CHANGELOG

Note: *This is in reverse chronological order, so newer entries are added to the top.*

Swift 5.7

- SE-0326:

  It's now possible to infer parameter and result types from the body of a multi-statement closure. The distinction between single- and multi-statement closures has been removed.

  Use of closures becomes less cumbersome by removing the need to constantly specify explicit closure types which sometimes could be pretty large e.g. when there are multiple parameters or a complex tuple result type.

  For example:

  ```
  func map<T>(fn: (Int) -> T) -> T {
    return fn(42)
  }

  func computeResult<U: BinaryInteger>(_: U) -> U { /* processing */ }

  let _ = map {
    if let $0 < 0 {
       // do some processing
    }

    return computeResult($0)
  }
  ```

  The result type of `map` can now be inferred from the body of the trailing closure passed as an argument.

- SE-0345:

  It is now possible to unwrap optional variables with a shorthand syntax that shadows the existing declaration. For example, the following:

  ```
  let foo: String? = "hello world"

  if let foo {
    print(foo) // prints "hello world"
  }
  ```

  is equivalent to:

  ```
  let foo: String? = "hello world"
  ```

```
if let foo = foo {
  print(foo) // prints "hello world"
}
```

- SE-0340:

  It is now possible to make declarations unavailable from use in asynchronous contexts with the `@available(*, noasync)` attribute.

  This is to protect the consumers of an API against undefined behavior that can occur when the API uses thread-local storage, or encourages using thread-local storage, across suspension points, or protect developers against holding locks across suspension points which may lead to undefined behavior, priority inversions, or deadlocks.

- SE-0343:

  Top-level scripts support asynchronous calls.

  Using an `await` by calling an asynchronous function or accessing an isolated variable transitions the top-level to an asynchronous context. As an asynchronous context, top-level variables are `@MainActor`-isolated and the top-level is run on the `@MainActor`.

  Note that the transition affects function overload resolution and starts an implicit run loop to drive the concurrency machinery.

  Unmodified scripts are not affected by this change unless `-warn-concurrency` is passed to the compiler invocation. With `-warn-concurrency`, variables in the top-level are isolated to the main actor and the top-level context is isolated to the main actor, but is not an asynchronous context.

- SE-0336:

  It is now possible to declare `distributed actor` and `distributed func`s inside of them.

  Distributed actors provide stronger isolation guarantees than "local" actors, and enable additional checks to be made on return types and parameters of distributed methods, e.g. checking if they conform to `Codable`. Distributed methods can be called on "remote" references of distributed actors, turning those invocations into remote procedure calls, by means of pluggable and user extensible distributed actor system implementations.

  Swift does not provide any specific distributed actor system by itself, however, packages in the ecosystem fulfil the role of providing those implementations.

```
distributed actor Greeter {
  var greetingsSent = 0

  distributed func greet(name: String) -> String {
    greetingsSent += 1
    return "Hello, \(name)!"
  }
}

func talkTo(greeter: Greeter) async throws {
  // isolation of distributed actors is stronger, it is impossible to refer to
  // any stored properties of distributed actors from outside of them:
  greeter.greetingsSent // distributed actor-isolated property 'name' can not be access

  // remote calls are implicitly throwing and async,
  // to account for the potential networking involved:
  let greeting = try await greeter.greet(name: "Alice")
  print(greeting) // Hello, Alice!
}
```

- The compiler now emits a warning when a non-final class conforms to a protocol that imposes a same-type requirement between `Self` and an associated type. This is because such a requirement makes the conformance unsound for subclasses.

  For example, Swift 5.6 would allow the following code, which at runtime would construct an instance of `C` and not `SubC` as expected:

```
protocol P {
  associatedtype A : Q where Self == Self.A.B
}

protocol Q {
  associatedtype B

  static func getB() -> B
}

class C : P {
  typealias A = D
}

class D : Q {
  typealias B = C

  static func getB() -> C { return C() }
}

extension P {
  static func getAB() -> Self {
    // This is well-typed, because `Self.A.getB()` returns
    // `Self.A.B`, which is equivalent to `Self`.
    return Self.A.getB()
  }
}

class SubC : C {}

// P.getAB() declares a return type of `Self`, so it should
// return `SubC`, but it actually returns a `C`.
print(SubC.getAB())
```

To make the above example correct, either the class `C` needs to become `final` (in which case `SubC` cannot be declared) or protocol `P` needs to be re-designed to not include the same-type requirement `Self == Self.A.B`.

- SE-0341:

  Opaque types can now be used in the parameters of functions and subscripts, when they provide a shorthand syntax for the introduction of a generic parameter. For example, the following:

  ```
  func horizontal(_ v1: some View, _ v2: some View) -> some View {
    HStack {
      v1
      v2
    }
  }
  ```

  is equivalent to

  ```
  func horizontal<V1: View, V2: View>(_ v1: V1, _ v2: V2) -> some View {
    HStack {
      v1
      v2
    }
  }
  ```

  With this, `some` in a parameter type provides a generalization where the caller chooses the parameter's type as well as its value, whereas `some` in the result type provides a generalization where the callee chooses the resulting type and value.

- The compiler now correctly emits warnings for more kinds of expressions where a protocol conformance is used and may be unavailable at runtime. Previously, member reference expressions and type erasing expressions that used potentially unavailable conformances were not diagnosed, leading to potential crashes at runtime.

  ```
  struct Pancake {}
  protocol Food {}
  ```

```
extension Food {
  var isGlutenFree: Bool { false }
}

@available(macOS 12.0, *)
extension Pancake: Food {}

@available(macOS 11.0, *)
func eatPancake(_ pancake: Pancake) {
  if (pancake.isGlutenFree) { // warning: conformance of 'Pancake' to 'Food' is only av
    eatFood(pancake) // warning: conformance of 'Pancake' to 'Food' is only available i
  }
}

func eatFood(_ food: Food) {}
```

- SE-0328:

  Opaque types (expressed with 'some') can now be used in structural positions within a result type, including having multiple opaque types in the same result. For example:

```
func getSomeDictionary() -> [some Hashable: some Codable] {
  return [ 1: "One", 2: "Two" ]
}
```

Swift 5.6

- SE-0327:

  In Swift 5 mode, a warning is now emitted if the default-value expression of an instance-member property requires global-actor isolation. For example:

```
@MainActor
func partyGenerator() -> [PartyMember] { fatalError("todo") }

class Party {
  @MainActor var members: [PartyMember] = partyGenerator()
  //                                      ^~~~~~~~~~~~~~~~
  // warning: expression requiring global actor 'MainActor' cannot
  //          appear in default-value expression of property 'members'
}
```

  Previously, the isolation granted by the type checker matched the isolation of the property itself, but at runtime that is not guaranteed. In Swift 6, such default-value expressions will become an error if they require isolation.

- Actor isolation checking now understands that `defer` bodies share the isolation of their enclosing function.

  ```swift
  // Works on global actors
  @MainActor
  func runAnimation(controller: MyViewController) async {
    controller.hasActiveAnimation = true
    defer { controller.hasActiveAnimation = false }

    // do the animation here...
  }

  // Works on actor instances
  actor OperationCounter {
    var activeOperationCount = 0

    func operate() async {
      activeOperationCount += 1
      defer { activeOperationCount -= 1 }

      // do work here...
    }
  }
  ```

- SE-0335:

  Swift now allows existential types to be explicitly written with the `any` keyword, creating a syntactic distinction between existential types and protocol conformance constraints. For example:

  ```swift
  protocol P {}

  func generic<T>(value: T) where T: P {
    ...
  }

  func existential(value: any P) {
     ...
  }
  ```

- SE-0337:

  Swift now provides an incremental migration path to data race safety, allowing APIs to adopt concurrency without breaking their clients that themselves have not adopted concurrency. An existing declaration can introduce concurrency-related annotations (such as making its closure parameters `@Sendable`) and use the `@preconcurrency` attribute to maintain its behavior for clients who have not themselves adopted concurrency:

```
// module A
@preconcurrency func runOnSeparateTask(_ workItem: @Sendable () -> Void)

// module B
import A

class MyCounter {
  var value = 0
}

func doesNotUseConcurrency(counter: MyCounter) {
  runOnSeparateTask {
    counter.value += 1 // no warning, because this code hasn't adopted concurrency
  }
}

func usesConcurrency(counter: MyCounter) async {
  runOnSeparateTask {
    counter.value += 1 // warning: capture of non-Sendable type 'MyCounter'
  }
}
```

One can enable warnings about data race safety within a module with the `-warn-concurrency` compiler option. When using a module that does not yet provide `Sendable` annotations, one can suppress warnings for types from that module by marking the import with `@preconcurrency`:

```
/// module C
public struct Point {
  public var x, y: Double
}

// module D
@preconcurrency import C

func centerView(at location: Point) {
  Task {
    await mainView.center(at: location) // no warning about non-Sendable 'Point' becaus
  }
}
```

- SE-0302:

  Swift will now produce warnings to indicate potential data races when non-`Sendable` types are passed across actor or task boundaries. For example:

  ```
  class MyCounter {
    var value = 0
  ```

```
}

func f() -> MyCounter {
  let counter = MyCounter()
  Task {
    counter.value += 1  // warning: capture of non-Sendable type 'MyCounter'
  }
  return counter
}
```

- SE-0331:

  The conformance of the unsafe pointer types (e.g., `UnsafePointer`, `UnsafeMutableBufferPointer`) to the `Sendable` protocols has been removed, because pointers cannot safely be transferred across task or actor boundaries.

- References to `Self` or so-called "`Self` requirements" in the type signatures of protocol members are now correctly detected in the parent of a nested type. As a result, protocol members that fall under this overlooked case are no longer available on values of protocol type:

  ```
  struct Outer<T> {
    struct Inner {}
  }

  protocol P {}
  extension P {
    func method(arg: Outer<Self>.Inner) {}
  }

  func test(p: P) {
    // error: 'method' has a 'Self' requirement and cannot be used on a value of
    // protocol type (use a generic constraint instead).
    _ = p.method
  }
  ```

- SE-0324:

  Relax diagnostics for pointer arguments to C functions. The Swift compiler now accepts limited pointer type mismatches when directly calling functions imported from C as long as the C language allows those pointer types to alias. Consequently, any Swift `Unsafe[Mutable]Pointer<T>` or `Unsafe[Mutable]RawPointer` may be passed to C function arguments declared as `[signed|unsigned] char *`. Swift `Unsafe[Mutable]Pointer<T>` can also be passed to C function arguments with an integer type that differs from `T` only in its signedness.

  For example, after importing a C function declaration:

```
long long decode_int64(const char *ptr_to_int64);
```

Swift can now directly pass a raw pointer as the function argument:

```
func decodeAsInt64(data: Data) -> Int64 {
    data.withUnsafeBytes { (bytes: UnsafeRawBufferPointer) in
        decode_int64(bytes.baseAddress!)
    }
}
```

- SE-0322:

  The standard library now provides a new operation `withUnsafeTemporaryAllocation` which provides an efficient temporarily allocation within a limited scope, which will be optimized to use stack allocation when possible.

- SE-0320:

  Dictionaries with keys of any type conforming to the new protocol `CodingKeyRepresentable` can now be encoded and decoded. Formerly, encoding and decoding was limited to keys of type `String` or `Int`.

- SE-0315:

  Type expressions and annotations can now include "type placeholders" which directs the compiler to fill in that portion of the type according to the usual type inference rules. Type placeholders are spelled as an underscore ("`_`") in a type name. For instance:

  ```
  // This is OK--the compiler can infer the key type as `Int`.
  let dict: [_: String] = [0: "zero", 1: "one", 2: "two"]
  ```

- SE-0290:

  It is now possible to write inverted availability conditions by using the new `#unavailable` keyword:

  ```
  if #unavailable(iOS 15.0) {
      // Old functionality
  } else {
      // iOS 15 functionality
  }
  ```

Add new entries to the top of this section, not here!

Swift 5.5

2021-09-20 (Xcode 13.0)

- SE-0323:

  The main function is executed with `MainActor` isolation applied, so functions and variables with `MainActor` isolation may be called and modified synchronously from the main function. If the main function is annotated with a global actor explicitly, it must be the main actor or an error is emitted. If no global actor annotation is present, the main function is implicitly run on the main actor.

The main function is executed synchronously up to the first suspension point. Any tasks enqueued by initializers in Objective-C or C++ will run after the main function runs to the first suspension point. At the suspension point, the main function suspends and the tasks are executed according to the Swift concurrency mechanisms.

- SE-0313:

  Parameters of actor type can be declared as `isolated`, which means that they represent the actor on which that code will be executed. `isolated` parameters extend the actor-isolated semantics of the `self` parameter of actor methods to arbitrary parameters. For example:

  ```
  actor MyActor {
    func f() { }
  }

  func g(actor: isolated MyActor) {
    actor.f()   // okay, this code is always executing on "actor"
  }

  func h(actor: MyActor) async {
    g(actor: actor)        // error, call must be asynchronous
    await g(actor: actor)  // okay, hops to "actor" before calling g
  }
  ```

  The `self` parameter of actor methods are implicitly `isolated`. The `nonisolated` keyword makes the `self` parameter no longer `isolated`.

- SR-14731:

  The compiler now correctly rejects the application of generic arguments to the special `Self` type:

  ```
  struct Box<T> {
    // previously interpreted as a return type of Box<T>, ignoring the <Int> part;
    // now we diagnose an error with a fix-it suggesting replacing `Self` with `Box`
    static func makeBox() -> Self<Int> {...}
  }
  ```

- SR-14878:

  The compiler now correctly rejects `@available` annotations on enum cases with associated values with an OS version newer than the current deployment target:

```swift
@available(macOS 12, *)
public struct Crayon {}

public enum Pen {
  case pencil

  @available(macOS 12, *)
  case crayon(Crayon)
}
```

While this worked with some examples, there is no way for the Swift runtime to perform the requisite dynamic layout needed to support this in general, which could cause crashes at runtime.

Note that conditional availability on stored properties in structs and classes is not supported for similar reasons; it was already correctly detected and diagnosed.

- SE-0311:

  Task local values can be defined using the new `@TaskLocal` property wrapper. Such values are carried implicitly by the task in which the binding was made, as well as any child-tasks, and unstructured task created from the tasks context.

```swift
struct TraceID {
  @TaskLocal
  static var current: TraceID?
}

func printTraceID() {
  if let traceID = TraceID.current {
    print("\(traceID)")
  } else {
    print("nil")
  }
}

func run() async {
  printTraceID()    // prints: nil
  TraceID.$current.withValue("1234-5678") {
    printTraceID()  // prints: 1234-5678
    inner()         // prints: 1234-5678
  }
  printTraceID()    // prints: nil
}

func inner() {
  // if called from a context in which the task-local value
  // was bound, it will print it (or 'nil' otherwise)
  printTraceID()
}
```

- SE-0316:

  A type can be defined as a global actor. Global actors extend the notion of actor isolation outside of a single actor type, so that global state (and the functions that access it) can benefit from actor isolation, even if the state and functions are scattered across many different types, functions and modules. Global actors make it possible to safely work with global variables in a concurrent program, as well as modeling other global program constraints such as code that must only execute on the “main thread” or “UI thread”. A new global actor can be defined with the `globalActor` attribute:

  ```
  @globalActor
  struct DatabaseActor {
    actor ActorType { }

    static let shared: ActorType = ActorType()
  }
  ```

  Global actor types can be used as custom attributes on various declarations, which ensures that those declarations are only accessed on the actor described by the global actor’s `shared` instance. For example:

  ```
  @DatabaseActor func queryDB(query: Query) throws -> QueryResult

  func runQuery(queryString: String) async throws -> QueryResult {
    let query = try Query(parsing: queryString)
    return try await queryDB(query: query) // 'await' because this implicitly hops to Dat
  }
  ```

  The concurrency library defines one global actor, `MainActor`, which represents the main thread of execution. It should be used for any code that must execute on the main thread, e.g., for updating UI.

