Windows

Every iOS application needs at least one window—an instance of the `UIWindow` class—and some may include more than one window. A window object has several responsibilities:

- It contains your application's visible content.
- It plays a key role in the delivery of touch events to your views and other application objects.
- It works with your application's view controllers to facilitate orientation changes.

In iOS, windows do not have title bars, close boxes, or any other visual adornments. A window is always just a blank container for one or more views. Also, applications do not change their content by showing new windows. When you want to change the displayed content, you change the frontmost views of your window instead.

Most iOS applications create and use only one window during their lifetime. This window spans the entire main screen of the device and is loaded from the application's main nib file (or created programmatically) early in the life of the application. However, if an application supports the use of an external display for video out, it can create an additional window to display content on that external display. All other windows are typically created by the system, and are usually created in response to specific events, such as an incoming phone call.

Tasks That Involve Windows

For many applications, the only time the application interacts with its window is when it creates the window at startup. However, you can use your application's window object to perform a few application-related tasks:

- **Use the window object to convert points and rectangles to or from the window's local coordinate system.** For example, if you are provided with a value in window coordinates, you might want to convert it to the coordinate system of a specific view before trying to use it. For information on how to convert coordinates, see Converting Coordinates in the View Hierarchy.
- **Use window notifications to track window-related changes.** Windows generate notifications when they are shown or hidden or when they accept or resign the key status. You can use these notifications to perform actions in other parts of your application. For more information, see Monitoring Window Changes.

Creating and Configuring a Window

You can create and configure your application's main window programmatically or using Interface Builder. In either case, you create the window at launch time and should retain it and store a reference to it in your application delegate object. If your application creates additional windows, have the application create them lazily when they are needed. For example, if your application supports displaying content on an external display, it should wait until a display is connected before creating the corresponding window.

You should always create your application's main window at launch time regardless of whether your application is being launched into the foreground or background. Creating and configuring a window is not an expensive operation by itself. However, if your application is launched straight into the background, you should avoid making the window visible until your application enters the foreground.

Creating Windows in Interface Builder

Creating your application's main window using Interface Builder is simple because the Xcode project templates do it for you. Every new Xcode application project includes a main nib file (usually with the name `MainWindow.xib` or some variant thereof) that includes the application's main window. In addition, these templates also define an outlet for that window in the application delegate object. You use this outlet to access the window object in your code.

> **Important:** When creating your window in Interface Builder, it is recommended that you enable the Full Screen at Launch option in the attributes inspector. If this option is not enabled and your window is smaller than the screen of the target device, touch events will not be received by some of your views. This is because windows (like all views) do not receive touch events outside of their bounds rectangle. Because views are not clipped to the window's bounds by default, the views still appear visible but events do not reach them. Enabling the Full Screen at Launch option ensures that the window is sized appropriately for the current screen.

If you are retrofitting a project to use Interface Builder, creating a window using Interface Builder is a simple matter of dragging a window object to your nib file. Of course, you should also do the following:

- To access the window at runtime, you should connect the window to an outlet, typically one defined in your application delegate or the File's Owner of the nib file.
- If your retrofit plans include making your new nib file the main nib file of your application, you must also set the `NSMainNibFile` key in your application's `Info.plist` file to the name of your nib file. Changing the value of this key ensures that the nib file is loaded and available for use by the time the `application:didFinishLaunchingWithOptions:` method of your application delegate is called.

For more information about creating and configuring nib files, see *Interface Builder User Guide*. For information about how to load nib files into your application at runtime, see Nib Files in *Resource Programming Guide*.

Creating a Window Programmatically

If you prefer to create your application's main window programmatically, you should include code similar to the following in the `application:didFinishLaunchingWithOptions:` method of your application delegate:

```
self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]]
autorelease];
```

In the preceding example, `self.window` is assumed to be a declared property of your application delegate that is configured to retain the window object. If you were creating a window for an external display instead, you would assign it to a different variable and you would need to specify the bounds of the non main `UIScreen` object representing that display.

When creating windows, you should always set the size of the window to the full bounds of the screen. You should not reduce the size of the window to accommodate the status bar or any other items. The status bar always floats on top of the window anyway, so the only thing you should shrink to accommodate the status bar is the view you put into your window. And if you are using view controllers, the view controller should handle the sizing of your views automatically.

Adding Content to Your Window

Each window typically has a single root view object (managed by a corresponding view controller) that contains all of the other views representing your content. Using a single root view simplifies the process of changing your interface; to display new content, all you have to do is replace the root view. To install a view in your window, use the `addSubview:` method. For example, to install a view that is managed by a view controller, you would use code similar to the following:

```
[window addSubview:viewController.view];
```

In place of the preceding code, you can alternatively configure the `rootViewController` property of the window in your nib file. This property offers a convenient way to configure the root view of the window using a nib file instead of programmatically. If this property is set when the window is loaded from its nib file, UIKit automatically installs the view from the associated view controller as the root view of the window. This property is used only to install the root view and is not used by the window to communicate with the view controller.

