends on which domain you store it in. Some domains store preferences persistently by writing them to the user's defaults database. Such preferences continue to exist from one app launch to the next. Other domains store preferences in a more volatile way, preserving preference values only for the life of the corresponding user defaults object.

A search for the value of a given preference proceeds through the domains in an `NSUserDefaults` object's search list. Only domains in the search list are searched and they are searched in the order shown in Table 1-2, starting with the `NSArgumentDomain` domain. A search ends when a preference with the specified name is found. If multiple domains contain the same preference, the value is taken from the domain nearest the beginning of the search list.

Table 1-2 Search order for domains

Domain	State
`NSArgumentDomain`	volatile
Application (Identified by the app's identifier)	persistent
`NSGlobalDomain`	persistent
Languages (Identified by the language names)	volatile
`NSRegistrationDomain`	volatile

The Argument Domain

The argument domain comprises values set from command- line arguments (if you started the app from the command line) and is identified by the `NSArgumentDomain` constant. Values set from the command line are automatically placed into this domain by the system. To add a value to this domain, specify the preference name on the command line (preceded with a hyphen) and follow it with the corresponding value. For example, the following command launches Xcode and sets the value of its `IndexOnOpen` preference to `NO`:

```
localhost> Xcode.app/Contents/MacOS/Xcode -IndexOnOpen NO
```

Preferences set from the command line temporarily override the established values stored in the user's defaults database. In the preceding example, setting the `IndexOnOpen` preference to `NO` prevents Xcode from indexing projects automatically, even if the preference is set to `YES` in the user defaults database.

The Application Domain

The application domain contains app-specific preferences that are stored in the user defaults database of the current user. When you use the shared `NSUserDefaults` object (or a `NSUserDefaultsController` object in OS X) to write preferences, those preferences are automatically placed in this domain.

Because this domain is app-specific, the contents of the domain are tied to your app's bundle identifier. The contents of this domain are stored in a file that is managed by the system. Currently, this file is located in the *$HOME*`/Library/Preferences/` directory, where *$HOME* is either the app's home directory or the user's home directory (depending on the platform and whether your app is in a sandbox). The name of the user defaults database file is <*ApplicationBundleIdentifer*>`.plist`, where <*ApplicationBundleIdentifer*> is your app's bundle identifier. You should not modify this file directly but can inspect it during debugging to make sure preference values are being written by your app.

The Global Domain

The global domain contains preferences that are applicable to all apps and is identified by the `NSGlobalDomain` constant. This domain is typically used by system frameworks to store system-wide values and should not be used by your app to store app-specific values. If you want to change the value of a preference in the global domain, write that same preference to the application domain with the new value.

Examples of how the system frameworks use this domain:

- Instances of the `NSRuleView` class store the user's preferred measurement units in the `AppleMeasurementUnits` key. Using this storage location causes ruler views in all apps to use the same units.
- The system uses the `AppleLanguages` key to store the user's preferred languages as an array of strings. For example, a user could specify English as the preferred language, followed by Spanish, French, German, Italian, and Swedish.

The Languages Domains

For each language in the `AppleLanguages` preference, the system records language-specific preference values in a domain whose name is based on the language name. Each language-specific domain contains preferences for the corresponding locale. Many classes in the Foundation framework (such as the `NSDate`, `NSDateFormatter`, `NSTimeZone`, `NSString`, and `NSScanner` classes) use this locale information to modify their behavior. For example, when you request a string representation of an `NSCalendarDate` object, the `NSCalendarDate` object uses the locale information to find the names of months and the days of the week for the user's preferred language.

The Registration Domain

The registration domain defines the set of default values to use if a given preference is not set explicitly in one of the other domains. At launch time, an app can call the `registerDefaults:` method of `NSUserDefaults` to specify a default set of values for important preferences. When an app launches for the first time, most preferences have no values, so retrieving them would yield undefined results. Registering a set of default values ensures that your app always has a known good set of values to operate on.

The contents of the registration domain can be set only by using the `registerDefaults:` method.

Viewing Preferences Using the Defaults Tool

In OS X, the `defaults` command-line tool provides a way for you to examine the contents of the user defaults database. During app development, you might use this tool to validate the preferences your app is writing to disk. To do that, you would use a command of the following form from the Terminal app:

```
defaults read <application-bundle-identifier>
```

To read the contents of the global domain, you would use the following command:

```
defaults read NSGlobalDomain
```

For more information about using the defaults tool to read and write preference values, see `defaults` man page.

Accessing Preference Values

You use the `NSUserDefaults` class to gain access to your app's preferences. Each app is provided with a single instance of this class, accessible from the `standardUserDefaults` class method. You use the shared user defaults object to:

- Specify any default values for your app's preferences at launch time.
- Get and set individual preference values stored in the app domain.
- Remove preference values.
- Examine the contents of the volatile preference domains.

Mac apps that use Cocoa bindings can use an `NSUserDefaultsController` object to set and get preferences automatically. You typically add such an object to the same nib file you use for displaying user-facing preferences. You bind your user interface controls to items in the user defaults controller, which handles the process of getting and setting values in the user defaults database.