- SE-0313:

  Declarations inside an actor that would normally be actor-isolated can explicitly become non-isolated using the `nonisolated` keyword. Non-isolated declarations can be used to conform to synchronous protocol requirements:

  ```
  actor Account: Hashable {
    let idNumber: Int
    var balance: Double
  ```

```
  nonisolated func hash(into hasher: inout Hasher) { // okay, non-isolated satisfies sy
    hasher.combine(idNumber) // okay, can reference idNumber from outside the let
    hasher.combine(balance) // error: cannot synchronously access actor-isolated proper
  }
}
```

- SE-0300:

  Async functions can now be suspended using the `withUnsafeContinuation` and `withUnsafeThrowingContinuation` functions. These both take a closure, and then suspend the current async task, executing that closure with a continuation value for the current task. The program must use that continuation at some point in the future to resume the task, passing in a value or error, which then becomes the result of the `withUnsafeContinuation` call in the resumed task.

- Type names are no longer allowed as an argument to a subscript parameter that expects a metatype type

  ```
  struct MyValue {
  }

  struct MyStruct {
    subscript(a: MyValue.Type) -> Int { get { ... } }
  }

  func test(obj: MyStruct) {
    let _ = obj[MyValue]
  }
  ```

  Accepting subscripts with `MyValue` as an argument was an oversight because `MyValue` requires explicit `.self` to reference its metatype, so correct syntax would be to use `obj[MyValue.self]`.

- SE-0310:

  Read-only computed properties and subscripts can now define their `get` accessor to be `async` and/or `throws`, by writing one or both of those keywords between the `get` and `{`. Thus, these members can now make asynchronous calls or throw errors in the process of producing a value:

  ```
  class BankAccount: FinancialAccount {
    var manager: AccountManager?

    var lastTransaction: Transaction {
      get async throws {
        guard manager != nil else { throw BankError.notInYourFavor }
        return await manager!.getLastTransaction()
      }
    }
  ```

```
  subscript(_ day: Date) -> [Transaction] {
    get async {
      return await manager?.getTransactions(onDay: day) ?? []
    }
  }
}

protocol FinancialAccount {
  associatedtype T
  var lastTransaction: T { get async throws }
  subscript(_ day: Date) -> [T] { get async }
}
```

Accesses to such members, like `lastTransaction` above, will require appropriate marking with `await` and/or `try`:

```
extension BankAccount {
  func meetsTransactionLimit(_ limit: Amount) async -> Bool {
    return try! await self.lastTransaction.amount < limit
    //                    ^~~~~~~~~~~~~~~~ this access is async & throws
  }
}


func hadWithdrawlOn(_ day: Date, from acct: BankAccount) async -> Bool {
  return await !acct[day].allSatisfy { $0.amount >= Amount.zero }
  //            ^~~~~~~~~ this access is async
}
```

- SE-0306:

  Swift 5.5 includes support for actors, a new kind of type that isolates its instance data to protect it from concurrent access. Accesses to an actor's instance declarations from outside the must be asynchronous:

```
actor Counter {
  var value = 0

  func increment() {
    value = value + 1
  }
}

func useCounter(counter: Counter) async {
  print(await counter.value) // interaction must be async
  await counter.increment()  // interaction must be async
}
```

- The determination of whether a call to a `rethrows` function can throw now considers default arguments of `Optional` type.

  In Swift 5.4, such default arguments were ignored entirely by `rethrows` checking. This meant that the following example was accepted:

  ```
  func foo(_: (() throws -> ())? = nil) rethrows {}
  foo()  // no 'try' needed
  ```

  However, it also meant that the following was accepted, even though the call to `foo()` can throw and the call site is not marked with `try`:

  ```
  func foo(_: (() throws -> ())? = { throw myError }) rethrows {}
  foo()  // 'try' *should* be required here
  ```

  The new behavior is that the first example is accepted because the default argument is syntactically written as `nil`, which is known not to throw. The second example is correctly rejected, on account of missing a `try` since the default argument *can* throw.

- SE-0293:

  Property wrappers can now be applied to function and closure parameters:

  ```
  @propertyWrapper
  struct Wrapper<Value> {
    var wrappedValue: Value

    var projectedValue: Self { return self }

    init(wrappedValue: Value) { ... }

    init(projectedValue: Self) { ... }
  }

  func test(@Wrapper value: Int) {
    print(value)
    print($value)
    print(_value)
  }

  test(value: 10)

  let projection = Wrapper(wrappedValue: 10)
  test($value: projection)
  ```

  The call-site can pass a wrapped value or a projected value, and the property wrapper will be initialized using `init(wrappedValue:)` or `init(projectedValue:)`, respectively.

- SE-0299:

It is now possible to use leading-dot syntax in generic contexts to access static members of protocol extensions where `Self` is constrained to a fully concrete type:

```
public protocol ToggleStyle { ... }

public struct DefaultToggleStyle: ToggleStyle { ... }

extension ToggleStyle where Self == DefaultToggleStyle {
  public static var `default`: Self { .init() }
}

struct Toggle {
  func applyToggle<T: ToggleStyle>(_ style: T) { ... }
}

Toggle(...).applyToggle(.default)
```

- Whenever a reference to `Self` does not impede the usage of a protocol as a value type, or a protocol member on a value of protocol type, the same is now true for references to `[Self]` and `[Key : Self]`:

```
protocol Copyable {
  func copy() -> Self
  func copy(count: Int) -> [Self]
}

func test(c: Copyable) {
  let copy: Copyable = c.copy() // OK
  let copies: [Copyable] = c.copy(count: 5) // also OK
}
```

- SE-0296:

  Asynchronous programming is now natively supported using async/await. Asynchronous functions can be defined using `async`:

```
func loadWebResource(_ path: String) async throws -> Resource { ... }
func decodeImage(_ r1: Resource, _ r2: Resource) async throws -> Image
func dewarpAndCleanupImage(_ i : Image) async -> Image
```

  Calls to `async` functions may suspend, meaning that they give up the thread on which they are executing and will be scheduled to run again later. The potential for suspension on asynchronous calls requires the `await` keyword, similarly to the way in which `try` acknowledges a call to a `throws` function:

```
func processImageData() async throws -> Image {
  let dataResource  = try await loadWebResource("dataprofile.txt")
  let imageResource = try await loadWebResource("imagedata.dat")
  let imageTmp      = try await decodeImage(dataResource, imageResource)
  let imageResult   = await dewarpAndCleanupImage(imageTmp)
  return imageResult
}
```

- The 'lazy' keyword now works in local contexts, making the following valid:

  ```
  func test(useIt: Bool) {
    lazy var result = getPotentiallyExpensiveResult()
    if useIt {
      doIt(result)
    }
  }
  ```

- SE-0297:

  An Objective-C method that delivers its results asynchronously via a completion handler block will be translated into an `async` method that directly returns the result (or throws). For example, the following Objective-C method from CloudKit:

  ```
  - (void)fetchShareParticipantWithUserRecordID:(CKRecordID *)userRecordID
      completionHandler:(void (^)(CKShareParticipant * _Nullable, NSError * _Nullable))co
  ```

  will be translated into an `async throws` method that returns the participant instance:

  ```
  func fetchShareParticipant(
      withUserRecordID userRecordID: CKRecord.ID
  ) async throws -> CKShare.Participant
  ```

  Swift callers can invoke this `async` method within an `await` expression:

  ```
  guard let participant = try? await container.fetchShareParticipant(withUserRecordID: us
      return nil
  }
  ```

- SE-0298:

  The "for" loop can be used to traverse asynchronous sequences in asynchronous code:

  ```
  for try await line in myFile.lines() {
    // Do something with each line
  }
  ```

  Asynchronous for loops use asynchronous sequences, defined by the protocol `AsyncSequence` and its corresponding `AsyncIterator`.

Add new entries to the top of this section, not here!

Swift 5.4

2021-04-26 (Xcode 12.5)

- Protocol conformance checking now considers `where` clauses when evaluating if a `typealias` is a suitable witness for an associated type requirement. The following code is now rejected:

  ```
  protocol Holder {
    associatedtype Contents
  }

  struct Box<T> : Holder {}
  // error: type 'Box<T>' does not conform to protocol 'Holder'

  extension Box where T : Hashable {
    typealias Contents = T
  }
  ```

  In most cases, the compiler would either crash or produce surprising results when making use of a `typealias` with an unsatisfied `where` clause, but it is possible that some previously-working code is now rejected. In the above example, the conformance can be fixed in one of various ways:

  1) making it conditional (moving the `: Holder` from the definition of `Box` to the extension)
  2) moving the `typealias` from the extension to the type itself
  3) relaxing the `where` clause on the extension

- Availability checking now rejects protocols that refine less available protocols. Previously, this was accepted by the compiler but could result in linker errors or runtime crashes:

  ```
  @available(macOS 11, *)
  protocol Base {}

  protocol Bad : Base {}
  // error: 'Base' is only available in macOS 11 or newer

  @available(macOS 11, *)
  protocol Good : Base {} // OK
  ```

- The `@available` attribute is no longer permitted on generic parameters, where it had no effect:

  ```
  struct Bad<@available(macOS 11, *) T> {}
  // error: '@available' attribute cannot be applied to this declaration

  struct Good<T> {} // equivalent
  ```

- If a type is made to conform to a protocol via an extension, the availability of the extension is now taken into account when forming generic types that use this protocol conformance. For example, consider a `Box` type whose conformance to `Hashable` uses features only available on macOS 11:

  ```
  public struct Box {}

  @available(macOS 11, *)
  extension Box : Hashable {
    func hash(into: inout Hasher) {
      // call some new API to hash the value...
    }
  }

  public func findBad(_: Set<Box>) -> Box {}
  // warning: conformance of 'Box' to 'Hashable' is only available in macOS 11 or newer

  @available(macOS 11, *)
  public func findGood(_: Set<Box>) -> Box {} // OK
  ```

  In the above code, it is not valid for `findBad()` to take a `Set<Box>`, since `Set` requires that its element type conform to `Hashable`; however the conformance of `Box` to `Hashable` is not available prior to macOS 11.

  Note that using an unavailable protocol conformance is a warning, not an error, to avoid potential source compatibility issues. This is because it was technically possible to write code in the past that made use of unavailable protocol conformances but worked anyway, if the optimizer had serendipitously eliminated all runtime dispatch through this conformance, or the code in question was entirely unreachable at runtime.

  Protocol conformances can also be marked as completely unavailable or deprecated, by placing an appropriate `@available` attribute on the extension:

  ```
  @available(*, unavailable, message: "Not supported anymore")
  extension Box : Hashable {}

  @available(*, deprecated, message: "Suggest using something else")
  extension Box : Hashable {}
  ```

  If a protocol conformance is defined on the type itself, it inherits availability from the type. You can move the protocol conformance to an extension if you need it to have narrower availability than the type.

- When `swift` is run with no arguments, it starts a REPL (read eval print loop) that uses LLDB. The compiler also had a second REPL implementation, known as the "integrated REPL", formerly accessible by running `swift -frontend -repl`. The "integrated REPL" was only intended for use by compiler developers, and has now been removed.

Note that this does not take away the ability to put Swift code in a script and run it with `swift myScript.swift`. This so-called “script mode” is distinct from the integrated REPL, and continues to be supported.

- Property wrappers now work in local contexts, making the following valid:

  ```
  @propertyWrapper
  struct Wrapper<T> {
    var wrappedValue: T
  }

  func test() {
    @Wrapper var value = 10
  }
  ```

- SR-10069:

  Function overloading now works in local contexts, making the following valid:

  ```
  func outer(x: Int, y: String) {
    func doIt(_: Int) {}
    func doIt(_: String) {}

    doIt(x) // calls the first 'doIt(_:)' with an Int value
    doIt(y) // calls the second 'doIt(_:)' with a String value
  }
  ```

- SE-0284:

  Functions, subscripts, and initializers may now have more than one variadic parameter, as long as all parameters which follow variadic parameters are labeled. This makes declarations like the following valid:

  ```
  func foo(_ a: Int..., b: Double...) { }

  struct Bar {
    subscript(a: Int..., b b: Int...) -> [Int] { a + b }

    init(a: String..., b: Float...) { }
  }
  ```

- SE-0287:

  Implicit member expressions now support chains of member accesses, making the following valid:

  ```
  let milky: UIColor = .white.withAlphaComponent(0.5)
  let milky2: UIColor = .init(named: "white")!.withAlphaComponent(0.5)
  let milkyChance: UIColor? = .init(named: "white")?.withAlphaComponent(0.5)
  ```

As is the case with the existing implicit member expression syntax, the resulting type of the chain must be the same as the (implicit) base, so it is not well-formed to write:

```swift
let cgMilky: CGColor = .white.withAlphaComponent(0.5).cgColor
```

(Unless, of course, appropriate `white` and `withAlphaComponent` members were defined on `CGColor`.)

Members of a "chain" can be properties, method calls, subscript accesses, force unwraps, or optional chaining question marks. Furthermore, the type of each member along the chain is permitted to differ (again, as long as the base of the chain matches the resulting type) meaning the following successfully typechecks:

```swift
struct Foo {
  static var foo = Foo()
  static var bar = Bar()

  var anotherFoo: Foo { Foo() }
  func getFoo() -> Foo { Foo() }
  var optionalFoo: Foo? { Foo() }
  subscript() -> Foo { Foo() }
}

struct Bar {
  var anotherFoo = Foo()
}

let _: Foo? = .bar.anotherFoo.getFoo().optionalFoo?.optionalFoo![]
```

Add new entries to the top of this section, not here!

Swift 5.3

2020-09-16 (Xcode 12.0)

- SE-0279 & SE-0286:

  Trailing closure syntax has been extended to allow additional labeled closures to follow the initial unlabeled closure:

  ```swift
  // Single trailing closure argument
  UIView.animate(withDuration: 0.3) {
    self.view.alpha = 0
  }
  // Multiple trailing closure arguments
  UIView.animate(withDuration: 0.3) {
    self.view.alpha = 0
  } completion: { _ in
  ```

```
  self.view.removeFromSuperview()
}
```

Additionally, trailing closure arguments now match the appropriate parameter according to a forward-scan rule (as opposed to the previous backward-scan rule):

```
func takesClosures(first: () -> Void, second: (Int) -> Void = { _ in }) {}

takesClosures {
  print("First")
}
```

In the above example, the trailing closure argument matches parameter `first`, whereas pre-Swift-5.3 it would have matched `second`. In order to ease the transition to this new rule, cases in which the forward-scan and backward-scan match a single trailing closure to different parameters, the backward-scan result is preferred and a warning is emitted. This is expected to be upgraded to an error in the next major version of Swift.

- SR-7083:

  Property observers such as `willSet` and `didSet` are now supported on `lazy` properties:

```
class C {
  lazy var property: Int = 0 {
    willSet { print("willSet called!") } // Okay
    didSet { print("didSet called!") } // Okay
  }
}
```

  Note that the initial value of the property will be forced and made available as the `oldValue` for the `didSet` observer, if the property hasn't been accessed yet.

```
class C {
  lazy var property: Int = 0 {
    didSet { print("Old value: ", oldValue) }
  }
}

let c = C()
c.property = 1 // Prints 'Old value: 0'
```

  This could have side-effects, for example if the lazy property's initializer is doing other work.

