Using Weakly Linked Classes in iOS

If your Xcode project uses weakly linked classes, you must ensure the availability of those classes at run time before using them. Attempting to use an unavailable class may generate a runtime binding error from the dynamic linker, which may terminate the corresponding process.

Xcode projects that use a base SDK of iOS 4.2 or later should use the `NSObject` `class` method to check the availability of weakly linked classes at run time. This simple, efficient mechanism takes advantage of the `NS_CLASS_AVAILABLE` class availability macro, available for most frameworks in iOS.

> **Important:** Check the most recent iOS Release Notes for the list of frameworks that do not yet support the `NS_CLASS_AVAILABLE` macro.

For iOS frameworks that support the `NS_CLASS_AVAILABLE` macro, conditionalize your code for weakly linked classes as demonstrated in the following example:

```
if ([UIPrintInteractionController class]) {
    // Create an instance of the class and use it.
} else {
    // Alternate code path to follow when the
    // class is not available.
}
```

This works because if a weakly linked class is not available, sending a message to it is like sending a message to `nil`. If you subclass a weakly linked class and the superclass is unavailable, then the subclass also appears unavailable.

To use the `class` method as shown here, you need to do more than ensure that a framework supports the `NS_CLASS_AVAILABLE` macro. You must also configure certain project settings. With these required settings in place, the preceding code safely tests the availability of a class, even when running on a version of iOS in which the class is not present. These settings are as follows:

- The base SDK for your Xcode project must be iOS 4.2 or newer. The name for this setting in the build settings editor is `SDKROOT (Base SDK)`.
- The deployment target for your project must be iOS 3.1 or newer. The name for this setting is `MACOSX_DEPLOYMENT_TARGET (Mac OS X Deployment Target)`.
- The compiler for your project must be the LLVM-GCC 4.2 compiler or newer, or the LLVM compiler (Clang) 1.5 or newer. The name for this setting is `GCC_VERSION (C/C++ Compiler Version)`.
- You must ensure that any frameworks not available in your project's deployment target are weakly linked, rather than required. See "Weak Linking to an Entire Framework" and "Linking Libraries and Frameworks" in *Xcode Project Management Guide*.

For information on using the Xcode build settings editor, see "Building Products" in *Xcode Project Management Guide*.

In Mac OS X (and in iOS projects that do not meet the set of conditions just listed), you cannot use the `class` method to determine if a weakly linked class is available. Instead, use the

the `class` method to determine if a weakly linked class is available. Instead, use the `NSClassFromString` function in code similar to the following:

```
Class cls = NSClassFromString (@"NSRegularExpression");
if (cls) {
    // Create an instance of the class and use it.
} else {
    // Alternate code path to follow when the
    // class is not available.
}
```

Using Weakly Linked Methods, Functions, and Symbols

If your project uses weakly linked methods, functions, or external symbols, you must ensure their availability at run time before using them. If you attempt to use an unavailable item, the dynamic linker may generate a runtime binding error and terminate the corresponding process.

Suppose you set the base SDK in your Xcode project to iOS 4.0. This allows your code to use features in that version of the operating system when running in that version. Suppose also that you want your software to run in iOS 3.1, even though it cannot use the newer features in that version of the OS. Allow this by setting the deployment target to the earlier version of the operating system.

In Objective-C, the `instancesRespondToSelector:` method tells you if a given method selector is available. For example, to use the `availableCaptureModesForCameraDevice:` method, first available in iOS 4.0, you could use code like the following:

Listing 3-1 Checking the availability of an Objective-C method

```
if ([UIImagePickerController instancesRespondToSelector:
                @selector (availableCaptureModesForCameraDevice:)]) {
    // Method is available for use.
    // Your code can check if video capture is available and,
    // if it is, offer that option.
} else {
    // Method is not available.
    // Alternate code to use only still image capture.
}
```

When your code runs in iOS 4.0 or later, it can call `availableCaptureModesForCameraDevice:` to determine if video capture is available on the device. When it runs in iOS 3.1, however, it must assume that only still image capture is available.

If you were to build this code with various settings, you would see the following results:

- If you specify a base SDK setting of `iphoneos3.1`:

The build would fail because the `availableCaptureModesForCameraDevice:` method is not defined in that system version.

