Xcode Release Notes

About Xcode 6.3 beta

Supported Configurations

Xcode 6.3 requires a Mac running OS X 10.10.

Xcode 6.3 includes SDKs for iOS 8.3 and OS X versions 10.9 and 10.10. To develop apps targeting prior versions of OS X or iOS, see the section "About SDKs and the iOS Simulator" in *What's New in Xcode* available on developer.apple.com or from the Help > What's New in Xcode command when running Xcode.

Installation

This release is a single application bundle. To install, double-click the downloaded DMG file, and drag the Xcode-Beta.app file to your Applications folder.

From within Xcode, you launch additional developer tools, such as Instruments and FileMerge, via the Xcode > Open Developer Tool command. You can keep the additional tools in the Dock for direct access when Xcode is not running.

Installing Xcode on OS X Server

To use Xcode's Continuous Integration service with this Xcode beta, you need OS X 10.10 with OS X Server 4.0.

Once you have installed OS X, OS X Server and Xcode, point OS X Server to this Xcode beta:

1. Open Server.app
2. Select the Xcode service
3. Choose Xcode

Technical Support and Learning Resources

Apple offers a number of resources where you can get Xcode development support:

• http://developer.apple.com: The Apple Developer website is the best source for up-to-date technical documentation on Xcode, iOS, and OS X.

• http://developer.apple.com/xcode: The Xcode home page on the Apple Developer website provides information on acquiring the latest version of Xcode.

• http://devforums.apple.com: The Apple Developer Forums are a good place to interact with fellow developers and Apple engineers, in a moderated web forum that also offers email notifications. The Developer Forums also feature a dedicated topic for Xcode developer previews.

Use http://bugreport.apple.com to report issues to Apple. Include detailed information of the issue, including the system and developer tools version information, and any relevant crash logs or console messages.

New Features in Xcode 6.3

Swift 1.2

Xcode 6.3 includes a new version of the Swift language. It includes a number of noteworthy changes to the language, detailed in the *New in Xcode 6.3 beta* section. Xcode 6.3 provides a migrator for moving your code to Swift 1.2.

Objective-C

Xcode 6.3 includes enhancements that ease interoperability between Swift and Objective-C code.

New in Xcode 6.3 beta

Swift Migrator from Swift 1.1 to Swift 1.2

- A source migrator tool has been provided to help migrate your Swift source code from Swift 1.1 (Xcode 6.1.1) to Swift 1.2 (Xcode 6.3.) In Xcode, select `Edit > Convert > To Swift 1.2`.

Swift Language Enhancements

- Swift now supports building targets incrementally, i.e. not rebuilding every Swift source file in a target when a single file is changed. This is based on a conservative dependency analysis, so you may still see more files rebuilding than absolutely necessary. If you find any cases where a file is not rebuilt when it should be, please file a bug report; running Clean on your target should then allow you to complete your build normally. (18248514)

- A new `Set` data structure is included which provides a generic collection of unique elements, with full value semantics. It bridges with `NSSet`, providing functionality analogous to `Array` and `Dictionary`. (14661754)

- The "`if let`" construct has been expanded to allow testing multiple optionals and guarding conditions in a single if (or while) statement using syntax similar to generic constraints:

```
if let a = foo(), b = bar() where a < b,
   let c = baz() {
}
```

  This allows you to test multiple optionals and include intervening boolean conditions, without introducing undesirable nesting (i.e., to avoid the "pyramid of doom"). (19382942)

- `let` constants have been generalized to no longer require immediate initialization. The new rule is that a `let` constant must be initialized before use (like a `var`), and that it may only be initialized: not reassigned or mutated after initialization. This enables patterns like:

```
let x: SomeThing
if condition {
  x = foo()
} else {
  x = bar()
}
use(x)
```

  which formerly required the use of a `var`, even though there is no mutation taking place. (16181314)

- "`static`" methods and properties are now allowed in classes (as an alias for "`class final`"). You are now allowed to declare `static` stored properties in classes, which have global storage and are lazily initialized on first access (like global variables). Protocols now declare type requirements as "`static`" requirements instead of declaring them as "`class`" requirements. (17198298)

- Type inference for single-expression closures has been improved in several ways:
  - Closures that are comprised of a single return statement are now type checked as single-expression closures.
  - Unannotated single-expression closures with non-`Void` return types can now be used in `Void` contexts.
  - Situations where a multi-statement closure's type could not be inferred because of a missing return-type annotation are now properly diagnosed.