- SR-11700:

Exclusivity violations within code that computes the `default` argument during Dictionary access are now diagnosed.

```
struct Container {
   static let defaultKey = 0

   var dictionary = [defaultKey:0]

   mutating func incrementValue(at key: Int) {
     dictionary[key, default: dictionary[Container.defaultKey]!] += 1
   }
}
// error: overlapping accesses to 'self.dictionary', but modification requires exclusi
//     dictionary[key, default: dictionary[Container.defaultKey]!] += 1
//     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// note: conflicting access is here
//     dictionary[key, default: dictionary[Container.defaultKey]!] += 1
//                              ~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~
```

The exclusivity violation can be avoided by precomputing the `default` argument using a local variable.

```
struct Container {
  static let defaultKey = 0

  var dictionary = [defaultKey:0]

  mutating func incrementValue(at key: Int) {
    let defaultValue = dictionary[Container.defaultKey]!
    dictionary[key, default: defaultValue] += 1
  }
}
// No error.
```

- SE-0268:

  A `didSet` observer which does not refer to the `oldValue` in its body or does not explicitly request it by placing it in the parameter list (i.e. `didSet(oldValue)`) will no longer trigger a call to the property getter to fetch the `oldValue`.

```
class C {
  var value: Int = 0 {
    didSet { print("didSet called!") }
  }
}

let c = C()
// This does not trigger a call to the getter for 'value'
```

```
// because the 'didSet' observer on 'value' does not
// refer to the 'oldValue' in its body, which means
// the 'oldValue' does not need to be fetched.
c.value = 1
```

- SE-0276:

  Catch clauses in a `do-catch` statement can now include multiple patterns in a comma-separated list. The body of a `catch` clause will be executed if a thrown error matches any of its patterns.

  ```
  do {
    try performTask()
  } catch TaskError.someFailure(let msg),
          TaskError.anotherFailure(let msg) {
    showMessage(msg)
  }
  ```

- SE-0280:

  Enum cases can now satisfy static protocol requirements. A static get-only property of type `Self` can be witnessed by an enum case with no associated values and a static function with arguments and returning `Self` can be witnessed by an enum case with associated values.

  ```
  protocol P {
    static var foo: Self { get }
    static func bar(value: Int) -> Self
  }

  enum E: P {
    case foo // matches 'static var foo'
    case bar(value: Int) // matches 'static func bar(value:)'
  }
  ```

- SE-0267:

  Non-generic members that support a generic parameter list, including nested type declarations, are now allowed to carry a contextual `where` clause against outer generic parameters. Previously, such declarations could only be expressed by placing the member inside a dedicated constrained extension.

  ```
  struct Box<Wrapped> {
    func boxes() -> [Box<Wrapped.Element>] where Wrapped: Sequence { ... }
  }
  ```

  Since contextual `where` clauses are effectively visibility constraints, overrides adopting this feature must be at least as visible as the overridden method. In practice, this implies any instance of `Derived` that can access `Base.foo` must also be able to access `Derived.foo`.

“‘swift class Base { func foo() where T == Int { … } }

class Derived: Base { // OK, has broader visibility than override func foo() where U: Equatable { … } }

- SR-75:

  Unapplied references to protocol methods are now supported. Previously this only worked for methods defined in structs, enums and classes.

  ```
  protocol Cat {
    func play(catToy: Toy)
  }

  let fn = Cat.play(catToy:)
  fn(myCat)(myToy)
  ```

- SE-0266:

  Enumerations with no associated values, or only `Comparable` associated values, can opt-in to synthesized `Comparable` conformance by declaring conformance to the `Comparable` protocol. The synthesized implementation orders the cases first by case-declaration order, and then by lexicographic order of the associated values (if any).

  ```
  enum Foo: Comparable {
    case a(Int), b(Int), c
  }

  // .a(0) < .a(1) < .b(0) < .b(1) < .c
  ```

- SE-0269:

  When an escaping closure explicitly captures `self` in its capture list, the use of implicit `self` is enabled within that closure. This means that the following code is now valid:

  ```
  func doStuff(_ stuff: @escaping () -> Void) {}

  class C {
    var x = 0

    func method() {
      doStuff { [self] in
        x += 1
      }
    }
  }
  ```

  This proposal also introduces new diagnostics for inserting `self` into the closure’s capture list in addition to the existing ‘use `self.` explicitly’ fix-it.

Swift 5.2

2020-03-24 (Xcode 11.4)

- SR-11841:

  When chaining calls to `filter(_:)` on a lazy sequence or collection, the filtering predicates will now be called in the same order as eager filters.

  ```
  let evens = (1...10).lazy
      .filter { $0.isMultiple(of: 2) }
      .filter { print($0); return true }
  _ = evens.count
  // Prints 2, 4, 6, 8, and 10 on separate lines
  ```

  Previously, the predicates were called in reverse order.

- SR-2790:

  The compiler will now emit a warning when attempting to pass a temporary pointer argument produced from an array, string, or inout argument to a parameter which is known to escape it. This includes the various initializers for the `UnsafePointer`/`UnsafeBufferPointer` family of types, as well as memberwise initializers.

  ```
  struct S {
    var ptr: UnsafePointer<Int8>
  }

  func foo() {
    var i: Int8 = 0
    let ptr = UnsafePointer(&i)
    // warning: initialization of 'UnsafePointer<Int8>' results in a
    // dangling pointer

    let s1 = S(ptr: [1, 2, 3])
    // warning: passing '[Int8]' to parameter, but argument 'ptr' should be a
    // pointer that outlives the call to 'init(ptr:)'

    let s2 = S(ptr: "hello")
    // warning: passing 'String' to parameter, but argument 'ptr' should be a
    // pointer that outlives the call to 'init(ptr:)'
  }
  ```

  All 3 of the above examples are unsound because each argument produces a temporary pointer only valid for the duration of the call they are passed to. Therefore the returned value in each case references a dangling pointer.

- SR-2189:

The compiler now supports local functions whose default arguments capture values from outer scopes.

```swift
func outer(x: Int) -> (Int, Int) {
  func inner(y: Int = x) -> Int {
    return y
  }

  return (inner(), inner(y: 0))
}
```

- SR-11429:

  The compiler will now correctly strip argument labels from function references used with the `as` operator in a function call. As a result, the `as` operator can now be used to disambiguate a call to a function with argument labels.

  ```swift
  func foo(x: Int) {}
  func foo(x: UInt) {}

  (foo as (Int) -> Void)(5)  // Calls foo(x: Int)
  (foo as (UInt) -> Void)(5) // Calls foo(x: UInt)
  ```

  Previously this was only possible for functions without argument labels.

  This change also means that a generic type alias can no longer be used to preserve the argument labels of a function reference through the `as` operator. The following is now rejected:

  ```swift
  typealias Magic<T> = T
  func foo(x: Int) {}
  (foo as Magic)(x: 5) // error: Extraneous argument label 'x:' in call
  ```

  The function value must instead be called without argument labels:

  ```swift
  (foo as Magic)(5)
  ```

- SR-11298:

  A class-constrained protocol extension, where the extended protocol does not impose a class constraint, will now infer the constraint implicitly.

  ```swift
  protocol Foo {}
  class Bar: Foo {
    var someProperty: Int = 0
  }

  // Even though 'Foo' does not impose a class constraint, it is automatically
  // inferred due to the Self: Bar constraint.
  extension Foo where Self: Bar {
    var anotherProperty: Int {
  ```

```
    get { return someProperty }
    // As a result, the setter is now implicitly nonmutating, just like it would
    // be if 'Foo' had a class constraint.
    set { someProperty = newValue }
  }
}
```

- SE-0253:

  Values of types that declare `func callAsFunction` methods can be called like functions. The call syntax is shorthand for applying `func callAsFunction` methods.

  ```
  struct Adder {
    var base: Int
    func callAsFunction(_ x: Int) -> Int {
      return x + base
    }
  }
  var adder = Adder(base: 3)
  adder(10) // returns 13, same as `adder.callAsFunction(10)`
  ```

  - `func callAsFunction` argument labels are required at call sites.
  - Multiple `func callAsFunction` methods on a single type are supported.
  - `mutating func callAsFunction` is supported.
  - `func callAsFunction` works with `throws` and `rethrows`.
  - `func callAsFunction` works with trailing closures.

- SE-0249:

  A `\Root.value` key path expression is now allowed wherever a `(Root) -> Value` function is allowed. Such an expression is implicitly converted to a key path application of `{ $0[keyPath: \Root.value] }`.

  For example:

  ```
  struct User {
    let email: String
    let isAdmin: Bool
  }

  users.map(\.email) // this is equivalent to: users.map { $0[keyPath: \User.email] }
  ```

- SR-4206:

  A method override is no longer allowed to have a generic signature with requirements not imposed by the base method. For example:

  ```
  protocol P {}
  ```

```
class Base {
  func foo<T>(arg: T) {}
}

class Derived: Base {
  override func foo<T: P>(arg: T) {}
}
```

will now be diagnosed as an error.

- SR-6118:

  Subscripts can now declare default arguments:

  ```
  struct Subscriptable {
    subscript(x: Int, y: Int = 0) {
      ...
    }
  }

  let s = Subscriptable()
  print(s[0])
  ```

Swift 5.1

2019-09-20 (Xcode 11.0)

- SR-8974:

  Duplicate tuple element labels are no longer allowed, because it leads to incorrect behavior. For example:

  ```
  let dupLabels: (foo: Int, foo: Int) = (foo: 1, foo: 2)

  enum Foo { case bar(x: Int, x: Int) }
  let f: Foo = .bar(x: 0, x: 1)
  ```

  will now be diagnosed as an error.

  Note: You can still use duplicate argument labels when declaring functions and subscripts, as long as the internal parameter names are different. For example:

  ```
  func foo(bar x: Int, bar y: Int) {}
  subscript(a x: Int, a y: Int) -> Int {}
  ```

- SE-0244:

  Functions can now hide their concrete return type by declaring what protocols it conforms to instead of specifying the exact return type:

```
func makeMeACollection() -> some Collection {
  return [1, 2, 3]
}
```

Code that calls the function can use the interface of the protocol, but does not have visibility into the underlying type.

- SE-0254:

  Subscripts can now be declared `static` or (inside classes) `class`.

- SE-0252:

  The existing `@dynamicMemberLookup` attribute has been extended with a support for strongly-typed keypath implementations:

```
@dynamicMemberLookup
struct Lens<T> {
  let getter: () -> T
  let setter: (T) -> Void

  var value: T {
    get {
      return getter()
    }
    set {
      setter(newValue)
    }
  }

  subscript<U>(dynamicMember keyPath: WritableKeyPath<T, U>) -> Lens<U> {
    return Lens<U>(
        getter: { self.value[keyPath: keyPath] },
        setter: { self.value[keyPath: keyPath] = $0 })
  }
}
```

- SR-8546, SR-9043:

  More thorough checking has been implemented for restrictions around escaping closures capturing `inout` parameters or values of noescape type. While most code should not be affected, there are edge cases where the Swift 5.0 compiler would accept code violating these restrictions. This could result in runtime crashes or silent data corruption.

  An example of invalid code which was incorrectly accepted by the Swift 5.0 compiler is an `@escaping` closure calling a local function which references an `inout` parameter from an outer scope:

```
struct BadCaptureExample {
  var escapingClosure: () -> ()
```

```swift
  mutating func takesInOut(_ x: inout Int) {
    func localFunction() {
      x += 1
    }

    escapingClosure = { localFunction() }
  }
}
```

The compiler now correctly diagnoses the above code by pointing out that the capture of `x` by `localFunction()` is invalid, since `localFunction()` is referenced from an `@escaping` closure.

This also addresses certain cases where the compiler incorrectly diagnosed certain code as invalid, when in fact no violation of restrictions had taken place. For example,

```swift
func takesNoEscape(_ fn: () -> ()) {
  func localFunction() {
    fn()
  }

  { localFunction() }()
}
```

- SR-2672:

  Conversions between tuple types are now fully implemented. Previously, the following would diagnose an error:

  "'swift let values: (Int, Int) = (10, 15) let converted: (Int?, Any) = values

- SE-0242:

  The memberwise initializer for structures now provide default values for variables that hold default expressions.

  ```swift
  struct Dog {
    var name = "Generic dog name"
    var age = 0

    // The synthesized memberwise initializer
    init(name: String = "Generic dog name", age: Int = 0)
  }

  let sparky = Dog(name: "Sparky") // Dog(name: "Sparky", age: 0)
  ```

- SE-0068:

  It is now possible to use `Self` to refer to the innermost nominal type inside struct, enum and class declarations. For example, the two method declarations inside this struct are equivalent:

```
struct Box<Value> {
  func transform1() -> Self { return self }
  func transform2() -> Box<Value> { return self }
}
```

In classes, `Self` is the dynamic type of the `self` value, as before. Existing restrictions on `Self` in declaration types still apply; that is, `Self` can only appear as the return type of a method. However, `Self` can now be used inside the body of a method without limitation.

- SR-7799:

  Enum cases can now be matched against an optional enum without requiring a '?' at the end of the pattern.

  ```
  enum Foo { case zero, one }

  let foo: Foo? = .zero

  switch foo {
    case .zero: break
    case .one: break
    case .none: break
  }
  ```

- SR-9827:

  `weak` and `unowned` stored properties no longer inhibit the automatic synthesis of `Equatable` or `Hashable` conformance.

- SR-2688:

  An `@autoclosure` parameter can now be declared with a typealias type.

  ```
  class Foo {
    typealias FooClosure = () -> String
    func fooFunction(closure: @autoclosure FooClosure) {}
  }
  ```

- SR-7601:

  Methods declared `@objc` inside a class can now return `Self`:

  ```
  class MyClass : NSObject {
    @objc func clone() -> Self { return self }
  }
  ```

- SR-2176:

Assigning '.none' to an optional enum which also has a 'none' case or comparing such an enum with '.none' will now warn. Such expressions create an ambiguity because the compiler chooses Optional.none over Foo.none.

```swift
enum Foo { case none }

// Assigned Optional.none instead of Foo.none
let foo: Foo? = .none
// Comparing with Optional.none instead of Foo.none
let isEqual = foo == .none
```

The compiler will provide a warning along with a fix-it to replace '.none' with 'Optional.none' or 'Foo.none' to resolve the ambiguity.

- Key path expressions can now include references to tuple elements.
- Single-parameter functions accepting values of type `Any` are no longer preferred over other functions.

  ```swift
  func foo(_: Any) { print("Any") }
  func foo<T>(_: T) { print("T") }
  foo(0) // prints "Any" in Swift < 5.1, "T" in Swift 5.1
  ```

- SE-0245:

  `Array` and `ContiguousArray` now have `init(unsafeUninitializedCapacity:initializingWith:)`, which provides access to the array's uninitialized storage.

Swift 5.0

2019-03-25 (Xcode 10.2)

- SE-0235:

  The standard library now contains a `Result` type for manually propagating errors.

  ```swift
  enum Result<Success, Failure: Error> {
      case success(Success)
      case failure(Failure)
  }
  ```

  This type serves a complementary role to that of throwing functions and initializers. Use `Result` in situations where automatic error propagation or `try-catch` blocks are undesirable, such as in asynchronous code or when accumulating the results of successive error-producing operations.

- `Error` now conforms to itself. This allows for the use of `Error` itself as the argument for a generic parameter constrained to `Error`.

- Swift 3 mode has been removed. Supported values for the `-swift-version` flag are `4`, `4.2`, and `5`.

- SE-0228:

  String interpolation has been overhauled to improve its performance, clarity, and efficiency.

  Note that the old `_ExpressibleByStringInterpolation` protocol has been removed; any code making use of this protocol will need to be updated for the new design. An `#if compiler` block can be used to conditionalize code between 4.2 and 5.0, for example:

  ```
  #if compiler(<5.0)
  extension MyType : _ExpressibleByStringInterpolation { ... }
  #else
  extension MyType : ExpressibleByStringInterpolation { ... }
  #endif
  ```

- SE-0213:

  If `T` conforms to one of the `ExpressibleBy*` protocols and `literal` is a literal expression, then `T(literal)` will construct a literal of type `T` using the corresponding protocol, rather than calling a constructor member of `T` with a value of the protocol's default literal type.

  For example, expressions like `UInt64(0xffff_ffff_ffff_ffff)` are now valid, where previously they would overflow the default integer literal type of `Int`.

- SE-0230:

  In Swift 5 mode, `try?` with an expression of Optional type will flatten the resulting Optional, instead of returning an Optional of an Optional.

- SR-5719:

  In Swift 5 mode, `@autoclosure` parameters can no longer be forwarded to `@autoclosure` arguments in another function call. Instead, you must explicitly call the function value with `()`; the call itself is wrapped inside an implicit closure, guaranteeing the same behavior as in Swift 4 mode.