Preference values must be one of the standard property list object types: `NSData`, `NSString`, `NSNumber`, `NSDate`, `NSArray`, or `NSDictionary`. The `NSUserDefaults` class also provides built-in manipulations for storing `NSURL` objects as preference values. For more information about property lists and their contents, see *Property List Programming Guide* .

Registering Your App's Default Preferences

At launch time, an app should register default values for any preferences that it expects to be present and valid. When you request the value of a preference that has never been set, the methods of the `NSUserDefaults` class return default values that are appropriate for the data type. For numerical scalar values, this typically means returning `0`, but for strings and other objects it means returning `nil`. If these standard default values are not appropriate for your app, you can register your own default values using the `registerDefaults:` method. This method places your custom default values in the `NSRegistrationDomain` domain, which causes them to be returned when a preference is not explicitly set.

When calling the `registerDefaults:` method, you must provide a dictionary of all the default values you need to register. Listing 2-1 shows an example where an iOS app registers its default values early in the launch cycle. You can register default values at any time, of course, but should always register them before attempting to retrieve any preference values.

Listing 2-1 Registering default preference values

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

   // Register the preference defaults early.

    NSDictionary *appDefaults = [NSDictionary

        dictionaryWithObject:[NSNumber numberWithBool:YES]
forKey:@"CacheDataAgressively"];

    [[NSUserDefaults standardUserDefaults] registerDefaults:appDefaults];


   // Other initialization...

}
```

When registering default values for scalar types, use an `NSNumber` object to specify the value for the number. If you want to register a preference whose value is a URL, use the `archivedDataWithRootObject:` method of `NSKeyedArchiver` to encode the URL in an `NSData` object first. Although you can use a similar technique for other types of objects, you should avoid doing so when a simpler option is available.

Getting and Setting Preference Values

You get and set preference values using the methods of the `NSUserDefaults` class. This class has methods for getting and setting preferences with scalar values of type Boolean, `integer`, `float`, and `double`. It also has methods for getting and setting preferences whose value is an object of type `NSData`, `NSDate`, `NSString`, `NSNumber`, `NSArray`, `NSDictionary`, and `NSURL`. There are two situations where you might get preference values and one where you might set them:

- Get preference values:
    - When you need to use the value to configure your app's behavior.
    - When you need to display the value in your preferences interface.
- Set preference values when the user changes them in your preferences interface.

The following code shows how you might get a preference value in your code. In this example, the code retrieves the value of the `CacheDataAggressively` key, which is custom key that the app might use to determine its caching strategy. Code like this can be used anywhere to handle custom configuration of your app. If you wanted to display this particular preference value to the user, you would use similar code to configure the controls of your preferences interface.

```
if ([[NSUserDefaults standardUserDefaults] boolForKey:@"CacheDataAggressively"])
{
   // Delete the backup file.
}
```

To set a preference value programmatically, you call the corresponding setter methods of `NSUserDefaults`. When setting object values, you must use the `setObject:forKey:` method. When calling this method, you must make sure that the object is one of the standard property list types. The following example sets some preferences based on the state of the app's preferences interface.

```
NSUserDefaults* defaults = [NSUserDefaults standardUserDefaults];
if ([cacheAgressivelyButton state] == NSOnState) {
   // The user wants to cache files aggressively.
   [defaults setBool:YES forKey:@"CacheDataAggressively"];
   [defaults setObject:[NSDate dateWithTimeIntervalSinceNow:(3600 * 24 * 7)]
             forKey:@"CacheExpirationDate"]; // Set a 1-week expiration
} else {
    // The user wants to use lazy caching.
   [defaults setBool:NO forKey:@"CacheDataAggressively"];
   [defaults removeObjectForKey:@"CacheExpirationDate"];
}
```

You do not have to display a preferences interface to manage all values. Your app can use preferences to cache interesting information. For example, `NSWindow` objects store their current location in the user defaults system. This data allows them to return to the same location the next time the user starts the app.

Synchronizing and Detecting Preference Changes

Because the `NSUserDefaults` class caches values, it is sometimes necessary to synchronize the cached values with the current contents of the user defaults database. Your app is not always the only entity modifying the user defaults database. In iOS, the Settings app can modify the values of preferences for apps that have a Settings bundle. In OS X, the system and other apps might modify preferences values in response to user actions. For example, if the user changes preferred languages, the system writes the new values to the user defaults database. In Mac OS X v10.5 and later, the shared `NSUserDefaults` object synchronizes its caches automatically at periodic intervals. However, apps can call the `synchronize` method manually to force an update of the cached values.

To detect when changes to a preference value occur, apps can also register for the notification `NSUserDefaultsDidChangeNotification`. The shared `NSUserDefaults` object sends this notification to your app whenever it detects a change to a preference located in one of the persistent domains. You can use this notification to respond to changes that might impact your user interface. For example, you could use it to detect changes to the user's preferred language and update your app content appropriately.

Managing Preferences Using Cocoa Bindings

Mac apps can use Cocoa bindings to set preference values directly from their user interfaces. Modifying preferences using bindings involves adding an `NSUserDefaultsController` object to the appropriate nib files and binding the values of your controls to the preference values in the user defaults database. When your app shows the interface, the user defaults controller automatically loads values from the user defaults database and uses them to set the value of controls. Similarly, when the user changes the value in a control, the user defaults controller updates the value in the user defaults database.

For more information on how to use the `NSUserDefaultsController` class to bind preference values to your user interface, see "User Defaults and Bindings" in *Cocoa Bindings Programming Topics*.

Managing Preferences Using Core Foundation

The Core Foundation framework provides its own set of interfaces for accessing preferences stored in the user defaults database. Like the `NSUserDefaults` class, you can use Core Foundation functions to get and set preference values and synchronize the user defaults database. Unlike `NSUserDefaults`, you can use the Core Foundation functions to write preferences for different apps and on different computers. Note that modifying some preferences domains (those not belonging to the current app and user) requires root privileges (or admin privileges prior to OS X v10.6); for information on how to gain suitable privileges, see *Authorization Services Programming Guide*. Writing outside the app domain is not possible for apps installed in a sandbox.

For information about the Core Foundation functions for getting and setting preferences, see *Preferences Utilities Reference*.

Setting a Preference Value Using Core Foundation

Preferences are stored as key-value pairs. The key must be a CFString object, but the value can be any Core Foundation property list value (see *Property List Programming Topics for Core Foundation*), including the container types. For example, you might have a key called `defaultWindowWidth` that defines the width in

pixels of any new windows that your app creates. Its value would most likely be of type CFNumber. You might also decide to combine window width and height into a single preference called `defaultWindowSize` and make its value be a CFArray object containing two CFNumber objects.

The code in Listing 2-2 demonstrates how to create a simple preference for the app MyTextEditor. The example sets the default text color for the app to blue.

Listing 2-2 Writing a simple default