- If you specify a base SDK setting of `iphoneos4.0`, and then set the deployment target to:
  - `iphoneos4.0`: The software would run only in iOS 4.0 or later and would fail to launch on earlier systems.
  - `iphoneos3.1`: The software would run in iOS 4.0 and in iOS 3.1, but would fail to launch on earlier systems. When running in iOS 3.1, the software would use the alternate code to capture images.

Check the availability of an Objective-C property by passing the getter method name (which is the same as the property name) to `instancesRespondToSelector:`.

To determine if a weakly linked C function is available, use the fact that the linker sets the address of unavailable functions to `NULL`. Check a function's address—and hence, its availability—by comparing the address to `NULL` or `nil`. For example, before using the `CGColorCreateGenericCMYK` function in a project whose deployment target is earlier than Mac OS X v10.5, use code like the following:

Listing 3-2 Checking the availability of a C function

```
if (CGColorCreateGenericCMYK != NULL) {
    CGColorCreateGenericCMYK (0.1,0.5.0.0,1.0,0.1);
} else {
    // Function is not available.
    // Alternate code to create a color object with earlier technology
}
```

Note: To check the availability of a function, explicitly compare its address to `NULL` or `nil`. You cannot use the negation operator (`!`) to negate the address of a function to check its availability. Also, note that the name of a C function is synonymous with its address. That is, `&myFunction` is equivalent to `myFunction`.

Check the availability of an external (`extern`) constant or a notification name by explicitly comparing its address—and *not* the symbol's bare name—to `NULL` or `nil`.

Weak Linking to an Entire Framework

If you are using a recently added framework—one that became available after your deployment target—you must explicitly weak link to the framework itself. For example, say you want to link to the Accelerate framework, first available in iOS 4.0, to use its features on systems in which they're available. In addition, say you set your deployment target to iOS 3.1.3, allowing users of that version of iOS to use your app without the new features. In this example, you must weak link to the Accelerate framework.

When using a framework that *is* available in your deployment target, you should require that framework (and not weakly link it).

framework (and not weakly link it).

For information on how to weakly link to a framework, refer to "Linking Libraries and Frameworks" in *Xcode Project Management Guide*.

Conditionally Compiling for Different SDKs

If you use one set of source code to build for more than one base SDK, you might need to conditionalize for the base SDK in use. Do this by using preprocessor directives with the macros defined in `Availability.h`.

> **Note:** The `Availability.h` header is for targeting iOS and for targeting Mac OS X v10.6 and later. The older `AvailabilityMacros.h` header was introduced in Mac OS X v10.2. These files reside in the `/usr/include` directory.

Suppose you want to compile the code shown in Listing 3-2 using a base SDK setting of `macosx10.4`. The portion of the code that refers to the `CGColorCreateGenericCMYK` function —introduced in Mac OS X v10.5—must be masked during the build. This is because any reference to the unavailable `CGColorCreateGenericCMYK` function would cause a compiler error.

To allow the code to build, use the `__MAC_OS_X_VERSION_MAX_ALLOWED` macro to:

- Ensure that the project's target is Mac OS X and not iOS
- Hide code that is unavailable in the base SDK

The following code excerpt demonstrates this. Notice the use of the numerical value `1050` instead of the symbol `__MAC_10_5` in the `#if` comparison clause: If the code is loaded on an older system that does not include the symbol definition, the comparison still works.

Listing 3-3 Using preprocessor directives for conditional compilation

```
#ifdef __MAC_OS_X_VERSION_MAX_ALLOWED
    // code only compiled when targeting Mac OS X and not iOS
    // note use of 1050 instead of __MAC_10_5
#if __MAC_OS_X_VERSION_MAX_ALLOWED >= 1050
    if (CGColorCreateGenericCMYK != NULL) {
        CGColorCreateGenericCMYK(0.1,0.5.0.0,1.0,0.1);
    } else {
#endif
    // code to create a color object with earlier technology
#if __MAC_OS_X_VERSION_MAX_ALLOWED >= 1050
    }
#endif
#endif
}
```

In addition to using preprocessor macros, the preceding code also assumes that the code could be compiled using a newer base SDK but deployed on a computer running Mac OS X v10.4 and earlier. Specifically, it checks for the existence of the weakly linked `CGColorCreateGenericCMYK` symbol before attempting to call it. This prevents the code from generating a runtime error, which can occur if you build the code against a newer SDK but deploy it on an older system. For more information about instituting runtime checks to determine the presence of symbols, see "Using Weakly Linked Classes in iOS" and "Using Weakly Linked Methods and Functions."