  Example:

  ```
  func foo(_ fn: @autoclosure () -> Int) {}
  func bar(_ fn: @autoclosure () -> Int) {
    foo(fn)   // Incorrect, `fn` can't be forwarded and has to be called
    foo(fn()) // Ok
  }
  ```

- SR-8109:

Single-element labeled tuple expressions, for example `(label: 123)`, were allowed in some contexts but often resulted in surprising, inconsistent behavior that varied across compiler releases. They are now completely disallowed.

Note that single-element labeled *types*, for example `var x: (label: Int)`, have already been prohibited since Swift 3.

- SR-695:

  In Swift 5 mode, a class method returning `Self` can no longer be overridden with a method returning a non-final concrete class type. Such code is not type safe and will need to be updated.

  For example,

  ```
  class Base {
    class func factory() -> Self { ... }
  }

  class Derived : Base {
    class override func factory() -> Derived { ... }
  }
  ```

- In Swift 5 mode, the type of `self` in a convenience initializer of a non-final class is now the dynamic `Self` type, and not the concrete class type.

- SR-5581:

  Protocols can now constrain their conforming types to those that subclasses a given class. Two equivalent forms are supported:

  ```
  protocol MyView : UIView { ... }
  protocol MyView where Self : UIView { ... }
  ```

  Note that Swift 4.2 accepted the second form, but it was not fully implemented and could sometimes crash at compile time or run time.

- SR-631:

  Extension binding now supports extensions of nested types which themselves are defined inside extensions. Previously this could fail with some declaration orders, producing spurious “undeclared type” errors.

- SR-7139:

  Exclusive memory access is now enforced at runtime by default in optimized (`-O`/`-Osize`) builds. Programs that violate exclusivity will trap at runtime with an “overlapping access” diagnostic message. This can be disabled via a command line flag: `-enforce-exclusivity=unchecked`, but doing so may result in undefined behavior.

Runtime violations of exclusivity typically result from simultaneous access of class properties, global variables (including variables in top-level code), or variables captured by escaping closures.

- SE-0216:

  The `@dynamicCallable` attribute enables nominal types to be "callable" via a simple syntactic sugar. The primary use case is dynamic language interoperability.

  Toy example:

  ```
  @dynamicCallable
  struct ToyCallable {
    func dynamicallyCall(withArguments: [Int]) {}
    func dynamicallyCall(withKeywordArguments: KeyValuePairs<String, Int>) {}
  }
  let x = ToyCallable()
  x(1, 2, 3) // desugars to `x.dynamicallyCall(withArguments: [1, 2, 3])`
  x(label: 1, 2) // desugars to `x.dynamicallyCall(withKeywordArguments: ["label": 1, "":
  ```

- SR-7251:

  In Swift 5 mode, attempting to declare a static property with the same name as a nested type is now always correctly rejected. Previously, it was possible to perform such a redeclaration in an extension of a generic type.

  For example:

  ```
  struct Foo<T> {}
  extension Foo {
    struct i {}

    // compiler error: Invalid redeclaration of 'i'
    // (prior to Swift 5, this did not produce an error)
    static var i: Int { return 0 }
  }
  ```

- SR-4248:

  In Swift 5 mode, when casting an optional value to a generic placeholder type, the compiler will be more conservative with the unwrapping of the value. The result of such a cast now more closely matches the result you would get in a non-generic context.

  For example:

  ```
  func forceCast<U>(_ value: Any?, to type: U.Type) -> U {
    return value as! U
  }

  let value: Any? = 42
  ```

```
print(forceCast(value, to: Any.self))
// prints: Optional(42)
// (prior to Swift 5, this would print: 42)

print(value as! Any)
// prints: Optional(42)
```

- SE-0227:

  Key paths now support the `\.self` keypath, which is a `WritableKeyPath` that refers to its entire input value:

  ```
  let id = \Int.self

  var x = 2
  print(x[keyPath: id]) // prints 2
  x[keyPath: id] = 3
  print(x[keyPath: id]) // prints 3
  ```

- SE-0214:

  The `DictionaryLiteral` type has been renamed to `KeyValuePairs`. A typealias preserves the old name for compatibility.

- SR-2608

  Default arguments are now printed in SourceKit-generated interfaces for Swift modules, instead of just using a placeholder `default`.

- `unowned` and `unowned(unsafe)` variables now support Optional types.

- Designated initializers with variadic parameters are now correctly inherited in subclasses.

- Extensions of concrete subclasses of generic classes can now contain `@objc` members.

- Complex recursive type definitions involving classes and generics that would previously cause deadlocks at run time are now fully supported.

- SR-419

  In Swift 5 mode, when setting a property from within its own `didSet` or `willSet` observer, the observer will now only avoid being recursively called if the property is set on `self` (either implicitly or explicitly).

  For example:

  ```
  class Node {
    var children = [Node]()

    var depth: Int = 0 {
      didSet {
  ```

```
        if depth < 0 {
          // Will not recursively call didSet, as setting depth on self (same
          // with `self.depth = 0`).
          depth = 0
        }

        // Will call didSet for each of the children, as we're not setting the
        // property on self (prior to Swift 5, this did not trigger property
        // observers to be called again).
        for child in children {
          child.depth = depth + 1
        }
      }
    }
  }
```

Swift 4.2

2018-09-17 (Xcode 10.0)

- SE-0202

  The standard library now provides a unified set of randomization functionality. Integer types, floating point types, and Bool all introduce a new static method that creates a random value.

  ```
  let diceRoll = Int.random(in: 1 ... 6)
  let randomUnit = Double.random(in: 0 ..< 1)
  let randomBool = Bool.random()
  ```

  There are also additions to select a random element from a collection or shuffle its contents.

  ```
  let greetings = ["hey", "hello", "hi", "hola"]
  let randomGreeting = greetings.randomElement()! // This returns an Optional
  let newGreetings = greetings.shuffled() // ["hola", "hi", "hey", "hello"]
  ```

  Core to the randomization functionality is a new `RandomNumberGenerator` protocol. The standard library defines its own random number generator called `SystemRandomNumberGenerator` which is backed by a secure and thread-safe random number generator on each platform. All the randomization functions have a `using:` parameter that take a `RandomNumberGenerator` that users can pass in their own random number generator.

  ```
  struct MersenneTwister: RandomNumberGenerator {
    func next() -> UInt64 {
      // implementation
    }
  ```

```
}

var mt = MersenneTwister()
let diceRoll = Int.random(in: 1 ... 6, using: &mt)
```

- SE-0194

  The new CaseIterable protocol describes types which have a static "allCases" property that is used to describe all of the cases of the type. Swift will synthesize this "allCases" property for enums that have no associated values. For example:

  ```
  enum Suit: CaseIterable {
    case heart
    case club
    case diamond
    case spade
  }

  print(Suit.allCases) // prints [Suit.heart, Suit.club, Suit.diamond, Suit.spade]
  ```

- SE-0185

  Protocol conformances are now able to be synthesized in extensions in the same file as the type definition, allowing automatic synthesis of conditional conformances to `Hashable`, `Equatable` and `Codable` (both `Encodable` and `Decodable`). For instance, if there is a generic wrapper type that can only be `Equatable` when its wrapped type is also `Equatable`, the `==` method can be automatically constructed by the compiler:

  ```
  struct Generic<Param> {
    var property: Param
  }

  extension Generic: Equatable where Param: Equatable {}
  // Automatically synthesized inside the extension:
  // static func ==(lhs: Generic, rhs: Generic) -> Bool {
  //   return lhs.property == rhs.property
  // }
  ```

  Code that wants to be as precise as possible should generally not conditionally conform to `Codable` directly, but rather its two constituent protocols `Encodable` and `Decodable`, or else one can only (for instance) decode a `Generic<Param>` if `Param` is `Encodable` in addition to `Decodable`, even though `Encodable` is likely not required:

  ```
  // Unnecessarily restrictive:
  extension Generic: Codable where Param: Codable {}

  // More precise:
  ```

```swift
extension Generic: Encodable where Param: Encodable {}
extension Generic: Decodable where Param: Decodable {}
```

Finally, due to `Decodable` having an `init` requirement, it is not possible to conform to `Decodable` in an extension of a non-final class: such a class needs to have any `init`s from protocols be `required`, which means they need to be in the class definition.

- SE-0054

  `ImplicitlyUnwrappedOptional<T>` is now an unavailable typealias of `Optional<T>`. Declarations annotated with `!` have the type `Optional<T>`. If an expression involving one of these values will not compile successfully with the type `Optional<T>`, it is implicitly unwrapped, producing a value of type `T`.

  In some cases this change will cause code that previously compiled to need to be adjusted. Please see this blog post for more information.

- SE-0206

  The standard library now uses a high-quality, randomly seeded, universal hash function, represented by the new public `Hasher` struct.

  "Random seeding" varies the result of `hashValue` on each execution of a Swift program, improving the reliability of the standard library's hashed collections such as `Set` and `Dictionary`. In particular, random seeding enables better protection against (accidental or deliberate) hash-flooding attacks.

  This change fulfills a long-standing prophecy in Hashable's documentation:

  > Hash values are not guaranteed to be equal across different executions of your program. Do not save hash values to use during a future execution.

  As a consequence of random seeding, the elements in `Set` and `Dictionary` values may have a different order on each execution. This may expose some bugs in existing code that accidentally relies on repeatable ordering.

  Additionally, the `Hashable` protocol now includes an extra function requirement, `hash(into:)`. The new requirement is designed to be much easier to implement than the old `hashValue` property, and it generally provides better hashing. To implement `hash(into:)`, simply feed the exact same components of your type that you compare in `Equatable`'s `==` implementation to the supplied `Hasher`:

```swift
struct Foo: Hashable {
  var a: String?
  var b: [Int]
  var c: [String: Int]
  static func ==(lhs: Foo, rhs: Foo) -> Bool {
    return lhs.a == rhs.a && lhs.b == rhs.b && lhs.c == rhs.c
  }

  func hash(into hasher: inout Hasher) {
    hasher.combine(a)
    hasher.combine(b)
    hasher.combine(c)
  }
}
```

Automatic synthesis for `Hashable` (SE-0185) has been updated to generate `hash(into:)` implementations. For example, the `==` and `hash(into:)` implementations above are equivalent to the ones synthesized by the compiler, and can be removed without changing the meaning of the code.

Synthesis has also been extended to support deriving `hashValue` from `hash(into:)`, and vice versa. Therefore, code that only implements `hashValue` continues to work in Swift 4.2. This new compiler functionality works for all types that can implement `Hashable`, including classes.

Note that these changes don't affect Foundation's hashing interface. Classes that subclass `NSObject` should override the `hash` property, like before.

In certain controlled environments, such as while running particular tests, it may be helpful to selectively disable hash seed randomization, so that hash values and the order of elements in `Set`/`Dictionary` values remain consistent across executions. You can disable hash seed randomization by defining the environment variable `SWIFT_DETERMINISTIC_HASHING` with the value of `1`. The Swift runtime looks at this variable during process startup and, if it is defined, replaces the random seed with a constant value.

- SR-106

  The behavior of `.description` and `.debugDescription` for floating-point numbers has been changed. Previously these unconditionally printed a fixed number of decimal digits (e.g. 15 and 17 for Double, respectively). They now print exactly as many digits as are needed for the resulting string to convert back to the original source value, and no more. For more details, see the original bug report and the linked pull request.

- SE-0193

  Various function-like declarations can now be marked as `@inlinable`, making their bodies available for optimizations from other modules.

  Inlinable function bodies must only reference public declarations, unless the referenced declaration is marked as `@usableFromInline`.

Note that the presence of the attribute itself does not force inlining or any other optimization to be performed, nor does it have any effect on optimizations performed within a single module.

- The C `long double` type is now imported as `Float80` on i386 and x86_64 macOS and Linux. The tgmath functions in the Darwin and glibc modules now support `Float80` as well as `Float` and `Double`. Several tgmath functions have been made generic over `[Binary]FloatingPoint` so that they will automatically be available for any conforming type.
- SE-0143

  The standard library types `Optional`, `Array`, `ArraySlice`, `ContiguousArray`, `Dictionary`, `Range`, and `ClosedRange` now conform to the `Hashable` protocol when their element or bound types (as the case may be) conform to `Hashable`. This makes synthesized `Hashable` implementations available for types that include stored properties of these types.
- SE-0196

  Custom compile-time warnings or error messages can be emitted using the `#warning(_:)` and `#error(_:)` directives.

  ```
  #warning("this is incomplete")

  #if MY_BUILD_CONFIG && MY_OTHER_BUILD_CONFIG
    #error("MY_BUILD_CONFIG and MY_OTHER_BUILD_CONFIG cannot both be set")
  #endif
  ```
- Public classes may now have internal `required` initializers. The rule for `required` initializers is that they must be available everywhere the class can be subclassed, but previously we said that `required` initializers on public classes needed to be public themselves. (This limitation is a holdover from before the introduction of the open/public distinction in Swift 3.)
- C macros containing casts are no longer imported to Swift if the type in the cast is unavailable or deprecated, or produces some other diagnostic when referenced. (These macros were already only imported under very limited circumstances with very simple values, so this is unlikely to affect real-world code.)
- SE-0143

  Runtime query of conditional conformances is now implemented. Therefore, a dynamic cast such as `value as? P`, where the dynamic type of `value` conditionally conforms to `P`, will succeed when the conditional requirements are met.

Swift 4.1

2018-03-29 (Xcode 9.3)

- SE-0075

  Compile-time testing for the existence and importability of modules is now implemented as a build configuration test. The `canImport` test allows the development of features that require a possibly-failing import declaration across multiple platforms.

  ```
  #if canImport(UIKit)
    import UIKit
    class MyView : UIView {}
  #elseif canImport(AppKit)
    import AppKit
    class MyView : NSView {}
  #else
    class MyView : CustomView {}
  #endif
  ```

- SE-0189

  If an initializer is declared in a different module from a struct, it must use `self.init(...)` or `self = ...` before returning or accessing `self`. Failure to do so will produce a warning in Swift 4 and an error in Swift 5. This is to keep a client app from accidentally depending on a library's implementation details, and matches an existing restriction for classes, where cross-module initializers must be convenience initializers.

  This will most commonly affect code that extends a struct imported from C. However, most imported C structs are given a zeroing no-argument initializer, which can be called as `self.init()` before modifying specific properties.

  Swift library authors who wish to continue allowing initialization on a per-member basis should explicitly declare a public memberwise initializer for clients in other modules to use.

- SE-0166 / SE-0143

  The standard library now defines the conformances of `Optional`, `Array`, `Dictionary`, and `Set` to `Encodable` and `Decodable` as conditional conformances, available only when their type parameters conform to `Encodable` or `Decodable`, respectively.

- SE-0188

  Index types for most standard library collections now conform to `Hashable`. These indices can now be used in key-path subscripts and hashed collections:

```swift
let s = "Hashable"
let p = \String.[s.startIndex]
s[keyPath: p] // "H"
```

- SE-0143 The standard library types `Optional`, `Array`, `ArraySlice`, `ContiguousArray`, and `Dictionary` now conform to the `Equatable` protocol when their element types conform to `Equatable`. This allows the `==` operator to compose (e.g., one can compare two values of type `[Int : [Int?]?]` with `==`), as well as use various algorithms defined for `Equatable` element types, such as `index(of:)`.

- SE-0157 is implemented. Associated types can now declare "recursive" constraints, which require that the associated type conform to the enclosing protocol. The standard library protocols have been updated to make use of recursive constraints. For example, the `SubSequence` associated type of `Sequence` follows the enclosing protocol:

  ```swift
  protocol Sequence {
    associatedtype Element
    associatedtype SubSequence: Sequence
      where SubSequence.Element == Element,
            SubSequence.SubSequence == SubSequence
    // ...
  }

  protocol Collection: Sequence where Self.SubSequence: Collection {
    // ...
  }
  ```

  As a result, a number of new constraints have been introduced into the standard library protocols:

  – Make the `Indices` associated type have the same traversal requirements as its enclosing protocol, e.g., `Collection.Indices` conforms to `Collection`, `BidirectionalCollection.Indices` conforms to `BidirectionalCollection`, and so on
  – Make `Numeric.Magnitude` conform to `Numeric`
  – Use more efficient `SubSequence` types for lazy filter and map
  – Eliminate the `*Indexable` protocols

- SE-0161 is fully implemented. KeyPaths now support subscript, optional chaining, and optional force-unwrapping components.