```
CFStringRef textColorKey = CFSTR("defaultTextColor");
CFStringRef colorBLUE = CFSTR("BLUE");

// Set up the preference.
CFPreferencesSetAppValue(textColorKey, colorBLUE,
        kCFPreferencesCurrentApplication);

// Write out the preference data.
CFPreferencesAppSynchronize(kCFPreferencesCurrentApplication);
```

Notice that `CFPreferencesSetAppValue` by itself is not sufficient to create the new preference. A call to `CFPreferencesAppSynchronize` is required to actually save the value. If you are writing multiple preferences, it is more efficient to sync only once after the last value has been set than to sync after each individual value is set. For example, if you implement a preference pane you might synchronize only when the user presses an OK button. In other cases you might not want to sync at all until the app quits—although note that if the app crashes, all unsaved preferences settings will be lost.

Getting a Preference Value Using Core Foundation

The simplest way to locate and retrieve a preference value is to use the `CFPreferencesCopyAppValue` function. This call searches through the various preference domains in order until it finds the key you have specified. If a preference has been set in a less specific domain—Any Application, for example —its value is retrieved with this call if a more specific version cannot be found. Listing 2-3 shows how to retrieve the text color preference saved in Listing 2-2 (page 17).

Listing 2-3 Reading a simple default

```
CFStringRef textColorKey = CFSTR("defaultTextColor");
CFStringRef textColor;
```

```
// Read the preference.
textColor = (CFStringRef)CFPreferencesCopyAppValue(textColorKey,
        kCFPreferencesCurrentApplication);
// When finished with value, you must release it
// CFRelease(textColor);
```

All values returned from preferences are immutable, even if you have just set the value using a mutable object.

Storing Preferences in iCloud

An app can use the iCloud key-value store to share small amounts of data with other instances of itself on the user's other computers and iOS devices. The key-value store is intended for simple data types like those you might use for preferences. For example, a magazine app might store the current issue and page number being read by the user so that other instances of the app can open to the same page when launched. You should not use this store for large amounts of data or for complex data types.

To use the iCloud key-value store, do the following:

1. In Xcode, configure the `com.apple.developer.ubiquity-kvstore-identifier` entitlement for your app.
2. In your code, create the shared `NSUbiquitousKeyValueStore` object and register for change notifications.
3. Use the methods of `NSUbiquitousKeyValueStore` to get and set values.

Key-value data in iCloud is limited to simple property-list types (strings, numbers, dates, and so on).

Strategies for Using the iCloud Key-Value Store

The key-value store is not intended for storing large amounts of data. It is intended for storing configuration data, preferences, and small amounts of app-related data. To help you decide whether the key-value store is appropriate for your needs, consider the following:

- Each app is limited to 1 MB of total space in the key-value store. (There is also a separate per-key limit of 1 MB and a maximum of 1024 keys are allowed.) Thus, you cannot use the key-value store to share large amounts of data.
- The key-value store supports only property-list types. Property-list types include simple types such as `NSNumber`, `NSString`, and `NSDate` objects. You can also store raw blocks of data in `NSData` objects and arrange all of the types using `NSArray` and `NSDictionary` objects.
- The key-value store is intended for storing data that changes infrequently. If the apps on a device make frequent changes to the key-value store, the system may defer the synchronization of some changes in order to minimize the number of round trips to the server. The more frequently apps make changes, the more likely it is that later changes will be deferred and not show up on other devices right away.

- The key-value store is not a replacement for preferences or other local techniques for saving the same data. The purpose of the key-value store is to share data between apps, but if iCloud is not enabled or is not available on a given device, you still might want to keep a local copy of the data.

If you are using the key-value store to share preferences, one approach is to store the actual values in the user defaults database and synchronize them using the key-value store. (If you do not want to use the preferences system, you could also save the changes in a custom property-list file or some other local storage.) When you change the value of a key locally, write that change to both the user defaults database and to the iCloud key-value store at the same time. To receive changes from external sources, add an observer for the notification `NSUbiquitousKeyValueStoreDidChangeExternallyNotification` and use your handler method to detect which keys changed externally and update the corresponding data in the user defaults database. By doing this, your user defaults database always contains the correct configuration values. The iCloud key-value store simply becomes a mechanism for ensuring that the user defaults database has the most recent changes.

Configuring Your App to Use the Key-Value Store

In order to use of the key-value store, an app must be explicitly configured with the `com.apple.developer.ubiquity-kvstore-identifier` entitlement. You use Xcode to enable this entitlement and specify its value for your app:

1. In your Xcode project, select the target for your app.
2. In the Summary tab, enable the Entitlements option.
3. Specify a value for the iCloud Key-Value Store field.

When you enable entitlements, Xcode automatically fills in a default value for the iCloud Key-Value Store field that is based on the bundle identifier of your app. For most apps, the default value is what you want. However, if your app shares its key-value storage with another app, you must specify the bundle identifier for the other app instead. For example, if you have a lite version of your app, you might want it to use the same key-value store as the paid version.

Enabling the entitlement is all you have to do to use the shared `NSUbiquitousKeyValueStore` object. As long as the entitlement is configured and contains a valid value, the key-value store object writes its data to the appropriate location in the user's iCloud account. If there is a problem attaching to the specified iCloud container, any attempts to read or write key values will fail. To ensure the key-value store is configured properly and accessible, you should execute code similar to the following early in your app's launch cycle:

```
NSUbiquitousKeyValueStore* store = [NSUbiquitousKeyValueStore defaultStore];
[[NSNotificationCenter defaultCenter] addObserver:self
```

```
            selector:@selector(updateKVStoreItems:)
            name:NSUbiquitousKeyValueStoreDidChangeExternallyNotification
            object:store];
[store synchronize];
```

Creating the key-value store object early in your app's launch cycle is recommended because it ensures that your app receives updates from iCloud in a timely manner. The best way to determine if changes have been made to keys and values is to register for the notification `NSUbiquitousKeyValueStoreDidChangeExternallyNotification`. And at launch time, you should call the `synchronize` method manually to detect if any changes were made externally. You do not need to call that method at other times during you app's execution.

For more information about how to configure entitlements for an iOS app, see "Configuring Apps" in *Tools Workflow Guide for iOS*.

Accessing Values in the Key-Value Store

You get and set key-value store values using the methods of the `NSUbiquitousKeyValueStore` class. This class has methods for getting and setting preferences with scalar values of type Boolean, `long long`, and `double`. It also has methods for getting and setting keys whose values are `NSData`, `NSDate`, `NSString`, `NSNumber`, `NSArray`, or `NSDictionary` objects.

If you are using the key-value store as a way to update locally stored preferences, you could use code similar to that in Listing 3-1 to coordinate updates to the user defaults database. This example assumes that you use the same key names and corresponding values in both iCloud and the user defaults database. It also assumes that you previously registered the `updateKVStoreItems:` method as the method to call in response to the notification `NSUbiquitousKeyValueStoreDidChangeExternallyNotification`.

Listing 3-1 Updating local preference values using iCloud

```
- (void)updateKVStoreItems:(NSNotification*)notification {
   // Get the list of keys that changed.
   NSDictionary* userInfo = [notification userInfo];
   NSNumber* reasonForChange = [userInfo
objectForKey:NSUbiquitousKeyValueStoreChangeReasonKey];
   NSInteger reason = -1;


   // If a reason could not be determined, do not update anything.
```

```
    if (!reasonForChange)
        return;

    // Update only for changes from the server.
    reason = [reasonForChange integerValue];
    if ((reason == NSUbiquitousKeyValueStoreServerChange) ||
            (reason == NSUbiquitousKeyValueStoreInitialSyncChange)) {
        // If something is changing externally, get the changes
        // and update the corresponding keys locally.
        NSArray* changedKeys = [userInfo
objectForKey:NSUbiquitousKeyValueStoreChangedKeysKey];
      NSUbiquitousKeyValueStore* store = [NSUbiquitousKeyValueStore defaultStore];
        NSUserDefaults* userDefaults = [NSUserDefaults standardUserDefaults];