Finding Instances of Deprecated API Usage

As iOS and Mac OS X evolve, the APIs and technologies they encompass are sometimes changed to meet the needs of developers. As part of this evolution, less efficient interfaces are deprecated in favor of newer ones. Availability macros attached to declarations in header files help you find deprecated interfaces. Reference documentation also flags deprecated interfaces.

> **Note:** Deprecation does not mean the immediate deletion of an interface from a framework or library. It is simply a way to flag interfaces for which better alternatives exist. You can use deprecated APIs in your code. However, Apple recommends that you migrate to newer interfaces as soon as possible because deprecated APIs may be deleted from a future version of the OS. Check the header files or documentation of the deprecated API for information about any recommended replacement interfaces.

If you compile a project with a deployment target of Mac OS X v10.5 and use an interface tagged as deprecated, the compiler issues a corresponding warning. The warning includes the name of the deprecated interface and where in your code it was used. For example, if the `HPurge` function were deprecated, you would get an error similar to the following:

```
'HPurge' is deprecated (declared at /Users/steve/MyProject/main.c:51)
```

To locate instances of deprecated API use in your code, look for warnings of this type. If your project has many warnings, use the search field in Xcode to filter the warnings list based on the "deprecated" keyword.

Determining the Version of the Operating System or a Framework

In rare instances, checking the run time availability of a symbol is not a complete solution. For example, if the behavior of a method changed from one OS version to another, or if a bug was fixed in a previously available method, it's important to write your code to take those changes into account.

The technique to use for checking operating system version depends on your target platform, as follows:

- To check the version of iOS at run time, use code like this:.

```
NSString *osVersion = [[UIDevice currentDevice] systemVersion];
```

- To check the version of Mac OS X at run time, use the Gestalt function and the System Version Selectors constants.

Alternatively, for many frameworks, you can check the version of a specific framework at run time. To do this, use global framework-version constants—if they are provided by the framework. For example, the Application Kit (in `NSApplication.h`) declares the `NSAppKitVersionNumber` constant which you can use to detect different versions of the Application Kit framework:

```
APPKIT_EXTERN double NSAppKitVersionNumber;
#define NSAppKitVersionNumber10_0 577
#define NSAppKitVersionNumber10_1 620
#define NSAppKitVersionNumber10_2 663
#define NSAppKitVersionNumber10_2_3 663.6
#define NSAppKitVersionNumber10_3 743
#define NSAppKitVersionNumber10_3_2 743.14
#define NSAppKitVersionNumber10_3_3 743.2
#define NSAppKitVersionNumber10_3_5 743.24
#define NSAppKitVersionNumber10_3_7 743.33
#define NSAppKitVersionNumber10_3_9 743.36
#define NSAppKitVersionNumber10_4 824
#define NSAppKitVersionNumber10_4_1 824.1
#define NSAppKitVersionNumber10_4_3 824.23
#define NSAppKitVersionNumber10_4_4 824.33
#define NSAppKitVersionNumber10_4_7 824.41
#define NSAppKitVersionNumber10_5 949
#define NSAppKitVersionNumber10_5_2 949.27
#define NSAppKitVersionNumber10_5_3 949.33
```

You can compare against this constant's value to determine which version of the Application Kit your code is running against. One typical approach is to floor the value of the global constant and check the result against the constants declared in `NSApplication.h`. For example:

```
if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_0) {
  /* On a 10.0.x or earlier system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_1) {
  /* On a 10.1 - 10.1.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_2) {
  /* On a 10.2 - 10.2.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_3) {
  /* On 10.3 - 10.3.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_4) {
```

```
  /* On a 10.4 - 10.4.x system */
} else if (floor(NSAppKitVersionNumber) <= NSAppKitVersionNumber10_5) {
  /* On a 10.5 - 10.5.x system */
} else {
  /* 10.6 or later system */
}
```

Similarly, Foundation (in `NSObjCRuntime.h`) declares the `NSFoundationVersionNumber` global constant and specific values for each version.

Some individual headers for other objects and components may also declare the version numbers for `NSAppKitVersionNumber` where some bug fix or functionality is available in a given update.