- SE-0186

  It is no longer valid to use the ownership keywords `weak` and `unowned` for property declarations in protocols. These keywords are meaningless and misleading when used in a protocol as they don't have any effect.

  In Swift 3 and 4 mode the following example will produce a warning with a fix-it to remove the keyword. In Swift 5 mode and above an error will be produced.

```
class A {}

protocol P {
    weak var weakVar: A? { get set }
    unowned var unownedVar: A { get set }
}
```

- SE-0185

  Structs and enums that declare a conformance to `Equatable`/`Hashable` now get an automatically synthesized implementation of `==`/`hashValue`. For structs, all stored properties must be `Equatable`/`Hashable`. For enums, all enum cases with associated values must be `Equatable`/`Hashable`.

  ```
  public struct Point: Hashable {
    public let x: Int
    public let y: Int

    public init(x: Int, y: Int) {
      self.x = x
      self.y = y
    }
  }

  Point(3, 0) == Point(0, 3)  // false
  Point(3, 0) == Point(3, 0)  // true
  Point(3, 0).hashValue       // -2942920663782199421

  public enum Token: Hashable {
    case comma
    case identifier(String)
    case number(Int)
  }

  Token.identifier("x") == .number(5)         // false
  Token.identifier("x") == .identifier("x")   // true
  Token.number(50).hashValue                  // -2002318238093721609
  ```

  If you wish to provide your own implementation of `==`/`hashValue`, you still can; a custom implementation will replace the one synthesized by the compiler.

Swift 4.0

2017-09-19 (Xcode 9.0)

- SE-0165 and SE-0154

  The standard library's `Dictionary` and `Set` types have some new features. You can now create a new dictionary from a sequence of keys and values, and merge keys and values into an existing dictionary.

```
let asciiTable = Dictionary(uniqueKeysWithValues: zip("abcdefghijklmnopqrstuvwxyz", 97.
// ["w": 119, "n": 110, "u": 117, "v": 118, "x": 120, "q": 113, ...]

let vegetables = ["tomato", "carrot", "onion", "onion", "carrot", "onion"]
var vegetableCounts = Dictionary(zip(vegetables, repeatElement(1, count: Int.max)),
                                  uniquingKeysWith: +)
vegetableCounts.merge([("tomato", 1)], uniquingKeysWith: +)
// ["tomato": 2, "carrot": 2, "onion": 3]
```

  Filtering a set or a dictionary now results in the same type. You can also now transform just the values of a dictionary, keeping the same keys, using the `mapValues(_:)` method.

```
let vowels: Set<Character> = ["a", "e", "i", "o", "u"]
let asciiVowels = asciiTable.filter({ vowels.contains($0.key) })
asciiVowels["a"]  // 97
asciiVowels["b"]  // nil

let asciiHexTable = asciiTable.mapValues({ "0x" + String($0, radix: 16) })
// ["w": "0x77", "n": "0x6e", "u": "0x75", "v": "0x76", "x": "0x78", ...]
```

  When using a key as a dictionary subscript, you can now supply a default value to be returned if the key is not present in the dictionary.

```
for veg in ["tomato", "cauliflower"] {
    vegetableCounts[veg, default: 0] += 1
}
// ["tomato": 3, "carrot": 2, "onion": 3, "cauliflower": 1]
```

  Use the new `init(grouping:by:)` initializer to convert an array or other sequence into a dictionary, grouped by a particular trait.

```
let buttons = // an array of button instances
let buttonsByStatus = Dictionary(grouping: buttons, by: { $0.isEnabled })
// How many enabled buttons?
print("Enabled:", buttonsByStatus[true]?.count ?? 0)
```

  Additionally, dictionaries and sets now have a visible `capacity` property and a `reserveCapacity(_:)` method similar to arrays, and a dictionary's `keys` and `values` properties are represented by specialized collections.

- SE-0161 is partially implemented. Swift now natively supports key path objects for properties. Similar to KVC key path strings in Cocoa, key path objects allow a property to be referenced independently of accessing it from a value:

  ```
  struct Point {
    var x, y: Double
  }
  let x = \Point.x
  let y = \Point.y

  let p = Point(x: 3, y: 4)
  p[keyPath: x] // gives 3
  p[keyPath: y] // gives 4
  ```

- Core Foundation types implicitly conform to Hashable (and Equatable), using CFHash and CFEqual as the implementation. This change applies even to "Swift 3 mode", so if you were previously adding this conformance yourself, use `#if swift(>=3.2)` to restrict the extension to Swift 3.1 and below. (SR-2388)

- SE-0156

  Protocol composition types can now contain one or more class type terms, forming a class-constrained protocol composition.

  For example:

  ```
  protocol Paintable {
    func paint()
  }

  class Canvas {
    var origin: CGPoint
  }

  class Wall : Canvas, Paintable {
    func paint() { ... }
  }

  func render(_: Canvas & Paintable) { ... }

  render(Wall())
  ```

  Note that class-constrained protocol compositions can be written and used in both Swift 3 and Swift 4 mode.

  Generated headers for Swift APIs will map class-constrained protocol compositions to Objective-C protocol-qualified class types in both Swift 3 and

Swift 4 mode (for instance, `NSSomeClass & SomeProto & OtherProto` in Swift becomes `NSSomeClass <SomeProto, OtherProto>` in Objective-C).

Objective-C APIs which use protocol-qualified class types differ in behavior when imported by a module compiled in Swift 3 mode and Swift 4 mode. In Swift 3 mode, these APIs will continue to import as protocol compositions without a class constraint (eg, `SomeProto & OtherProto`).

In Swift 4 mode, protocol-qualified class types import as class-constrained protocol compositions, for a more faithful mapping of APIs from Objective-C to Swift.

Note that the current implementation of class-constrained protocol compositions lacks three features outlined in the Swift evolution proposal:

  - In the evolution proposal, a class-constrained is permitted to contain two different classes as long as one is a superclass of the other. The current implementation only allows multiple classes to appear in the composition if they are identical.
  - In the evolution proposal, associated type and class inheritance clauses are generalized to allow class-constrained protocol compositions. The current implementation does not allow this.
  - In the evolution proposal, protocol inheritance clauses are allowed to contain a class, placing a requirement that all conforming types are a subclass of the given class. The current implementation does not allow this.

These missing aspects of the proposal can be introduced in a future release without breaking source compatibility with existing code.

- SE-0142

  Protocols and associated types can now contain `where` clauses that provide additional restrictions on associated types. For example:

```
protocol StringRepresentable: RawRepresentable
where RawValue == String { }

protocol RawStringWrapper {
  associatedtype Wrapped: RawRepresentable
    where Wrapper.RawValue == String
}
```

- SE-0160

  In Swift 4 mode, a declaration is inferred to be `@objc` where it is required for semantic consistency of the programming model. Specifically, it is inferred when:

  - The declaration is an override of an `@objc` declaration

  - The declaration satisfies a requirement in an `@objc` protocol
  - The declaration has one of the following attributes: `@IBAction`, `@IBOutlet`, `@IBInspectable`, `@GKInspectable`, or `@NSManaged`

  Additionally, in Swift 4 mode, `dynamic` declarations that don't have `@objc` inferred based on the rules above will need to be explicitly marked `@objc`.

  Swift 3 compatibility mode retains the more-permissive Swift 3 rules for inference of `@objc` within subclasses of `NSObject`. However, the compiler will emit warnings about places where the Objective-C entry points for these inference cases are used, e.g., in a `#selector` or `#keyPath` expression, via messaging through `AnyObject`, or direct uses in Objective-C code within a mixed project. The warnings can be silenced by adding an explicit `@objc`. Uses of these entrypoints that are not statically visible to the compiler can be diagnosed at runtime by setting the environment variable `SWIFT_DEBUG_IMPLICIT_OBJC_ENTRYPOINT` to a value between 1 and 3 and testing the application. See the migration discussion in SE-0160.

- SE-0138:

  Slicing a raw buffer no longer results in the same raw buffer type. Specifically, `Unsafe[Mutable]BufferPointer.SubSequence` now has type `[Mutable]RandomAccessSlice<Unsafe[Mutable]RawBufferPointer>`. Therefore, indexing into a raw buffer slice is no longer zero-based. This is required for raw buffers to fully conform to generic `Collection`. Changing the slice type resulted in the following behavioral changes:

  Passing a region within buffer to another function that takes a buffer can no longer be done via subscript:

  Incorrect: `takesRawBuffer(buffer[i..<j])`

  This now requires explicit initialization, using a `rebasing:` initializer, which converts from a slice to a zero-based `Unsafe[Mutable]RawBufferPointer`:

  Correct: `takesRawBuffer(UnsafeRawBufferPointer(rebasing: buffer[i..<j]))`

  Subscript assignment directly from a buffer no longer compiles:

  Incorrect: `buffer[n..<m] = smaller_buffer`

  This now requires creation of a slice from the complete source buffer:

  Correct: `buffer[n..<m] = smaller_buffer.suffix(from: 0)`

  `UnsafeRawBufferPointer`'s slice type no longer has a nonmutating subscript setter. So assigning into a mutable `let` buffer no longer compiles:

  ```
  let slice = buffer[n..<m]
  slice[i..<j] = buffer[k..<l]
  ```

  The assigned buffer slice now needs to be a `var`.

```
var slice = buffer[n..<m]
slice[i..<j] = buffer[k..<l]
```

- SR-1529:

  Covariant method overrides are now fully supported, fixing many crashes and compile-time assertions when defining or calling such methods. Examples:

  ```
  class Bed {}
  class Nook : Bed {}

  class Cat<T> {
    func eat(snack: T) {}
    func play(game: String) {}
    func sleep(where: Nook) {}
  }

  class Dog : Cat<(Int, Int)> {
    // 'T' becomes concrete
    override func eat(snack: (Int, Int)) {}

    // 'game' becomes optional
    override func play(game: String?) {}

    // 'where' becomes a superclass
    override func sleep(where: Bed) {}
  }
  ```

- SE-0148:

  Subscript declarations can now be defined to have generic parameter lists. Example:

  ```
  extension JSON {
    subscript<T>(key: String) -> T?
        where T : JSONConvertible {
      // ...
    }
  }
  ```

- SE-0110:

  In Swift 4 mode, Swift's type system properly distinguishes between functions that take one tuple argument, and functions that take multiple arguments.

- More types of C macros which define integer constants are supported by the importer. Specifically the `+`, `-`, `*`, `/`, `^`, `>>`, `==`, `<`, `<=`, `>`, `>=` operators are now recognized, and the previously-supported `<<`, `&&`, `||`,

`&`, `|` operators always look through importable macros on each side of the operator. Logical AND and OR macros (`&&` and `||`) are now imported as Boolean constants, rather than integers of value 0 or 1.

```
#define HIGHER    (5 + 5)
#define THE_EDGE  (INT64_MAX - 1)
#define FORTY_TWO (6 * 9)
#define SPLIT     (THE_EDGE / FORTY_TWO)

#define HALF_AND_HALF (UINT64_MAX ^ UINT32_MAX)

#define SMALL   (BITWIDTH == 32)
#define TINY    (BITWIDTH <= 16)
#define LIMITED (SMALL || TINY)   // now imported as Bool.
```

Swift 3.1

2017-03-27 (Xcode 8.3)

- SE-0080:

  Adds a new family of conversion initializers to all numeric types that either complete successfully without loss of information or return nil.

- Swift will now warn when an `NSObject` subclass attempts to override the class `initialize` method. Swift doesn't guarantee that references to class names trigger Objective-C class realization if they have no other side effects, leading to bugs when Swift code attempted to override `initialize`.

- SR-2394

  C functions that "return twice" are no longer imported into Swift. Instead, they are explicitly made unavailable, so attempting to reference them will result in a compilation error.

  Examples of functions that "return twice" include `vfork` and `setjmp`. These functions change the control flow of a program in ways that that Swift has never supported. For example, definitive initialization of variables, a core Swift language feature, could not be guaranteed when these functions were used.

  Swift code that references these functions will no longer compile. Although this could be considered a source-breaking change, it's important to note that any use of these functions would have most likely crashed at runtime. Now, the compiler will prevent them from being used in the first place.

- Indirect fields from C structures and unions are now always imported, while they previously weren't imported if they belonged to a union. This is done by naming anonymous fields. For example:

```
typedef struct foo_t {
  union {
    int a;
    double b;
  };
} foo_t;
```

Get imported as:

```
struct foo_t {
  struct __Unnamed_union___Anonymous_field0 {
    var a : Int { get set }
    var b : Double { get set }
  }
  var __Anonymous_field0 : foo_t.__Unnamed_union___Anonymous_field0

  // a and b are computed properties accessing the content of __Anonymous_field0
  var a : Int { get set }
  var b : Double { get set }
}
```

Since new symbols are exposed from imported structure/unions, this may conflict with existing code that extended C types in order to provide their own accessors to the indirect fields.

- The `withoutActuallyEscaping` function from SE-0103 has been implemented. To pass off a non-escaping closure to an API that formally takes an `@escaping` closure, but which is used in a way that will not in fact escape it in practice, use `withoutActuallyEscaping` to get an escapable copy of the closure and delimit its expected lifetime. For example:

```
func doSimultaneously(_ f: () -> (), and g: () -> (), on q: DispatchQueue) {
  // DispatchQueue.async normally has to be able to escape its closure
  // since it may be called at any point after the operation is queued.
  // By using a barrier, we ensure it does not in practice escape.
  withoutActuallyEscaping(f) { escapableF in
    withoutActuallyEscaping(g) { escapableG in
      q.async(escapableF)
      q.async(escapableG)
      q.sync(flags: .barrier) {}
    }
  }
  // `escapableF` and `escapableG` must be dequeued by the point
  // `withoutActuallyEscaping` returns.
}
```

The old workaround of using `unsafeBitCast` to cast to an `@escaping` type is not guaranteed to work in future versions of Swift, and will now raise a warning.

- SR-1446

  Nested types may now appear inside generic types, and nested types may have their own generic parameters:

  ```
  struct OuterNonGeneric {
      struct InnerGeneric<T> {}
  }

  struct OuterGeneric<T> {
      struct InnerNonGeneric {}

      struct InnerGeneric<T> {}
  }

  extension OuterNonGeneric.InnerGeneric {}
  extension OuterGeneric.InnerNonGeneric {}
  extension OuterGeneric.InnerGeneric {}
  ```

- SR-1009:

  Constrained extensions allow same-type constraints between generic parameters and concrete types. This enables you to create extensions, for example, on `Array` with `Int` elements:

  ```
  extension Array where Element == Int { }
  ```

- SE-0045:

  The `Sequence` protocol adds two new members `prefix(while:)` and `drop(while:)` for common utility. `prefix(while:)` requests the longest subsequence satisfying a predicate. `drop(while:)` requests the remaining subsequence after dropping the longest subsequence satisfying a predicate.

Swift 3.0

2016-09-13 (Xcode 8.0)

- SE-0101:

The functions `sizeof()`, `strideof()`, and `alignof()` have been removed. Memory layout properties for a type `T` are now spelled `MemoryLayout<T>.size`, `MemoryLayout<T>.stride`, and `MemoryLayout<T>.alignment`, respectively.

- SE-0136:

  The functions `sizeofValue()`, `strideofValue()`, and `alignofValue()` have been renamed to `MemoryLayout.size(ofValue:)`, `MemoryLayout.stride(ofValue:)`, and `MemoryLayout.alignment(ofValue:)`.

- SE-0125:

The functions `isUniquelyReferenced()` and `isUniquelyReferencedNonObjC()` have been removed. Call the function `isKnownUniquelyReferenced()` instead.

Classes using `isUniquelyReferenced()` needed to inherit from `NonObjectiveCBase`. The `NonObjectiveCBase` class has been removed.