You can use any view you want for a window's root view. Depending on your interface design, the root view can be a generic `UIView` object that acts as a container for one or more subviews, the root view can be a standard system view, or the root view can be a custom view that you define. Some standard system views that are commonly used as root views include scroll views, table views, and image views.

When configuring the root view of the window, you are responsible for setting its initial size and position within the window. For applications that do not include a status bar, or that display a translucent status bar, set the view size to match the size of the window. For applications that show an opaque status bar, position your view below the status bar and reduce its size accordingly. Subtracting the status bar height from the height of your view prevents the top portion of your view from being obscured.

Note: If the root view of your window is provided by a container view controller (such as a tab bar controller, navigation controller, or split-view controller), you do not need to set the initial size of the view yourself. The container view controller automatically sizes its view appropriately based on whether the status bar is visible.

Changing the Window Level

Each `UIWindow` object has a configurable `windowLevel` property that determines how that window is positioned relative to other windows. For the most part, you should not need to change the level of your application's windows. New windows are automatically assigned to the normal window level at creation time. The normal window level indicates that the window presents application-related content. Higher window levels are reserved for information that needs to float above the application content, such as the system status bar or alert messages. And although you can assign windows to these levels yourself, the system usually does this for you when you use specific interfaces. For example, when you show or hide the status bar or display an alert view, the system automatically creates the needed windows to display those items.

Monitoring Window Changes

If you want to track the appearance or disappearance of windows inside your application, you can do so using these window-related notifications:

- `UIWindowDidBecomeVisibleNotification`
- `UIWindowDidBecomeHiddenNotification`
- `UIWindowDidBecomeKeyNotification`
- `UIWindowDidResignKeyNotification`

These notifications are delivered in response to programmatic changes in your application's windows. Thus, when your application shows or hides a window, the `UIWindowDidBecomeVisibleNotification` and `UIWindowDidBecomeHiddenNotification` notifications are delivered accordingly. These notifications are not delivered when your application moves into the background execution state. Even though your window is not displayed on the screen while your application is in the background, it is still considered visible within the context of your application.

The `UIWindowDidBecomeKeyNotification` and `UIWindowDidResignKeyNotification` notifications help your application keep track of which window is the *key window*—that is, which window is currently receiving keyboard events and other non touch-related events. Whereas touch events are delivered to the window in which the touch occurred, events that do not have an associated coordinate value are delivered to the key window of your application. Only one window at a time may be key.

Displaying Content on an External Display

To display content on an external display, you must create an additional window for your application and associate it with the screen object representing the external display. New windows are normally associated with the main screen by default. Changing the window's associated screen object causes the contents of that window to be rerouted to the corresponding display. Once the window is associated with the correct screen, you can add views to it and show it just like you do for your application's main screen.

The `UIScreen` class maintains a list of screen objects representing the available hardware displays. Normally, there is only one screen object representing the main display for any iOS-based device, but devices that support connecting to an external display can have an additional screen object available. Devices that support an external display include iPhone and iPod touch devices that have Retina displays and the iPad. Older devices, such as iPhone 3GS, do not support external displays.

> **Note:** Because external displays are essentially a video-out connection, you should not expect touch events for views and controls in a window that is associated with an external display. In addition, it is your application's responsibility to update the contents of the window as needed. Thus, to mirror the contents of your main window, your application would need to create a duplicate set of views for the external display's window and update them in tandem with the views in your main window.

The process for displaying content on an external display is described in the following sections. However, the following steps summarize the basic process:

1. At application startup, register for the screen connection and disconnection notifications.
2. When it is time to display content on the external display, create and configure a window.
   - Use the `screens` property of `UIScreen` to obtain the screen object for the external display.
   - Create a `UIWindow` object and size it appropriately for the screen (or for your content).
   - Assign the `UIScreen` object for the external display to the `screen` property of the window.
   - Adjust the resolution of the screen object as needed to support your content.
   - Add any appropriate views to the window.
3. Show the window and update it normally.

Handling Screen Connection and Disconnection Notifications

Screen connection and disconnection notifications are crucial for handling changes to external displays gracefully. When the user connects or disconnects a display, the system sends appropriate notifications to your application. You should use these notifications to update your application state and create or release the window associated with the external display.

The important thing to remember about the connection and disconnection notifications is that they can come at any time, even when your application is suspended in the background. Therefore, it is best to observe the notifications from an object that is going to exist for the duration of your application's runtime, such as your application delegate. If your application is suspended, the notifications are queued until your application exits the suspended state and starts running in either the foreground or background.

Listing 2-1 shows the code used to register for connection and disconnection notifications. This method is called by the application delegate at initialization time but you could register for these notifications from other places in your application, too. The implementation of the handler methods is shown in Listing 2-2.

Listing 2-1 Registering for screen connect and disconnect notifications

```
- (void)setupScreenConnectionNotificationHandlers
{
    NSNotificationCenter* center = [NSNotificationCenter defaultCenter];

    [center addObserver:self selector:@selector(handleScreenConnectNotification:)
            name:UIScreenDidConnectNotification object:nil];
    [center addObserver:self selector:@selector(handleScreenDisconnectNotification:)
            name:UIScreenDidDisconnectNotification object:nil];
}
```

If your application is active when an external display is attached to the device, it should create a second window for that display and fill it with some content. The content does not need to be the final content you want to present. For example, if your application is not ready to use the extra screen, it can use the second window to display some placeholder content. If you do not create a window for the screen, or if you create a window but do not show it, a black field is displayed on the external display.