        // This loop assumes you are using the same key names in both
        // the user defaults database and the iCloud key-value store
        for (NSString* key in changedKeys) {
            id value = [store objectForKey:key];
            [userDefaults setObject:value forKey:key];
        }
    }
}
```

Defining the Scope of Key-Value Store Changes

Every call to one of the `NSUbiquitousKeyValueStore` methods is treated as a single atomic transaction. When transferring the data for that transaction to iCloud, the whole transaction either fails or succeeds. If it succeeds, all of the keys are written to the store and if it fails no keys are written. There is no partial writing of keys to the store. When a failure occurs, the system also generates a `NSUbiquitousKeyValueStoreDidChangeExternallyNotification` notification that contains the reason for the failure. If you are using the key-value store, you should use that notification to detect possible problems.

If you have a group of keys whose values must all be updated at the same time in order to be valid, save them together in a single transaction. To write multiple keys and values in a single transaction, create an `NSDictionary` object with all of the keys and values. Then write the dictionary object to the key-value store using the `setDictionary:forKey:` method. Writing an entire dictionary of changes ensures that all of the keys are written or none of them are.

Implementing an iOS Settings Bundle

In iOS, the Foundation framework provides the low-level mechanism for storing the preference data. Apps then have two options for presenting preferences:

- Display preferences inside the app.
- Use a Settings bundle to manage preferences from the Settings app.

Which option you choose depends on how you expect users to interact with preferences. The Settings bundle is generally the preferred mechanism for displaying preferences. However, games and other apps that contain configuration options or other frequently accessed preferences might want to present them inside the app instead. Regardless of how you present them, you use the `NSUserDefaults` class to access preference values from your code.

This chapter focuses on the creation of a Settings bundle for your app. A **Settings bundle** contains files that describe the structure and presentation style of your preferences. The Settings app uses this information to create an entry for your app and to display your custom preference pages.

For guidelines on how to manage and present settings and configuration options, see *iOS Human Interface Guidelines*.

The Settings App Interface

The Settings app implements a hierarchical set of pages for navigating app preferences. The main page of the Settings app lists the system and third-party apps whose preferences can be customized. Selecting a third-party app takes the user to the preferences for that app.

Every app with a Settings bundle has at least one page of preferences, referred to as the *main page*. If your app has only a few preferences, the main page may be the only one you need. If the number of preferences gets too large to fit on the main page, however, you can create child pages that link off the main page or other child pages. There is no specific limit to the number of child pages you can create, but you should strive to keep your preferences as simple and easy to navigate as possible.

The contents of each page consists of one or more controls that you configure. Table 4-1 lists the types of controls supported by the Settings app and describes how you might use each type. The table also lists the raw key name stored in the configuration files of your Settings bundle.

Table 4-1 Preference control types

Controltype	Description
Text field	The text field type displays a title (optional) and an editable text field. You can use this type for preferences that require the user to specify a custom string value. The key for this type is `PSTextFieldSpecifier`.
Title	The title type displays a read-only string value. You can use this type to display read-only preference values. (If the preference contains cryptic or nonintuitive values, this type lets you map the possible values to custom strings.) The key for this type is `PSTitleValueSpecifier`.
Toggle switch	The toggle switch type displays an ON/OFF toggle button. You can use this type to configure a preference that can have only one of two values. Although you typically use this type to represent preferences containing Boolean values, you can also use it with preferences containing non-Boolean values. The key for this type is `PSToggleSwitchSpecifier`.
Slider	The slider type displays a slider control. You can use this type for a preference that represents a range of values. The value for this type is a real number whose minimum and maximum value you specify. The key for this type is `PSSliderSpecifier`.
Multivalue	The multivalue type lets the user select one value from a list of values. You can use this type for a preference that supports a set of mutually exclusive values. The values can be of any type. The key for this type is `PSMultiValueSpecifier`.
Group	The group type is for organizing groups of preferences on a single page. The group type does not represent a configurable preference. It simply contains a title string that is displayed immediately before one or more configurable preferences. The key for this type is `PSGroupSpecifier`.
Child pane	The child pane type lets the user navigate to a new page of preferences. You use this type to implement hierarchical preferences. For more information on how you configure and use this preference type, see "Hierarchical Preferences" (page 27). The key for this type is `PSChildPaneSpecifier`.

For detailed information about the format of each preference type, see *Settings Application Schema Reference* . To learn how to create and edit Settings page files, see "Creating and Modifying the Settings Bundle" (page 29).

The Settings Bundle

A Settings bundle has the name `Settings.bundle` and resides in the top-level directory of your app's bundle. This bundle contains one or more Settings page files that describe the individual pages of preferences. It may also include other support files needed to display your preferences, such as images or localized strings. Table 4-2 lists the contents of a typical Settings bundle.

Table 4-2 Contents of the `Settings.bundle` directory

Item name	Description
`Root.plist`	The Settings page file containing the preferences for the root page. The name of this file must be `Root.plist`. The contents of this file are described in more detail in "The Settings Page File Format" (page 27).