The method `ManagedBufferPointer.holdsUniqueReference` has been renamed to `ManagedBufferPointer.isUniqueReference`.

```
// old
class SwiftKlazz : NonObjectiveCBase {}
expectTrue(isUniquelyReferenced(SwiftKlazz()))

var managedPtr : ManagedBufferPointer = ...
if !managedPtr.holdsUniqueReference() {
  print("not unique")
}

// new
class SwiftKlazz {}
expectTrue(isKnownUniquelyReferenced(SwiftKlazz()))

var managedPtr : ManagedBufferPointer = ...
if !managedPtr.isUniqueReference() {
  print("not unique")
}
```

- SE-0124:

Initializers on `Int` and `UInt` that accept an `ObjectIdentifier` must now use an explicit `bitPattern` label.

```
let x: ObjectIdentifier = ...

// old
let u = UInt(x)
let i = Int(x)

// new
let u = UInt(bitPattern: x)
let i = Int(bitPattern: x)
```

- SE-0120:

  The collection methods `partition()` and `partition(isOrderedBefore:)` have been removed from Swift. They are replaced by the method `partition(by:)` which takes a unary predicate.

  Calls to the `partition()` method can be replaced by the following code.

```
// old
let p = c.partition()

// new
let p = c.first.flatMap({ first in
    c.partition(by: { $0 >= first })
}) ?? c.startIndex
```

- SE-0103:

  Closure parameters are now non-escaping by default and do not require `@noescape` annotation. Use `@escaping` to indicate that a closure parameter can escape. `@autoclosure(escaping)` is now spelled `@autoclosure @escaping`. `@noescape` and `@autoclosure(escaping)` are deprecated.

- SE-0115:

  To clarify their roles, `*LiteralConvertible` protocols have been renamed to `ExpressibleBy*Literal`. The protocol requirements are unchanged.

- SE-0107:

  An `Unsafe[Mutable]RawPointer` type has been introduced. It replaces `Unsafe[Mutable]Pointer<Void>`. Conversion from `UnsafePointer<T>` to `UnsafePointer<U>` has been disallowed. `Unsafe[Mutable]RawPointer` provides an API for untyped memory access, and an API for binding memory to a type. Binding memory allows for safe conversion between pointer types.

  For detailed instructions on how to migrate your code to the new API refer to the UnsafeRawPointer migration guide. See also: See `bindMemory(to:capacity:)`, `assumingMemoryBound(to:)`, and `withMemoryRebound(to:capacity:)`.

- SE-0096:

  The `dynamicType` keyword has been removed from Swift. It's replaced by a new primitive function `type(of:)`. Existing code using the `.dynamicType` member to retrieve the type of an expression should migrate to this new primitive. Code using `.dynamicType` in conjunction with `sizeof` should migrate to the `MemoryLayout` structure introduced by SE-0101.

- SE-0113:

  The following two methods were added to `FloatingPoint`:

  ```
  func rounded(_ rule: FloatingPointRoundingRule) -> Self
  mutating func round( _ rule: FloatingPointRoundingRule)
  ```

  These methods bind the IEEE 754 roundToIntegral operations. They provide the functionality of the C / C++ `round()`, `ceil()`, `floor()`, and `trunc()` functions along with other rounding operations.

Following onto SE-0113 and SE-0067, the following `Darwin.C` and `glibc` module mathematical operations now operate on any type conforming to `FloatingPoint`: `fabs`, `sqrt`, `fma`, `remainder`, `fmod`, `ceil`, `floor`, `round`, and `trunc`.

See also: the changes associated with SE-0067.

- SE-0067:

  The `FloatingPoint` protocol has been expanded to include most IEEE 754 required operations. A number of useful properties have been added to the protocol, representing quantities like the largest finite value or the smallest positive normal value (these correspond to the macros such as FLT_MAX defined in C).

  While almost all of the changes are additive, four changes impact existing code:

  - The `%` operator is no longer available for `FloatingPoint` types. It was difficult to use correctly and its semantics did not match those of the corresponding integer operation. This made it something of an attractive nuisance. The new method `formTruncatingRemainder(dividingBy:)` provides the old semantics if they are needed.
  - The static property `.NaN` has been renamed `.nan`.
  - The static property `.quietNaN` was redundant and has been removed. Use `.nan` instead.
  - The predicate `isSignaling` has been renamed `isSignalingNaN`.

  See also: the changes associated with SE-0113.

- SE-0111:

  Argument labels have been removed from Swift function types. They are now part of the name of a function, subscript, or initializer. Calls to a function or initializer, and subscript uses, still require argument labels as they always have:

```
func doSomething(x: Int, y: Int) { }
doSomething(x: 0, y: 0)     // argument labels are required
```

  Unapplied references to functions or initializers no longer carry argument labels. For example:

```
let f = doSomething(x:y:)     // inferred type is now (Int, Int) -> Void
```

  Explicitly-written function types can no longer carry argument labels. You can still provide parameter names for documentation purposes using the '_' in the argument label position:

```
typealias CompletionHandler =
   (token: Token, error: Error?) -> Void   // error: function types cannot have argumen

typealias CompletionHandler =
   (_ token: Token, _ error: Error?) -> Void   // okay: names are for documentation pur
```

- SE-0025:

  The access level formerly known as `private` is now called `fileprivate`. A Swift 3 declaration marked `private` can no longer be accessed outside its lexical scope (essentially its enclosing curly braces `{}`). A `private` declaration at the top level of a file can be accessed anywhere within the same file, as it could in Swift 2.

- SE-0131:

  The standard library introduces the `AnyHashable` type for use in hashed heterogeneous collections. Untyped `NSDictionary` and `NSSet` Objective-C APIs now import as `[AnyHashable: Any]` and `Set<AnyHashable>`.

- SE-0102:

  Swift removes the `@noreturn` attribute on function declarations and replaces the attribute with an empty `Never` type:

  ```
  @noreturn func fatalError(msg: String) { ... }  // old
  func fatalError(msg: String) -> Never { ... }   // new

  func performOperation<T>(continuation: @noreturn T -> ()) { ... }  // old
  func performOperation<T>(continuation: T -> Never) { ... }         // new
  ```

- SE-0116:

  Swift now imports Objective-C `id` APIs as `Any`. In Swift 2, `id` imported as `AnyObject`. Swift also imports untyped `NSArray` and `NSDictionary` as `[Any]` and `[AnyHashable: Any]`, respectively.

- SE-0072:

  Swift eliminates implicit bridging conversions. Use `as` to force the conversion from a Swift value type to its corresponding object. For example, use `string as NSString`. Use `as AnyObject` to convert a Swift value to its boxed `id` representation.

- Collection subtype conversions and dynamic casts now work with protocol types:

  ```
  protocol P {}; extension Int: P {}
  var x: [Int] = [1, 2, 3]
  var p: [P] = x
  var x2 = p as! [Int]
  ```

- SR-2131:

  The `hasPrefix` and `hasSuffix` functions now consider the empty string to be a prefix and suffix of all strings.

- SE-0128:

  Some non-failable UnicodeScalar initializers now return an Optional. When a UnicodeScalar cannot be constructed, these initializers return nil.

  ```
  // Old
  var string = ""
  let codepoint: UInt32 = 55357 // Invalid
  let ucode = UnicodeScalar(codepoint) // Program crashes here.
  string.append(ucode)
  ```

  The updated initializers allow users to write code that safely works around invalid codepoints, like this example:

  ```
  // New
  var string = ""
  let codepoint: UInt32 = 55357 // Invalid
  if let ucode = UnicodeScalar(codepoint) {
      string.append(ucode)
  } else {
      // do something else
  }
  ```

- SE-0095:

  Swift removes the `protocol<...>` composition construct and introduces an infix type operator `&` in its place.

  ```
  let a: Foo & Bar
  let b = value as? A & B & C
  func foo<T : Foo & Bar>(x: T) { ... }
  func bar(x: Foo & Bar) { ... }
  typealias G = GenericStruct<Foo & Bar>
  ```

  Swift previously defined the empty protocol composition (the `Any` type) as `protocol<>`. This definition has been removed from the standard library. The `Any` keyword behavior remains unchanged.

- SE-0091:

  Swift permits you to define operators within types or their extensions. For example:

  ```
  struct Foo: Equatable {
    let value: Int

    static func ==(lhs: Foo, rhs: Foo) -> Bool {
  ```

```
    return lhs.value == rhs.value
  }
}
```

You must declare these operators as `static` (or, within a class, `class final`) and they must use the same signature as their global counterparts. As part of this change, protocol-declared operator requirements must be declared `static` explicitly:

```
protocol Equatable {
  static func ==(lhs: Self, rhs: Self) -> Bool
}
```

Note: The type checker performance optimization described by SE-0091 is not yet implemented.

- SE-0099:

  Condition clauses in `if`, `guard`, and `while` statements now use a more regular syntax. Each pattern or optional binding must be prefixed with `case` or `let` respectively, and all conditions are separated by `,` instead of `where`.

  ```
  // before
  if let a = a, b = b where a == b { }

  // after
  if let a = a, let b = b, a == b { }
  ```

- SE-0112:

  The `NSError` type now bridges to the Swift `Error` protocol type (formerly `ErrorProtocol` in Swift 3, `ErrorType` in Swift 2) in Objective-C APIs. `NSError` now bridges like other Objective-C types, e.g., `NSString` bridges to `String`.

  For example, the `UIApplicationDelegate` method `applicate(_:didFailToRegisterForRemoteNotific`
  previously accepted an `NSError` argument:

  ```
  optional func application(_ application: UIApplication,
    didFailToRegisterForRemoteNotificationsWithError error: NSError)
  ```

Now it accepts an `Error` argument:

```
optional func application(_ application: UIApplication,
  didFailToRegisterForRemoteNotificationsWithError error: Error)
```

Error types imported from Cocoa[Touch] maintain all of the information in the corresponding `NSError`. You no longer `catch let as NSError` to extract, for example, the user-info dictionary.

Specific error types now contain typed accessors for their common user-info keys. For example:

```
catch let error as CocoaError where error.code == .fileReadNoSuchFileError {
  print("No such file: \(error.url)")
}
```

Swift-defined error types can now provide localized error descriptions by adopting the new `LocalizedError` protocol, e.g.,

```
extension HomeworkError : LocalizedError {
  var errorDescription: String? {
    switch self {
    case .forgotten: return NSLocalizedString("I forgot it")
    case .lost: return NSLocalizedString("I lost it")
    case .dogAteIt: return NSLocalizedString("The dog ate it")
    }
  }
}
```

New `RecoverableError` and `CustomNSError` protocols allow additional control over the handling of the error.

- SE-0060:

  Function parameters with defaulted arguments are specified in declaration order. Call sites must now supply those arguments using that order:

  ```
  func requiredArguments(a: Int, b: Int, c: Int) {}
  func defaultArguments(a: Int = 0, b: Int = 0, c: Int = 0) {}

  requiredArguments(a: 0, b: 1, c: 2)
  requiredArguments(b: 0, a: 1, c: 2) // error
  defaultArguments(a: 0, b: 1, c: 2)
  defaultArguments(b: 0, a: 1, c: 2) // error
  ```

  Labeled parameters with default arguments may still be elided, so long as included arguments follow declaration order:

  ```
  defaultArguments(a: 0) // ok
  defaultArguments(b: 1) // ok
  defaultArguments(c: 2) // ok
  defaultArguments(a: 1, c: 2) // ok
  defaultArguments(b: 1, c: 2) // ok
  defaultArguments(c: 1, b: 2) // error
  ```

- Traps from force-unwrapping nil `Optional`s now show the source location of the force unwrap operator.

- SE-0093:

  Slice types add a `base` property that allows public readonly access to their base collections.

- Nested generic functions may now capture bindings from the environment, for example:

  ```
  func outer<T>(t: T) -> T {
    func inner<U>(u: U) -> (T, U) {
      return (t, u)
    }
    return inner(u: (t, t)).0
  }
  ```

- Initializers are now inherited even if the base class or derived class is generic:

  ```
  class Base<T> {
    let t: T

    init(t: T) {
      self.t = t
    }
  }

  class Derived<T> : Base<T> {
    // init(t: T) is now synthesized to call super.init(t: t)
  }
  ```

- SE-0081:

  "Move `where` clause to end of declaration" is now implemented. This change allows you to write `where` clauses after a declaration signature and before its body. For example, before this change was implemented, you'd write:

  ```
  func anyCommonElements<T : SequenceType, U : SequenceType
      where T.Generator.Element: Equatable, T.Generator.Element == U.Generator.Element>
      (lhs: T, _ rhs: U) -> Bool
  {
      ...
  }
  ```

  Now the `where` clause appears just before the body:

  ```
  func anyCommonElements<T : SequenceType, U : SequenceType>(lhs: T, _ rhs: U) -> Bool
      where T.Generator.Element: Equatable, T.Generator.Element == U.Generator.Element
  {
      ...
  }
  ```

  The old form is currently accepted for compatibility. It will eventually be rejected.

- SE-0071:

"Allow (most) keywords in member references" is implemented. This change allows the use of members after a dot without backticks, e.g. "foo.default", even though `default` is a keyword for `switch` statements.

- SE-0057:

  Objective-C lightweight generic classes are now imported as generic types in Swift. Some limitations apply because Objective-C generics are not represented at runtime:

  - When an ObjC generic class is used in a checked `as?`, `as!`, or `is` cast, the generic parameters are not checked at runtime. The cast succeeds if the operand is an instance of the ObjC class, regardless of parameters.

    ```
    let x = NSFoo<NSNumber>(value: NSNumber(integer: 0))
    let y: AnyObject = x
    let z = y as! NSFoo<NSString> // Succeeds
    ```

  - Swift subclasses can only inherit from an ObjC generic class when its generic parameters are fully specified.

    ```
    // Error: Can't inherit ObjC generic class with unbound parameter T
    class SwiftFoo1<T>: NSFoo<T> { }

    // OK: Can inherit ObjC generic class with specific parameters
    class SwiftFoo2<T>: NSFoo<NSString> { }
    ```

  - Swift can extend ObjC generic classes but the extensions cannot be constrained, and definitions inside the extension don't have access to the class's generic parameters.

    ```
    extension NSFoo {
      // Error: Can't access generic param T
      func foo() -> T {
        return T()
      }
    }

    // Error: extension can't be constrained
    extension NSFoo where T: NSString {
    }
    ```

  - Foundation container classes `NS[Mutable]Array`, `NS[Mutable]Set`, and `NS[Mutable]Dictionary` are still imported as nongeneric classes for the time being.

- SE-0036:

  Enum elements can no longer be accessed as instance members in instance methods.

  - As part of the changes for SE-0055 (see below), the *pointee* types of imported pointers (e.g. the `id` in `id *`) are no longer assumed to always be `_Nullable` even if annotated otherwise.
  - An implicit or explicit annotation of `_Null_unspecified` on a pointee type still imports as `Optional`.

- SE-0055:

  The types `UnsafePointer`, `UnsafeMutablePointer`, `AutoreleasingUnsafeMutablePointer`, `OpaquePointer`, `Selector`, and `Zone` (formerly `NSZone`) now represent non-nullable pointers, i.e. pointers that are never `nil`. A nullable pointer is now represented using `Optional`, e.g. `UnsafePointer<Int>?` For types imported from C, non-object pointers (such as `int *`) now have their nullability taken into account.

  One possible area of difficulty is passing a nullable pointer to a function that uses C variadics. Swift will not permit this directly. As a workaround, use the following idiom to pass a pointer-sized integer value instead:

  ```
  unsafeBitCast(nullablePointer, to: Int.self)
  ```

- SE-0046:

  Function parameters adopt consistent labeling across all function parameters. With this update, first parameter declarations match the existing behavior of the second and later parameters. This change makes the language simpler.

  Functions that were written and called as follows:

  ```
  func foo(x: Int, y: Int) {}
  foo(1, y: 2)

  func bar(a a: Int, b: Int) {}
  bar(a: 3, b: 4)
  ```

  Are now written as follows with the same behavior at call sites:

  ```
  func foo(_ x: Int, y: Int) {}
  foo(1, y: 2)

  func bar(a: Int, b: Int) {}
  bar(a: 3, b: 4)
  ```

- SE-0037:

  Comments are now treated as whitespace when determining whether an operator is prefix, postfix, or binary. For example, these now work:

  ```
  if /*comment*/!foo { ... }
  1 +/*comment*/2
  ```

Comments can no longer appear between a unary operator and its argument.

```
foo/* comment */! // no longer works
```

Parse errors resulting from this change can be resolved by moving the comment outside the expression.