- Swift enums can now be exported to Objective-C using the `@objc` attribute. `@objc` enums must declare an integer raw type, and cannot be generic or use associated values. Because Objective-C enums are not namespaced, enum cases are imported into Objective-C as the concatenation of the enum name and case name. (16967385)

  For example, this Swift declaration:

  ```
  @objc
  enum Bear: Int {
     case Black, Grizzly, Polar
  }
  ```

  imports into Objective-C as:

  ```
  typedef NS_ENUM(NSInteger, Bear) {
     BearBlack, BearGrizzly, BearPolar
  };
  ```

- Objective-C language extensions are now available to indicate the nullability of pointers and blocks in Objective-C APIs, allowing your Objective-C APIs to be imported without `ImplicitlyUnwrappedOptional`. See below for more details (18868820).

- Swift can now partially import C aggregates containing unions, bitfields, SIMD vector types, and other C language features that are not natively supported in Swift. The unsupported fields will not be accessible from Swift, but C and Objective-C APIs that have arguments and return values of these types can be used in Swift. This includes the Foundation `NSDecimal` type and the GLKit `GLKVector` and `GLKMatrix` types, among others. (15951448)

- Imported C structs now have a default initializer in Swift, which initializes all of the struct's fields to zero. For example:

  ```
  import Darwin
  var devNullStat = stat()
  stat("/dev/null", &devNullStat)
  ```

  If a structure contains fields that cannot be correctly zero initialized (i.e. pointer fields marked with the new `__nonnull` modifier), this default initializer will be suppressed. (18338802)

- New APIs for converting among the Index types for `String`, `String.UnicodeScalarView`, `String.UTF16View`, and `String.UTF8View` are available, as well as APIs for converting each of the String views into Strings. (18018911)

- Type values now print as the full demangled type name when used with println or string interpolation. (18947381)

```
    toString(Int.self)          // prints “Swift.Int"
    println([Float].self)       // prints "Swift.Array<Swift.Float>”
    println((Int, String).self) // prints "(Swift.Int, Swift.String)"
```

- A new “@noescape” attribute may be used on closure parameters to functions. This indicates that the parameter is only ever called (or passed as an @noescape parameter in a call), which means that it cannot outlive the lifetime of the call. This enables some minor performance optimizations, but more importantly disables the “self.” requirement in closure arguments. This enables control-flow-like functions to be more transparent about their behavior. In a future beta, the standard library will adopt this attribute in functions like `autoreleasepool()`. (16323038)

```
        func autoreleasepool(@noescape code: () -> ()) {
          pushAutoreleasePool()
          code()
          popAutoreleasePool()
        }
```

- Performance is substantially improved over Swift 1.1 in many cases. For example, multidimensional arrays are algorithmically faster in some cases, unoptimized code is much faster in many cases, and many other improvements have been made.

- The diagnostics emitted for expression type check errors are greatly improved in many cases. (18869019)

- Type checker performance for many common expression kinds has been greatly improved. This can significantly improve build times and reduces the number of “expression too complex” errors. (18868985)

Swift Language Changes

- The notions of guaranteed conversion and “forced failable” conversion are now separated into two operators. Forced failable conversion now uses the `as!` operator. The `!` makes it clear to readers of code that the cast may fail and produce a runtime error. The “`as`” operator remains for upcasts (e.g. “someDerivedValue as Base”) and type annotations (“0 as Int8”) which are guaranteed to never fail.(19031957)

- Immutable (`let`) properties in struct and class initializers have been revised to standardize on a general “`lets` are singly initialized but never reassigned or mutated” model. Previously, they were completely mutable within the body of initializers. Now, they are only allowed to be assigned to once to provide their value. If the property has an initial value in its declaration, that counts as the initial value for all initializers. (19035287)

- The implicit conversions from bridged Objective-C classes (`NSString/NSArray/NSDictionary`) to their corresponding Swift value types (`String/Array/Dictionary`) have been removed, making the Swift type system simpler and more predictable. (18311362)

  This means that the following code will no longer work:

  ```
      import Foundation
      func log(s: String) { println(x) }
      let ns: NSString = "some NSString" // okay: literals still work
      log(ns)   // fails with the error
               // "'NSString' is not convertible to 'String'"
  ```