Additional `.plist` files	If you build a set of hierarchical preferences using child panes, the contents for each child pane are stored in a separate Settings page file. You are responsible for naming these files and associating them with the correct child pane.
One or more `.lproj` directories	These directories store localized string resources for your Settings page files. Each directory contains a single strings file, whose title is specified in your Settings page file. The strings files provide the localized strings to display for your preferences.
Additional images	If you use the slider control, you can store the images for your slider in the top-level directory of the bundle.

In addition to the Settings bundle, the app bundle can contain a custom icon for your app settings. The Settings app displays the icon you provide next to the entry for your app preferences. For information about app icons and how you specify them, see *iOS App Programming Guide*.

When the Settings app launches, it checks each custom app for the presence of a Settings bundle. For each custom bundle it finds, it loads that bundle and displays the corresponding app's name and icon in the Settings main page. When the user taps the row belonging to your app, Settings loads the `Root.plist` Settings page file for your Settings bundle and uses that file to build your app's main page of preferences.

In addition to loading your bundle's `Root.plist` Settings page file, the Settings app also loads any language-specific resources for that file, as needed. Each Settings page file can have an associated `.strings` file containing localized values for any user-visible strings. As it prepares your preferences for display, the Settings app looks for string resources in the user's preferred language and substitutes them in your preferences page prior to display.

The Settings Page File Format

Each Settings page file is stored in the iPhone Settings property-list file format, which is a structured file format. The simplest way to edit Settings page files is to use the built-in editor facilities of Xcode; see "Preparing the Settings Page for Editing" (page 29). You can also edit property-list files using the Property List Editor app that comes with the Xcode tools.

> **Note** Xcode converts any XML-based property files in your project to binary format when building your app. This conversion saves space and is done for you automatically.

The root element of each Settings page file contains the keys listed in Table 4-3. Only one key is actually required, but it is recommended that you include both of them.

Table 4-3 Root-level keys of a preferences Settings page file

Key	Type	Value
`PreferenceSpecifiers` (required)	Array	The value for this key is an array of dictionaries, with each dictionary containing the information for a single control. For a list of control types, see Table 4-1 (page 25). For a description of the keys associated with each control, see *Settings Application Schema Reference*.
`StringsTable`	String	The name of the strings file associated with this file. A copy of this file (with appropriate localized strings) should be located in each of your bundle's language-specific project directories. If you do not include this key, the strings in this file are not localized. For information on how these strings are used, see "Localized Resources" (page 28).

Hierarchical Preferences

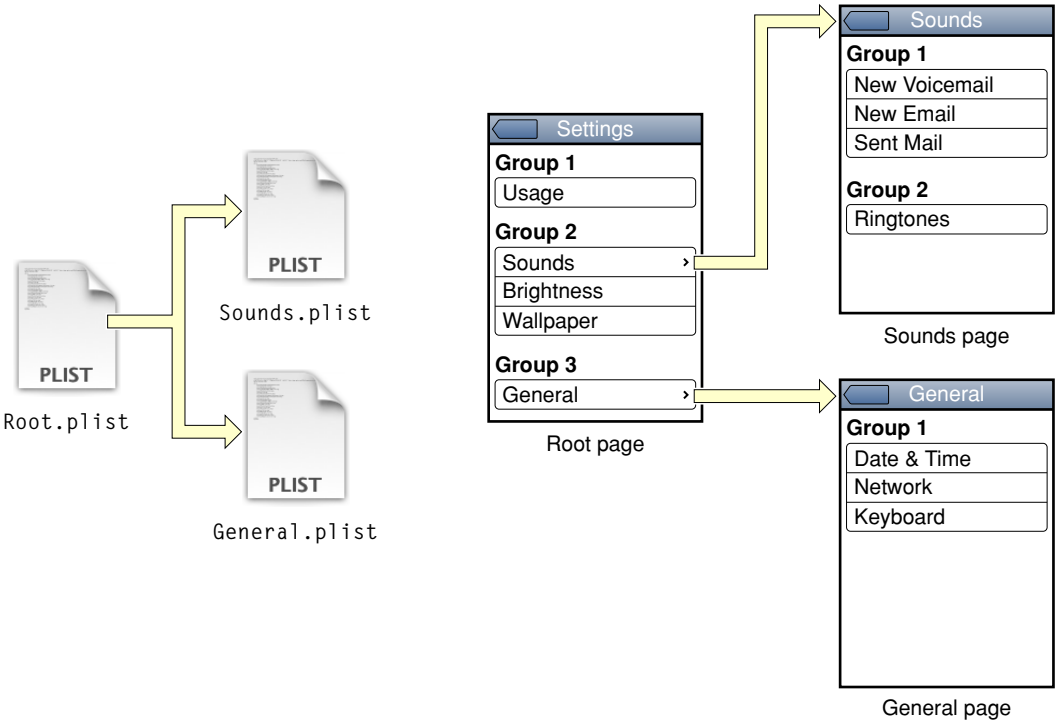
If you plan to organize your preferences hierarchically, each page you define must have its own separate `.plist` file. Each `.plist` file contains the set of preferences displayed only on that page. Your app's main preferences page is always stored in a file called `Root.plist`. Additional pages can be given any name you like.

To specify a link between a parent page and a child page, you include a child pane control in the parent page. A child pane control creates a row that, when tapped, displays a new page of settings. The `File` key of the child pane control identifies the name of the `.plist` file with the contents of the child page. The `Title` key

identifies the title of the child page; this title is also used as the text of the control used to display the child page. The Settings app automatically provides navigation controls on the child page to allow the user to navigate back to the parent page.

Figure 4-1 shows how this hierarchical set of pages works. The left side of the figure shows the `.plist` files, and the right side shows the relationships between the corresponding pages.

Figure 4-1 Organizing preferences using child panes



For more information about child pane controls and their associated keys, see *Settings Application Schema Reference*.

Localized Resources

Because preferences contain user-visible strings, you should provide localized versions of those strings with your Settings bundle. Each page of preferences can have an associated `.strings` file for each localization supported by your bundle. When the Settings app encounters a key that supports localization, it checks the appropriately localized `.strings` file for a matching key. If it finds one, it displays the value associated with that key.

When looking for localized resources such as `.strings` files, the Settings app follows the same rules that other iOS apps follow. It first tries to find a localized version of the resource that matches the user's preferred language setting. If no such resource exists, an appropriate fallback language is selected.

For information about the format of strings files, language-specific project directories, and how language-specific resources are retrieved from bundles, see *Internationalization Programming Topics*.

Creating and Modifying the Settings Bundle

Xcode provides a template for adding a Settings bundle to your current project. The default Settings bundle contains a `Root.plist` file and a default language directory for storing any localized resources. You can expand this bundle as needed to include additional property list files and resources needed by your Settings bundle.

Adding the Settings Bundle

To add a Settings bundle to your Xcode project:

1. Choose File > New > New File.
2. Under iOS, choose Resource, and then select the Settings Bundle template.
3. Name the file `Settings.bundle`.

In addition to adding a new Settings bundle to your project, Xcode automatically adds that bundle to the Copy Bundle Resources build phase of your app target. Thus, all you have to do is modify the property list files of your Settings bundle and add any needed resources.

The new Settings bundle has the following structure:

```
Settings.bundle/
    Root.plist
    en.lproj/
        Root.strings
```

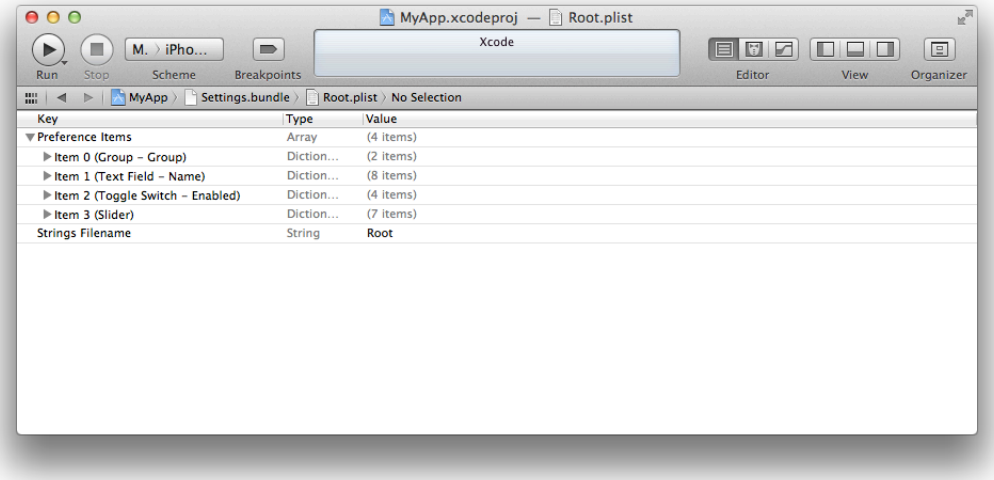
Preparing the Settings Page for Editing

Before editing any of the property-list files in your Settings bundle, you should configure the Xcode editor to format the contents of those files as iPhone settings. Xcode does this automatically for the `Root.plist` file, but you may need to format additional property-list files manually. To format a file as iPhone Settings, do the following:

1. Select the file.
2. Control-click the editor window and choose Property List Type > iPhone Settings plist if it is not already chosen.

Formatting a property list makes it easier to understand and edit the file's contents. Xcode substitutes human-readable strings (as shown in Figure 4-2) that are appropriate for the selected format.

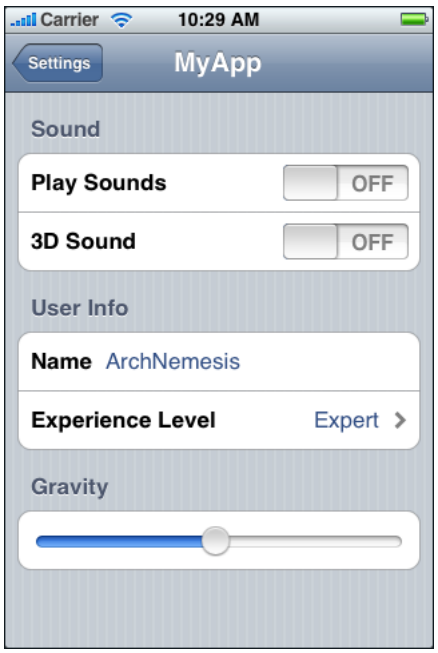
Figure 4-2 Formatted contents of the `Root.plist` file



Configuring a Settings Page: A Tutorial

This section shows you how to configure a Settings page to display the controls you want. The goal of the tutorial is to create a page like the one in Figure 4-3. If you have not yet created a Settings bundle for your project, you should do so as described in "Adding the Settings Bundle" (page 29) before proceeding with these steps.

Figure 4-3 A root Settings page



1. Disclose the Preference Items key to display the default items that come with the template.
2. Change the title of `Item 0` to `Sound`.
    - Disclose `Item 0` of `Preference Items`.
    - Change the value of the `Title` key from `Group` to `Sound`.
    - Leave the `Type` key set to `Group`.
    - Click the disclosure triangle of the item to hide its contents.
3. Create the first toggle switch for the renamed Sound group.
    - Select `Item 2` (the toggle switch item) of `Preference Items` and choose Edit > Cut.
    - Select `Item 0` and choose Edit > Paste. (This moves the toggle switch item in front of the text field item.)
    - Disclose the toggle switch item to reveal its configuration keys.
    - Change the value of the `Title` key to `Play Sounds`.
    - Change the value of the `Identifier` key to `play_sounds_preference`.

- Click the disclosure triangle of the item to hide its contents.

4. Create a second toggle switch for the Sound group.
   - Select `Item 1` (the Play Sounds toggle switch).
   - Choose Edit > Copy.
   - Choose Edit >Paste to place a copy of the toggle switch right after the first one.
   - Disclose the new toggle switch item to reveal its configuration keys.
   - Change the value of its `Title` key to `3D Sound`.
   - Change the value of its `Identifier` key to `3D_sound_preference`.
   - Click the disclosure triangle of the item to hide its contents.

   At this point, you have finished the first group of settings and are ready to create the User Info group.
5. Change `Item 3` into a Group control and name it `User Info`.
   - Click `Item 3` in the `Preferences Items`. This displays a pop-up menu with a list of item types.
   - From the pop-up menu, choose `Group` to change the type of the control.
   - Disclose the contents of `Item 3`.
   - Set the value of the `Title` key to `User Info`.
   - Click the disclosure triangle of the item to hide its contents.
6. Create the Name field.
   - Select `Item 4` in the `Preferences Items`.
   - Using the pop-up menu, change its type to `Text Field`.
   - Set the value of the `Title` key to `Name`.
   - Set the value of the `Identifier` key to `user_name`.
   - Click the disclosure triangle of the item to hide its contents.
7. Create the Experience Level settings.
   - Select `Item 4`.
   - Control-click the editor window and select Add Row to add a new item.
   - Set the type of the new item to `Multi Value`.
   - Disclose the item's contents and set its title to `Experience Level`, its identifier to `experience_preference`, and its default value to `0`.
   - With the Default Value key selected, Control-click and select Add Row to add a `Titles` array.