Listing 2-2 shows how to create a secondary window and fill it with some content. In this example, the application creates the window in the handler methods it uses to receive screen connection notifications. (For information about registering for connection and disconnection notifications, see Listing 2-1.) The handler method for the connection notification creates a secondary window, associates it with the newly connected screen and calls a method of the application's main view controller to add some content to the window and show it. The handler method for the disconnection notification releases the window and notifies the main view controller so that it can adjust its presentation accordingly.

Listing 2-2 Handling connect and disconnect notifications

```
- (void)handleScreenConnectNotification:(NSNotification*)aNotification
{
    UIScreen*    newScreen = [aNotification object];
    CGRect        screenBounds = newScreen.bounds;

    if (!_secondWindow)
    {
        _secondWindow = [[UIWindow alloc] initWithFrame:screenBounds];
        _secondWindow.screen = newScreen;

        // Set the initial UI for the window.
        [viewController displaySelectionInSecondaryWindow:_secondWindow];
    }
}

- (void)handleScreenDisconnectNotification:(NSNotification*)aNotification
{
    if (_secondWindow)
    {
        // Hide and then delete the window.
        _secondWindow.hidden = YES;
        [_secondWindow release];
        _secondWindow = nil;

        // Update the main screen based on what is showing here.
        [viewController displaySelectionOnMainScreen];
    }

}
```

Configuring a Window for an External Display

To display a window on an external screen, you must associate it with the correct screen object. This process involves locating the proper `UIScreen` object and assigning it to the window's `screen` property. You can get the list of screen objects from the `screens` class method of `UIScreen`. The array returned by this method always contains at least one object representing the main screen. If a second object is present, that object represents a connected external display.

Listing 2-3 shows a method that is called at application startup to see if an external display is already attached. If it is, the method creates a window, associates it with the external display, and adds some placeholder content before showing the window. In this case, the placeholder content is a white background and a label indicating that there is no content to display. To show the window, this method changes the value of its `hidden` property rather than calling `makeKeyAndVisible`. It does this because the window contains only static content and is not used to handle events.

Listing 2-3 Configuring a window for an external display

```
- (void)checkForExistingScreenAndInitializeIfPresent
{
    if ([[UIScreen screens] count] > 1)
    {
        // Associate the window with the second screen.
        // The main screen is always at index 0.
        UIScreen*    secondScreen = [[UIScreen screens] objectAtIndex:1];
        CGRect        screenBounds = secondScreen.bounds;

        _secondWindow = [[UIWindow alloc] initWithFrame:screenBounds];
        _secondWindow.screen = secondScreen;

        // Add a white background to the window
        UIView*            whiteField = [[UIView alloc] initWithFrame:screenBounds];
        whiteField.backgroundColor = [UIColor whiteColor];

        [_secondWindow addSubview:whiteField];
        [whiteField release];

        // Center a label in the view.
        NSString*    noContentString = [NSString stringWithFormat:@"<no content>"];
        CGSize        stringSize = [noContentString sizeWithFont:[UIFont
systemFontOfSize:18]];
```

```
        CGRect        labelSize = CGRectMake((screenBounds.size.width -
stringSize.width) / 2.0,
                                    (screenBounds.size.height - stringSize.height) /
2.0,
                                    stringSize.width, stringSize.height);

        UILabel*    noContentLabel = [[UILabel alloc] initWithFrame:labelSize];
        noContentLabel.text = noContentString;
        noContentLabel.font = [UIFont systemFontOfSize:18];
        [whiteField addSubview:noContentLabel];

        // Go ahead and show the window.
        _secondWindow.hidden = NO;
    }
}
```

Important: You should always associate a screen with a window before showing the window. While it is possible to change screens for a window that is currently visible, doing so is an expensive operation and should be avoided.

As soon as the window for an external screen is displayed, your application can begin updating it like any other window. You can add and remove subviews as needed, change the contents of subviews, animate changes to the views, and invalidate their contents as needed.

Configuring the Screen Mode of an External Display

Depending on your content, you might want to change the screen mode before associating your window with it. Many screens support multiple resolutions, some of which use different pixel aspect ratios. Screen objects use the most common screen mode by default, but you can change that mode to one that is more suitable for your content. For example, if you are implementing a game using OpenGL ES and your textures are designed for a 640 x 480 pixel screen, you might change the screen mode for screens with higher default resolutions.

If you plan to use a screen mode other than the default one, you should apply that mode to the `UIScreen` object before associating the screen with a window. The `UIScreenMode` class defines the attributes of a single screen mode. You can get a list of the modes supported by a screen from its `availableModes` property and iterate through the list for one that matches your needs.

For more information about screen modes, see *UIScreenMode Class Reference*.