  In order to perform such a bridging conversion, make the conversion explicit with the as keyword:

  ```
    log(ns as String) // succeeds
  ```

  Implicit conversions from Swift value types to their bridged Objective-C classes are still permitted. For example:

  ```
    func nsLog(ns: NSString) { println(ns) }
    let s: String = “some String”
    nsLog(s) // okay: implicit conversion from String to NSString is still
  permitted
  ```

- The `@autoclosure` attribute is now an attribute on a parameter, not an attribute on the parameter’s type. (15217242)

  Where before you might have used:

  ```
    func assert(predicate : @autoclosure () -> Bool) {… }
  ```

  you now write this as:

  ```
    func assert(@autoclosure predicate : () -> Bool) {… }
  ```

- The `@autoclosure` attribute on parameters now implies the new `@noescape` attribute. This intentionally limits the power of `@autoclosure` to control-flow and lazy evaluation use cases.

- Curried function parameters can now specify argument labels (17237268):

  ```
    func curryUnnamed(a: Int)(_ b: Int) { return a + b }
    curryUnnamed(1)(2)

    func curryNamed(first a: Int)(second b: Int) -> Int { return a + b }
    curryNamed(first: 1)(second: 2)
  ```

- Swift now detects discrepancies between overloading and overriding in the Swift type system and the effective behavior seen via the Objective-C runtime. (18391046, 18383574)

  For example, the following conflict between the Objective-C setter for “property” in a class and the method “setProperty” in its extension is now diagnosed:

  ```
  class A : NSObject {
    var property: String = "Hello" // note: Objective-C method 'setProperty:’
                                   // previously declared by setter for
                                   // 'property’ here
  }

  extension A {
    func setProperty(str: String) { }   // error: method ‘setProperty’
                                        // redeclares Objective-C method
                                        //'setProperty:’
  }
  ```

  Similar checking applies to accidental overrides in the Objective-C runtime:

  ```
  class B : NSObject {
    func method(arg: String) { }   // note: overridden declaration
                                   // here has type ‘(String) -> ()’
  }

  class C : B {
    func method(arg: [String]) { } // error: overriding method with
                                   // selector ‘method:’ has incompatible
                                   // type ‘([String]) -> ()’
  }
  ```

  as well as protocol conformances:

  ```
  class MyDelegate : NSObject, NSURLSessionDelegate {
    func URLSession(session: NSURLSession, didBecomeInvalidWithError: Bool){ }
        // error: Objective-C method 'URLSession:didBecomeInvalidWithError:'
        // provided by method 'URLSession(_:didBecomeInvalidWithError:)'
        // conflicts with optional requirement method
        // 'URLSession(_:didBecomeInvalidWithError:)' in protocol
        // 'NSURLSessionDelegate'
  }
  ```

Swift Language Fixes

- Dynamic casts ("`as!`", "`as?`" and "`is`") now work with Swift protocol types, so long as they have no associated types. (18869156)

- Adding conformances within a Playground now works as expected, for example:

```
struct Point {
  var x, y: Double
}

extension Point : Printable {
  var description: String {
    return "(\(x), \(y))"
  }
}

var p1 = Point(x: 1.5, y: 2.5)
println(p1) // prints "(1.5, 2.5)"
```

- Imported `NS_ENUM` types with undocumented values, such as `UIViewAnimationCurve`, can now be converted from their raw integer values using the `init(rawValue:)` initializer without being reset to nil. Code that used `unsafeBitCast` as a workaround for this issue can be written to use the raw value initializer. (19005771)

  For example:

```
    let animationCurve =
unsafeBitCast(userInfo[UIKeyboardAnimationCurveUserInfoKey].integerValue,
                     UIViewAnimationCurve.self)
```

  can now be written instead as:

```
    let animationCurve = UIViewAnimationCurve(rawValue:
          userInfo[UIKeyboardAnimationCurveUserInfoKey].integerValue)!
```

- Negative floating-point literals are now accepted as raw values in enums. (16504472)

- Unowned references to Objective-C objects, or Swift objects inheriting from Objective-C objects, no longer cause a crash if the object holding the unowned reference is deallocated after the referenced object has been released. (18091547)