   - Select the `Titles` array and press Return to add a new subitem.
   - Add two more subitems to create a total of three items.

- Set the values of the subitems to `Beginner`, `Expert`, and `Master`.
- Hide the key's subitems.
- Add a new item for the `Values` array.
- Add three subitems to the `Values` array and set their values to `0`, `1`, and `2`.
- Hide the contents of `Item 5`.

8. Add the final group to your settings page.
    - Create a new item and set its type to `Group` and its title to `Gravity`.
    - Create another new item and set its type to `Slider`, its identifier to `gravity_preference`, its default value to `1`, and its maximum value to `2`.

Creating Additional Settings Page Files

The Settings Bundle template includes the `Root.plist` file, which defines your app's top Settings page. To define additional Settings pages, you must add additional property list files to your Settings bundle.

To add a property list file to your Settings bundle in Xcode, do the following:

1. Choose File > New > New File.
2. Under iOS, select Resource, and then select the Property List template.
3. Select the new file to display its contents in the editor.
4. Control-click the editor pane and choose Property List Type > iPhone Settings plist to format the contents.
5. Control-click the editor pane again and choose Add Row to add a new key.
6. Add and configure any additional keys you need.

After adding a new Settings page to your Settings bundle, you can edit the page's contents as described in "Configuring a Settings Page: A Tutorial" (page 31). To display the settings for your page, you must reference it from a child pane control as described in "Hierarchical Preferences" (page 27).

> **Note** In Xcode 4, adding a property-list file to your project does not automatically associate it with your Settings bundle. You must use the Finder to move any additional property-list files into your Settings bundle.

Debugging Preferences for Simulated Apps

When running your app, iOS Simulator stores any preferences values for your app in `~/Library/Application Support/iOS Simulator/User/Applications/`<*APP_ID*>`/Library/Preferences`, where <*APP_ID*> is a programmatically generated directory name that iOS uses to identify your app.

Each time you build your app, Xcode preserves your app preferences and other relevant library files. If you want to remove the current preferences for testing purposes, you can delete the app from Simulator or choose Reset Contents and Settings from the iOS Simulator menu.

Document Revision History

This table describes the changes to *Preferences and Settings Programming Guide*.

Date	Notes
2012-03-01	Updated the document to reflect new limits for key and value sizes.
2011-10-12	Updated the document to include information about Settings bundles and iOS in general. Also incorporated iCloud information. Removed the articles on storing NSColor objects and using Cocoa bindings and now link to their locations instead. Changed document name from *User Defaults Programming Topics*.
2007-10-31	Updated information about periodic autosave behavior.
2007-01-08	Corrected typos and capitalization mistakes.
2006-11-07	Added overview of procedure for storing non-property-list objects in user defaults, and linked to related article.
2006-09-05	Made small additions to the content. Changed title from "User Defaults." Expanded explanation of user defaults in introduction. Noted requirement that a default's value must be a property list value at the beginning of the "Using NSUserDefaults" article.
2005-08-11	Included an article that describes the use of NSUserDefaultsController. Corrected minor typographical errors.
2004-02-03	Added article "Storing NSColor in User Defaults".
2003-05-09	Linked to the Core Foundation Preferences Programming Topic, which was also incorrectly named.

Date	Notes
2003-01-13	Added link in limitations area to CFPreferences. Corrected class name in Defaults Domains Concept.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

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