- SE-0031:

  The location of the inout attribute moves to after the colon (`:`) and before the parameter type.

  ```
  func foo(inout x: Int) {
  }
  ```

  will now be written as:

  ```
  func foo(x: inout Int) {
  }
  ```

- SE-0053:

  `let` is no longer accepted as a parameter attribute for functions. The compiler provides a fixit to remove it from the function declaration.

- SE-0003:

  `var` is no longer accepted as a parameter attribute for functions. The compiler provides a fixit to create a shadow copy in the function body.

  ```
  func foo(var x: Int) {
  }
  ```

  will now be written as:

  ```
  func foo(x: Int) {
    var x = x
  }
  ```

- The “none” members of imported NS_OPTIONS option sets are marked as unavailable when they are imported. Use `[]` to make an empty option set, instead of a None member.

- SE-0043

  Adds the ability to declare variables in multiple patterns in cases.

- SE-0005

  Allows the Clang importer to import ObjC symbols using substantially different Swift-like naming paradigms:

  – These updates generalize the use of `swift_name`, allowing arbitrary C and Objective-C entity import names. This adds fine-grained control over the import process.

  - Redundant type names are pruned (`documentForURL(_: NSURL)` becomes `document(for: URL)`). Selectors are guaranteed to never be empty, to be transformed into Swift keywords, to be vacuously named (like `get`, `set`, `with`, `for`). Additional pruning rules preserve readability and sense.
  - Common arguments are sensibly defaulted where the Objective-C API strongly hints at the need for a default argument. (For example, nullable trailing closures default to `nil`, option sets to `[]`, and `NSDictionary` parameters to `[:]`.) First argument labels are added for defaulted arguments.
  - Boolean properties are prepended with `is`, and read as assertions on the receiver.
  - Non-type values, including enumerators, are lowercased.
  - Classes that implement `compare(_:) -> NSComparisonResult` automatically import as `Comparable`.

- SE-0040

  Attributes change from using `=` in parameters lists to using `:`, aligning with function call syntax.

  ```
  // before
  @available(*, unavailable, renamed="MyRenamedProtocol")

  // after
  @available(*, unavailable, renamed: "MyRenamedProtocol")
  ```

- SE-0048

  Generic typealiases are now supported. For example:

  ```
  typealias StringDictionary<T> = Dictionary<String, T>
  typealias IntFunction<T> = (T) -> Int
  typealias MatchingTriple<T> = (T, T, T)
  typealias BackwardTriple<T1, T2, T3> = (T3, T2, T1)
  ```

  etc.

- SE-0049

  The `@noescape` attribute is extended to be a more general type attribute. You can now declare values of `@noescape` function type, e.g. in manually curried function signatures. You can now also declare local variables of `@noescape` type, and use `@noescape` in `typealiases`. For example, this is now valid code:

  ```
  func apply<T, U>(@noescape f: T -> U,
                   @noescape g: (@noescape T -> U) -> U) -> U {
    return g(f)
  }
  ```

- SE-0034

  The `#line` directive (which resets the logical source location for diagnostics and debug information) is renamed to `#sourceLocation`.

- SE-0002

  Curried function syntax (with successive parenthesized groups of arguments) is removed, and now produces a compile-time error. Use chained functional return types instead.

```
// Before
public func project(function f: FunctionType)(p0: CGPoint, p1: CGPoint)(x: CGFloat) -> CGPoi

// After
public func project(function f: FunctionType) -> (p0: CGPoint, p1: CGPoint) -> (x: CGFloat)
```

- Generic signatures can now contain superclass requirements with generic parameter types, for example:

  ```
  func f<Food : Chunks<Meat>, Meat : Molerat>(f: Food, m: Meat) {}
  ```

- Section markers are created in ELF binaries through special objects during link time. These objects allow for the deletion of `swift.ld` and the use of non-BFD linkers. A new argument to swiftc is provided to select the linker used, and the gold linker is set as the default for arm-based platforms.

- Catch blocks in `rethrows` functions may now `throw` errors. For example:

  ```
  func process(f: () throws -> Int) rethrows -> Int {
      do {
          return try f()
      } catch is SomeError {
          throw OtherError()
      }
  }
  ```

- Throwing closure arguments of a rethrowing function may now be optional. For example:

  ```
  func executeClosureIfNotNil(closure: (() throws -> Void)?) rethrows {
      try closure?()
  }
  ```

- SE-0064:

  The Objective-C selectors for the getter or setter of a property can now be referenced with `#selector`. For example:

  ```
  let sel1 = #selector(getter: UIView.backgroundColor) // sel1 has type Selector
  let sel2 = #selector(setter: UIView.backgroundColor) // sel2 has type Selector
  ```

- SE-0062:

  A key-path can now be formed with `#keyPath`. For example:

  ```
  person.valueForKeyPath(#keyPath(Person.bestFriend.lastName))
  ```

Swift 2.2

2016-03-21 (Xcode 7.3)

- SE-0011:

  Associated types in protocols can now be specified with a new `associatedtype` declaration, to replace the use of `typealias`:

  ```
  protocol P {
    associatedtype Ty
  }
  ```

  The `typealias` keyword is still allowed (but deprecated and produces a warning) in Swift 2.2. This warning will become an error in Swift 3.0.

- SE-0002:

  Curried function syntax has been deprecated, and is slated to be removed in Swift 3.0.

- SE-0004:

  The `++` and `--` operators have been deprecated, and are slated to be removed in Swift 3.0. As a replacement, please use `x += 1` on integer or floating point types, and `x = x.successor()` on Index types.

- SE-0029:

  The implicit tuple splat behavior in function application has been deprecated and will be removed in Swift 3.0. For example, this code:

  ```
  func foo(a : Int, b : Int) { ... }
  let x = (1, b: 2)
  foo(x)   // Warning, deprecated.
  ```

  should move to being written as:

  ```
  foo(x.0, x.b)
  ```

- SE-0028:

  New `#file`, `#line`, `#column`, and `#function` expressions have been introduced to replace the existing `__FILE__`, `__LINE__`, `__COLUMN__`, and `__FUNCTION__` symbols. The `__FILE__`-style symbols have been deprecated, and will be removed in Swift 3.0.

- The operator identifier lexer grammar has been revised to simplify the rules for operators that start with a dot (“.”). The new rule is that an operator that starts with a dot may contain other dots in it, but operators that start with some other character may not contain dots. For example:

```
x....foo   --> "x" "...." "foo"
x&%^.foo   --> "x" "&%^"  ".foo"
```

This eliminates a special case for the `..<` operator, folding it into a simple and consistent rule.

- SE-0007:

  The "C-style for loop", which is spelled `for init; comparison; increment {}` has been deprecated and is slated for removal in Swift 3.0.

- Three new doc comment fields, namely `- keyword:`, `- recommended:` and `- recommendedover:`, allow Swift users to cooperate with code completion engine to deliver more effective code completion results. The `- keyword:` field specifies concepts that are not fully manifested in declaration names. `- recommended:` indicates other declarations are preferred to the one decorated; to the contrary, `- recommendedover:` indicates the decorated declaration is preferred to those declarations whose names are specified.

- Designated class initializers declared as failable or throwing may now return `nil` or throw an error, respectively, before the object has been fully initialized. For example:

```
class Widget : Gadget {
  let complexity: Int

  init(complexity: Int, elegance: Int) throws {
    if complexity > 3 { throw WidgetError.TooComplex }
    self.complexity = complexity

    try super.init(elegance: elegance)
  }
}
```

- All slice types now have `removeFirst()` and `removeLast()` methods.

- `ArraySlice.removeFirst()` now preserves element indices.

- Global `anyGenerator()` functions have been changed into initializers on `AnyGenerator`, making the API more intuitive and idiomatic. They have been deprecated in Swift 2.2, and will be removed in Swift 3.0.

- Closures appearing inside generic types and generic methods can now be converted to C function pointers, as long as no generic type parameters are referenced in the closure's argument list or body. A conversion of a closure that references generic type parameters now produces a diagnostic instead of crashing.

  (rdar://problem/22204968)

- Anonymously-typed members of C structs and unions can now be accessed from Swift. For example, given the following struct 'Pie', the 'crust' and 'filling' members are now imported:

  ```
  struct Pie {
    struct { bool crispy; } crust;
    union { int fruit; } filling;
  }
  ```

  Since Swift does not support anonymous structs, these fields are imported as properties named `crust` and `filling` having nested types named `Pie.__Unnamed_crust` and `Pie.__Unnamed_filling`.

  (rdar://problem/21683348)

- SE-0001:

  Argument labels and parameter names can now be any keyword except `var`, `let`, or `inout`. For example:

  ```
  NSURLProtectionSpace(host: "somedomain.com", port: 443, protocol: "https",
                       realm: "Some Domain", authenticationMethod: "Basic")
  ```

  would previously have required `protocol` to be surrounded in back-ticks.

- SE-0015:

  Tuples (up to arity 6) whose elements are all `Comparable` or `Equatable` now implement the full set of comparison/equality operators. The comparison operators are defined in terms of lexicographical order.

- The `@objc(SomeName)` attribute is now supported on enums and enum cases to rename the generated Objective-C declaration.

  (rdar://problem/21930334)

- SE-0021:

  When referencing a function or initializer, one can provide the complete name, including argument labels. For example:

  ```
  let fn1 = someView.insertSubview(_:at:)
  let fn2 = someView.insertSubview(_:aboveSubview:)

  let buttonFactory = UIButton.init(type:)
  ```

- SE-0020:

  There is a new build configuration function, `#if swift(>=x.y)`, which tests if the current Swift language version is at least `x.y`. This allows you to conditionally compile code for multiple language versions in the same file, even with different syntax, by deactivating parsing in inactive code blocks. For example:

```
#if swift(>=2.2)
  // Only this code will be parsed in Swift 3.0
  func foo(x: Int) -> (y: Int) -> () {}
#else
  // This code is ignored entirely.
  func foo(x: Int)(y: Int) {}
#endif
```

- SE-0022:

  The Objective-C selector of a Swift method can now be determined directly with the #selector expression, e.g.,:

  ```
  let sel = #selector(insertSubview(_:aboveSubview:)) // sel has type Selector
  ```

  Along with this change, the use of string literals as selectors has been deprecated, e.g.,

  ```
  let sel: Selector = "insertSubview:aboveSubview:"
  ```

  Generally, such string literals should be replaced with uses of `#selector`, and the compiler will provide Fix-Its that use `#selector`. In cases where this is not possible (e.g., when referring to the getter of a property), one can still directly construct selectors, e.g.:

  ```
  let sel = Selector("propertyName")
  ```

  Note that the compiler is now checking the string literals used to construct Selectors to ensure that they are well-formed Objective-C selectors and that there is an `@objc` method with that selector.

Swift 2.1

2015-10-21 (Xcode 7.1)

- Enums imported from C now automatically conform to the `Equatable` protocol, including a default implementation of the `==` operator. This conformance allows you to use C enum pattern matching in switch statements with no additional code. **(17287720)**
- The `NSNumber.unsignedIntegerValue` property now has the type `UInt` instead of `Int`, as do other methods and properties that use the `NSUInteger` type in Objective-C and whose names contain `unsigned...` Most other uses of `NSUInteger` in system frameworks are imported as `Int` as they were in Xcode 7. **(19134055)**
- Field getters and setters are now created for named unions imported from C. In addition, an initializer with a named parameter for the field is provided. For example, given the following Objective-C `typedef`:

  ```
  typedef union IntOrFloat {
    int intField;
  ```

```
  float floatField;
} IntOrFloat;
```

Importing this `typedef` into Swift generates the following interface:

```
struct IntOrFloat {
  var intField: Int { get set }
  init(intField: Int)

  var floatField: Float { get set }
  init(floatField: Float)
}
```

(19660119)

- Bitfield members of C structs are now imported into Swift. **(21702107)**
- The type `dispatch_block_t` now refers to the type `@convention(block) () -> Void`, as it did in Swift 1.2. This change allows programs using `dispatch_block_create` to work as expected, solving an issue that surfaced in Xcode 7.0 with Swift 2.0.

  Note: Converting to a Swift closure value and back is not guaranteed to preserve the identity of a `dispatch_block_t`. **(22432170)**
- Editing a file does not trigger a recompile of files that depend upon it if the edits only modify declarations marked private. **(22239821)**
- Expressions interpolated in strings may now contain string literals. For example, `My name is \(attributes["name"]!)` is now a valid expression. **(14050788)**
- Error messages produced when the type checker cannot solve its constraint system continue to improve in many cases.

  For example, errors in the body of generic closures (for instance, the argument closure to `map`) are much more usefully diagnosed. **(18835890)**
- Conversions between function types are supported, exhibiting covariance in function result types and contravariance in function parameter types. For example, it is legal to assign a function of type `Any -> Int` to a variable of type `String -> Any`. **(19517003)**

Swift 2.0

2015-09-17 (Xcode 7.0)

Swift Language Features

- New `defer` statement. This statement runs cleanup code when the scope is exited, which is particularly useful in conjunction with the new error handling model. For example:

```
func xyz() throws {
   let f = fopen("x.txt", "r")
   defer { fclose(f) }
   try foo(f)                    // f is closed if an error is propagated.
   let f2 = fopen("y.txt", "r")
   defer { fclose(f2) }
   try bar(f, f2)                // f2 is closed, then f is closed if an error is propa
}                                // f2 is closed, then f is closed on a normal path
```

(17302850)

- Printing values of certain `enum` types shows the enum `case` and payload, if any. This is not supported for `@objc` enums or certain enums with multiple payloads. **(18334936)**

- You can specify availability information on your own declarations with the `@available()` attribute.

  For example:

  ```
  @available(iOS 8.0, OSX 10.10, *)
  func startUserActivity() -> NSUserActivity {
    ...
  }
  ```

  This code fragment indicates that the `startUserActivity()` method is available on iOS 8.0+, on OS X v10.10+, and on all versions of any other platform. **(20938565)**

- A new `@nonobjc` attribute is introduced to selectively suppress ObjC export for instance members that would otherwise be `@objc`. **(16763754)**

- Nongeneric classes may now inherit from generic classes. **(15520519)**

- Public extensions of generic types are now permitted.

  ```
  public extension Array { ... }
  ```

  (16974298)

- Enums now support multiple generic associated values, for example:

  ```
  enum Either<T, U> {
     case Left(T), Right(U)
  }
  ```

  (15666173)

- **Protocol extensions**: Extensions can be written for protocol types. With these extensions, methods and properties can be added to any type that conforms to a particular protocol, allowing you to reuse more of your code. This leads to more natural caller side "dot" method syntax that follows the principle of “fluent interfaces” and that makes the definition of generic code simpler (reducing “angle bracket blindness”). **(11735843)**

- **Protocol default implementations**: Protocols can have default implementations for requirements specified in a protocol extension, allowing “mixin” or “trait” like patterns.

- **Availability checking**. Swift reports an error at compile time if you call an API that was introduced in a version of the operating system newer than the currently selected deployment target.

  To check whether a potentially unavailable API is available at runtime, use the new `#available()` condition in an if or guard statement. For example:

  ```
  if #available(iOS 8.0, OSX 10.10, *) {
    // Use Handoff APIs when available.
    let activity =
      NSUserActivity(activityType:"com.example.ShoppingList.view")
    activity.becomeCurrent()
  } else {
    // Fall back when Handoff APIs not available.
  }
  ```

  (14586648)

- Native support for C function pointers: C functions that take function pointer arguments can be called using closures or global functions, with the restriction that the closure must not capture any of its local context. For example, the standard C qsort function can be invoked as follows:

  ```
  var array = [3, 14, 15, 9, 2, 6, 5]
  qsort(&array, array.count, sizeofValue(array[0])) { a, b in
    return Int32(UnsafePointer<Int>(a).memory - UnsafePointer<Int>(b).memory)
  }
  print(array)
  ```

  (16339559)

- **Error handling**. You can create functions that `throw`, `catch`, and manage errors in Swift.