- Variables and properties with observing accessors no longer require an explicit type if it can be inferred from the initial value expression. (18148072)

- Generic curried functions no longer produce random results when fully applied. (18988428)

- Comparing the result of a failed `NSClassFromString` lookup against nil now behaves correctly. (19318533)

- Subclasses that override base class methods with co- or contravariance in Optional types no longer cause crashes at runtime. (19321484)

  For example:

```
class Base {
  func foo(x: String) -> String? { return x }
}
class Derived: Base {
  override func foo(x: String?) -> String { return x! }
}
```

Objective-C Language Enhancements

- Objective-C APIs can now express the "nullability" of parameters, return types, properties, variables, etc. For example, here is the expression of nullability for several UITableView APIs:

```
    -(void)registerNib:(nonnull UINib *)nib forCellReuseIdentifier:(nonnull
NSString *)identifier;
     -(nullable UITableViewCell *)cellForRowAtIndexPath:(nonnull
NSIndexPath)indexPath;
    @property (nonatomic, readwrite, retain, nullable) UIView *backgroundView;
```

  The nullability qualifiers affect the optionality of the Objective-C APIs when in Swift. Instead of being imported as implicitly-unwrapped optionals (e.g., `UINib!`), `nonnull`-qualified types are imported as non-optional (e.g., `UINib`) and `nullable`-qualified types are imported as optional (e.g., `UITableViewCell?`), so the above APIs will be seen in Swift as:

```
func registerNib(nib: UINib, forCellReuseIdentifier identifier: String)
func cellForRowAtIndexPath(indexPath: NSIndexPath) -> UITableViewCell?
var backgroundView: UIView?
```

  Nullability qualifiers can also be applied to arbitrary pointer types, including C pointers, block pointers, and C++ member pointers, using double-underscored versions of the nullability qualifiers. For example, consider a C API such as:

```
void enumerateStrings(__nonnull CFStringRef (^ __nullable callback)(void));
```

  Here, the callback itself is nullable and the result type of that callback is nonnull. This API will be usable from Swift as:

```
func enumerateStrings(callback: (() -> CFString)?)
```

  In all, there are three different kinds of nullability specifiers, which can be spelled with a double-underscore (on any pointer type) or without (for Objective-C properties, method result types, and method parameter types):

Type qualifier spelling	Objective-C property/method spelling	Swift view	Meaning
`__nonnull`	`nonnull`	Non-optional, e.g., `UINib`	The value is never expected to be nil (except perhaps due to messaging nil in the argument)
`__nullable`	`nullable`	Optional, e.g., `UITableViewCell?`	The value can be nil
`__null_unspecified`	`null_unspecified`	Implicitly-unwrapped optional, e.g., `NSDate!`	It is unknown whether the value can be nil (very rare)

Particularly in Objective-C APIs, many pointers tend to be `nonnull`. Therefore, Objective-C provides "audited" regions (via a new `#pragma`) that assume that unannotated pointers are `nonnull`. For example, the following example is equivalent to the first example, but uses audited regions to simplify the presentation:

```
    #pragma clang assume_nonnull begin
    // …
    -(void)registerNib:(UINib *)nib forCellReuseIdentifier:(NSString
*)identifier;
     -(nullable UITableViewCell *)cellForRowAtIndexPath:(NSIndexPath)indexPath;
    @property (nonatomic, readwrite, retain, nullable) UIView *backgroundView;
    // …
    #pragma clang assume_nonnull end
```

For consistency, we recommend using audited regions in all Objective-C headers that describe the nullability of their APIs, and to avoid `null_unspecified` except as a transitional tool while introducing nullability into existing headers.

Adding nullability annotations to Objective-C APIs does not affect backward compatibility or the way in which the compiler generates code. For example, `nonnull` pointers can still end up being nil in some cases, such as when messaging a nil receiver. However, nullability annotations—in addition to improving the experience in Swift—provide new warnings in Objective-C if, for example, a nil argument is passed to a `nonnull` parameter, making Objective-C APIs more expressive and easier to use correctly. (18868820)

- Objective-C APIs can now express the nullability of properties whose setters allow nil (to "reset" the value to some default) but whose getters never produce nil (because they provide some default instead) using the `null_resettable` property attribute. One such property is UIView's `tintColor`, which substitutes a default system tint color when no tint color has been specified. (19051334)

  For example:

  ```
  @property (nonatomic, retain, null_resettable) UIColor *tintColor;
  ```

  Such APIs are imported into Swift via implicitly-unwrapped optionals, e.g.,

  ```
  var tintColor: UIColor!
  ```

- Parameters of C pointer type or block pointer type can be annotated with the new `noescape` attribute to indicate that pointer argument won't "escape" the function or method it is being passed to. (19389222)

  In such cases, it's safe to (for example) pass the address of a local variable. `noescape` block pointer parameters will be imported into Swift as `@noescape` parameters:

  ```
  void executeImmediately(__attribute__((noescape)) void (^callback)(void);
  ```

  is imported into Swift as:

  ```
  func executeImmediately(@noescape callback: () -> Void)
  ```

Debugger Enhancements

- LLDB now includes a prototype for `printf()` by default when evaluating C/C++/Objective-C expressions. This improves the expression evaluation experience on arm64 devices, but may conflict with user-defined expression prefixes in .lldbinit that have a conflicting declaration of `printf()`. If you see errors during expression evaluation this may be the root cause. (19024779)

Apple LLVM Compiler Version 6.1

- Xcode 6.3 updates the Apple LLVM compiler to version 6.1.0. This new compiler includes full support for the C++14 language standard, a wide range of enhanced warning diagnostics, and new optimizations. Support for the arm64 architecture has been significantly revised to align with ARM's implementation, where the most visible impact is that a few of the vector intrinsics have changed to match ARM's specifications.

ARM64 Intrinsics Changes

- The argument ordering for the arm64 vfma/vfms lane intrinsics has changed. Since that change may not trigger compile-time errors but will break code at run-time, we are staging the transition to make it less risky. By default, the compiler will now warn about any use of those intrinsics and will retain the old behavior. As soon as possible, you should adopt the new behavior and define the `USE_CORRECT_VFMA_INTRINSICS` macro to 1 to inform the compiler of that. You can also define that macro to 0 to silence the warnings and keep the old behavior, but do not leave your code in that state for long, since we plan to remove support for the old behavior in a future release. (17964959)

Issues Resolved in Xcode 6.3 beta

Xcode Interface Builder

- Views that have autoresizing masks and lie inside of UITableView, UICollectionView, or NSScrollView no longer get misaligned when opening the document. (18404033)

Known Issues in Xcode 6.3 beta

Xcode Interface Builder

- Sometimes Interface Builder fails to render a subclass of a designable class on the canvas, or it fails to show inspectable properties inherited from a superclass. (19512849)

  Workaround: Add `IB_DESIGNABLE` or `@IBDesignable` to the subclass' declaration

Migration Assistant

- "Convert to Swift 1.2" may generate build errors when run. These errors can be safely ignored and don't affect the source changes that are produced. (19650497)

Xcode Templates

- The UICollectionViewController file template has not been fully updated to the latest Swift language changes. In this seed, a newly created file for that class will have one build error. The error has a fix-it that automatically applies the appropriate fix. (19738761)
- The Finder Sync app extension template has not been fully updated to the latest Swift language changes. In this seed, a newly created Finder Sync extension will have one build error. The error has a fix-it that automatically applies the appropriate fix. (19739705)

WatchKit Projects

- When stopped at a breakpoint in a Watch app, tapping Stop doesn't stop running the app. (18991746)

  Workaround: Tap stop twice.

- Sharing a scheme in a Watch app project prevents the iOS and WatchKit App schemes from being created. (18941832)
- Running an iOS app and a Watch app concurrently in the simulator via Xcode is not supported. (18559453)

  Workaround: Build and run your Watch app, tap your iOS app's icon in the Simulator's home screen, then use Debug > Attach to Process... in Xcode to debug the iOS app.

- When creating a new WatchKit app with a Glance or Notification, the corresponding scheme for the Glance or Notification may have "Main" selected as the Watch Interface, rather than "Glance" or "Dynamic/Static Notification”. (19567626)

    Workaround: Select the correct interface in the Info tab of the Glance or Notification scheme’s Run action.

Debugger

- Under some circumstances, iOS extensions need to be manually enabled. (18603937)

Simulator

- iCloud accounts requiring two factor authentication are not supported in the iOS Simulator. (18522339)

    Workaround: For development, create an account that doesn’t use two factor authentication.