  Using this capability, you can surface and deal with recoverable errors, such as “file-not-found” or network timeouts. Swift’s error handling interoperates with `NSError`. **(17158652)**

- **Testability**: Tests of Swift 2.0 frameworks and apps are written without having to make internal routines public.

  Use `@testable import {ModuleName}` in your test source code to make all public and internal routines usable. The app or framework target needs to be compiled with the `Enable Testability` build setting set to `Yes`. The `Enable Testability` build setting should be used only in your

Debug configuration, because it prohibits optimizations that depend on not exporting internal symbols from the app or framework. **(17732115)**

- if statements can be labeled, and labeled break statements can be used as a jump out of the matching if statement.

  An unlabeled break does not exit the if statement. It exits the enclosing loop or switch statement, or it is illegal if none exists. (19150249)

- A new `x?` pattern can be used to pattern match against optionals as a synonym for `.Some(x)`. **(19382878)**
- Concatenation of Swift string literals, including across multiple lines, is now a guaranteed compile-time optimization, even at `-Onone`. **(19125926)**
- Nested functions can now recursively reference themselves and other nested functions. **(11266246)**
- Improved diagnostics:
  - A warning has been introduced to encourage the use of let instead of var when appropriate.
  - A warning has been introduced to signal unused variables.
  - Invalid mutation diagnostics are more precise.
  - Unreachable switch cases cause a warning.
  - The switch statement "exhaustiveness checker" is smarter. **(15975935,20130240)**
- Failable convenience initializers are allowed to return `nil` before calling `self.init`.

  Designated initializers still must initialize all stored properties before returning `nil`; this is a known limitation. **(20193929)**

- A new `readLine()` function has been added to the standard library. **(15911365)**
- **SIMD Support**: Clang extended vectors are imported and usable in Swift.

  This capability enables many graphics and other low-level numeric APIs (for example, `simd.h`) to be usable in Swift.

- New `guard` statement: This statement allows you to model an early exit out of a scope.

  For example:

  ```
  guard let z = bar() else { return }
  use(z)
  ```

  The `else` block is required to exit the scope (for example, with `return`, `throw`, `break`, `continue`, and so forth) or end in a call to a `@noreturn` function. **(20109722)**

- Improved pattern matching: `switch/case` pattern matching is available to many new conditional control flow statements, including `if/case`, `while/case`, `guard/case`, and `for-in/case`. `for-in` statements can also have `where` clauses, which combine to support many of the features of list comprehensions in other languages.
- A new `do` statement allows scopes to be introduced with the `do` statement.

  For example:

  ```
  do {
      //new scope
      do {
          //another scope
      }
  }
  ```

Swift Enhancements and Changes

- A new keyword `try?` has been added to Swift.

  `try?` attempts to perform an operation that may throw. If the operation succeeds, the result is wrapped in an optional; if it fails (that is, if an error is thrown), the result is `nil` and the error is discarded.

  For example:

  ```
  func produceGizmoUsingTechnology() throws -> Gizmo { ... }
  func produceGizmoUsingMagic() throws -> Gizmo { ... }

  if let result = try? produceGizmoUsingTechnology() { return result }
  if let result = try? produceGizmoUsingMagic() { return result }
  print("warning: failed to produce a Gizmo in any way")
  return nil
  ```

  `try?` always adds an extra level of `Optional` to the result type of the expression being evaluated. If a throwing function's normal return type is `Int?`, the result of calling it with `try?` will be `Int??`, or `Optional<Optional<Int>>`. **(21692467)**
- Type names and enum cases now print and convert to `String` without qualification by default. `debugPrint` or `String(reflecting:)` can still be used to get fully qualified names. For example:

  ```
  enum Fruit { case Apple, Banana, Strawberry }
  print(Fruit.Apple)      // "Apple"
  debugPrint(Fruit.Apple) // "MyApp.Fruit.Apple")
  ```

  (21788604)
- C `typedef`s of block types are imported as `typealias`s for Swift closures.

The primary result of this is that `typedef`s for blocks with a parameter of type `BOOL` are imported as closures with a parameter of type `Bool` (rather than `ObjCBool` as in the previous release). This matches the behavior of block parameters to imported Objective-C methods. **(22013912)**

- The type `Boolean` in `MacTypes.h` is imported as `Bool` in contexts that allow bridging between Swift and Objective-C types.

  In cases where the representation is significant, `Boolean` is imported as a distinct `DarwinBoolean` type, which is `BooleanLiteralConvertible` and can be used in conditions (much like the `ObjCBool` type). **(19013551)**

- Fields of C structs that are marked `__unsafe_unretained` are presented in Swift using `Unmanaged`.

  It is not possible for the Swift compiler to know if these references are really intended to be strong (+1) or unretained (+0). **(19790608)**

- The `NS_REFINED_FOR_SWIFT` macro can be used to move an Objective-C declaration aside to provide a better version of the same API in Swift, while still having the original implementation available. (For example, an Objective-C API that takes a `Class` could offer a more precise parameter type in Swift.)

  The `NS_REFINED_FOR_SWIFT` macro operates differently on different declarations:

  - `init` methods will be imported with the resulting Swift initializer having `__` prepended to its first external parameter name.

    ```
    // Objective-C
    - (instancetype)initWithClassName:(NSString *)name NS_REFINED_FOR_SWIFT;
    ```

    ```
    // Swift
    init(__className: String)
    ```

  - Other methods will be imported with `__` prepended to their base name.

    ```
    // Objective-C
    - (NSString *)displayNameForMode:(DisplayMode)mode NS_REFINED_FOR_SWIFT;
    ```

    ```
    // Swift
    func __displayNameForMode(mode: DisplayMode) -> String
    ```

  - Subscript methods will be treated like any other methods and will not be imported as subscripts.

  - Other declarations will have `__` prepended to their name.

    ```
    // Objective-C
    @property DisplayMode mode NS_REFINED_FOR_SWIFT;
    ```

```
// Swift
var __mode: DisplayMode { get set }
```

(20070465)

- Xcode provides context-sensitive code completions for enum elements and option sets when using the shorter dot syntax. **(16659653)**

- The `NSManaged` attribute can be used with methods as well as properties, for access to Core Data's automatically generated Key-Value-Coding-compliant to-many accessors.

  ```
  @NSManaged var employees: NSSet

  @NSManaged func addEmployeesObject(employee: Employee)
  @NSManaged func removeEmployeesObject(employee: Employee)
  @NSManaged func addEmployees(employees: NSSet)
  @NSManaged func removeEmployees(employees: NSSet)
  ```

  These can be declared in your `NSManagedObject` subclass. **(17583057)**

- The grammar has been adjusted so that lines beginning with '.' are always parsed as method or property lookups following the previous line, allowing for code formatted like this to work:

  ```
  foo
    .bar
    .bas = 68000
  ```

  It is no longer possible to begin a line with a contextual static member lookup (for example, to say `.staticVar = MyType()`). **(20238557)**

- Code generation for large `struct` and `enum` types has been improved to reduce code size. **(20598289)**

- Nonmutating methods of structs, enums, and protocols may now be partially applied to their self parameter:

  ```
  let a: Set<Int> = [1, 2, 3]
  let b: [Set<Int>] = [[1], [4]]
  b.map(a.union) // => [[1, 2, 3], [1, 2, 3, 4]]
  ```

  (21091944)

- Swift documentation comments recognize a new top-level list item: `- Throws: ...`

  This item is used to document what errors can be thrown and why. The documentation appears alongside parameters and return descriptions in Xcode QuickHelp. **(21621679)**

- Unnamed parameters now require an explicit `_:` to indicate that they are unnamed. For example, the following is now an error:

```
func f(Int) { }
```

and must be written as:

```
func f(_: Int) { }
```

This simplifies the argument label model and also clarifies why cases like `func f((a: Int, b: Int))` do not have parameters named `a` and `b`. **(16737312)**

- It is now possible to append a tuple to an array. **(17875634)**
- The ability to refer to the 0 element of a scalar value (producing the scalar value itself) has been removed. **(17963034)**
- Variadic parameters can now appear anywhere in the parameter list for a function or initializer. For example:

  ```
  func doSomethingToValues(values: Int... , options: MyOptions = [], fn: (Int) -&gt; Voi
  ```

  (20127197)
- Generic subclasses of Objective-C classes are now supported. **(18505295)**
- If an element of an enum with string raw type does not have an explicit raw value, it will default to the text of the enum's name. For example:

  ```
  enum WorldLayer : String {
      case Ground, BelowCharacter, Character
  }
  ```

  is equivalent to:

  ```
  enum WorldLayer : String {
      case Ground = "Ground"
      case BelowCharacter = "BelowCharacter"
      case Character = "Character"
  }
  ```

  (15819953)
- The `performSelector` family of APIs is now available for Swift code. **(17227475)**
- When delegating or chaining to a failable initializer (for example, with `self.init(...)` or `super.init(...)`), one can now force-unwrap the result with `!`. For example:

  ```
  extension UIImage {
    enum AssetIdentifier: String {
      case Isabella
      case William
      case Olivia
    }
  ```

```
  convenience init(assetIdentifier: AssetIdentifier) {
    self.init(named: assetIdentifier.rawValue)!
  }
}
```

(18497407)

- Initializers can now be referenced like static methods by referring to `.init` on a static type reference or type object. For example:

  ```
  let x = String.init(5)
  let stringType = String.self
  let y = stringType.init(5)

  let oneTwoThree = [1, 2, 3].map(String.init).reduce("", combine: +)
  ```

  `.init` is still implicit when constructing using a static type, as in `String(5)`. `.init` is required when using dynamic type objects or when referring to the initializer as a function value. **(21375845)**

- Enums and cases can now be marked indirect, which causes the associated value for the enum to be stored indirectly, allowing for recursive data structures to be defined. For example:

  ```
    enum List<T> {
    case Nil
    indirect case Cons(head: T, tail: List<T>)
  }

  indirect enum Tree<T> {
    case Leaf(T)
    case Branch(left: Tree<T>, right: Tree<T>)
  }
  ```

  (21643855)

- Formatting for Swift expression results has changed significantly when using `po` or `expr -O`. Customization that was introduced has been refined in the following ways:

  - The formatted summary provided by either `debugDescription` or `description` methods will always be used for types that conform to `CustomDebugStringConvertible` or `CustomStringConvertible` respectively. When neither conformance is present, the type name is displayed and reference types also display the referenced address to more closely mimic existing behavior for Objective-C classes.

  - Value types such as enums, tuples, and structs display all members indented below the summary by default, while reference types will not. This behavior can be customized for all types by implementing `CustomReflectable`.

These output customizations can be bypassed by using `p` or `expr` without the `-O` argument to provide a complete list of all fields and their values. **(21463866)**

- Properties and methods using `Unmanaged` can now be exposed to Objective-C. **(16832080)**
- Applying the `@objc` attribute to a class changes that class's compile-time name in the target's generated Objective-C header as well as changing its runtime name. This applies to protocols as well. For example:

  ```
  // Swift
  @objc(MyAppDelegate)
  class AppDelegate : NSObject, UIApplicationDelegate {
    // ...
  }
  ```

  ```
  // Objective-C
  @interface MyAppDelegate : NSObject <UIApplicationDelegate>
    // ...
  @end
  ```

  (17469485)
- Collections containing types that are not Objective-C compatible are no longer considered Objective-C compatible types themselves.

  For example, previously `Array<SwiftClassType>` was permitted as the type of a property marked `@objc`; this is no longer the case. **(19787270)**
- Generic subclasses of Objective-C classes, as well as nongeneric classes that inherit from such a class, require runtime metadata instantiation and cannot be directly named from Objective-C code.

  When support for generic subclasses of Objective-C classes was first added, the generated Objective-C bridging header erroneously listed such classes, which, when used, could lead to incorrect runtime behavior or compile-time errors. This has been fixed.

  The behavior of the `@objc` attribute on a class has been clarified such that applying `@objc` to a class which cannot appear in a bridging header is now an error.

  Note that this change does not result in a change of behavior with valid code because members of a class are implicitly `@objc` if any superclass of the class is an `@objc` class, and all `@objc` classes must inherit from NSObject. **(21342574)**

- The performance of `-Onone` (debug) builds has been improved by using prespecialized instances of generics in the standard library. It produces significantly faster executables in debug builds in many cases, without impacting compile time. **(20486658)**

- `AnyObject` and `NSObject` variables that refer to class objects can be cast back to class object types. For example, this code succeeds:

  ```
  let x: AnyObject = NSObject.self
  let y = x as! NSObject.Type
  ```

  Arrays, dictionaries, and sets that contain class objects successfully bridge with `NSArray`, `NSDictionary`, and `NSSet` as well. Objective-C APIs that provide `NSArray<Class> *` objects, such as `-[NSURLSessionConfiguration protocolClasses]`, now work correctly when used in Swift. **(16238475)**

- `print()` and reflection via Mirrors is able to report both the current case and payload for all enums with multiple payload types. The only remaining enum types that do not support reflection are `@objc` enums and enums imported from C. **(21739870)**

- Enum cases with payloads can be used as functions. For example:

  ```
  enum Either<T, U> { case Left(T), Right(U) }
  let lefts: [Either<Int, String>] = [1, 2, 3].map(Either.Left)
  let rights: [Either<Int, String>] = ["one", "two", "three"].map(Either.Right)
  ```

  (19091028)

- `ExtensibleCollectionType` has been folded into `RangeReplaceableCollectionType`. In addition, default implementations have been added as methods, which should be used instead of the free Swift module functions related to these protocols. **(18220295)**

Swift Standard Library

- The standard library moved many generic global functions (such as `map`, `filter`, and `sort`) to be methods written with protocol extensions. This allows those methods to be pervasively available on all sequence and collection types and allowed the removal of the global functions.

- Deprecated enum elements no longer affect the names of nondeprecated elements when an Objective-C enum is imported into Swift. This may cause the Swift names of some enum elements to change. **(17686122)**

- All enums imported from C are `RawRepresentable`, including those not declared with `NS_ENUM` or `NS_OPTIONS`. As part of this change, the value property of such enums has been renamed `rawValue`. **(18702016)**

- Swift documentation comments use a syntax based on the Markdown format, aligning them with rich comments in playgrounds.
  - Outermost list items are interpreted as special fields and are highlighted in Xcode QuickHelp.
  - There are two methods of documenting parameters: parameter outlines and separate parameter fields. You can mix and match these forms as you see fit in any order or continuity throughout the doc comment. Because these are parsed as list items, you can nest arbitrary content underneath them.
  - Parameter outline syntax:

    ```
    - Parameters:
      - x: ...
      - y: ...
    ```

  - Separate parameter fields:

    ```
    - parameter x: ...
    - parameter y: ...
    ```

  - Documenting return values:

    ```
    - returns: ...
    ```

  Other special fields are highlighted in QuickHelp, as well as rendering support for all of Markdown. (20180161)

- The `CFunctionPointer<T -> U>` type has been removed. C function types are specified using the new `@convention(c)` attribute. Like other function types, `@convention(c) T -> U` is not nullable unless made optional. The `@objc_block` attribute for specifying block types has also been removed and replaced with `@convention(block)`.

- Methods and functions have the same rules for parameter names. You can omit providing an external parameter name with `_`. To further simplify the model, the shorthand `#` for specifying a parameter name has been removed, as have the special rules for default arguments.

  ```
  // Declaration
    func printFunction(str: String, newline: Bool)
    func printMethod(str: String, newline: Bool)
    func printFunctionOmitParameterName(str: String, _  newline: Bool)

  // Call
    printFunction("hello", newline: true)
    printMethod("hello", newline: true)
    printFunctionOmitParameterName("hello", true)
  ```

  (17218256)

- `NS_OPTIONS` types get imported as conforming to the `OptionSetType` protocol, which presents a set-like interface for options. Instead of using bitwise operations such as:

  ```
  // Swift 1.2:
  object.invokeMethodWithOptions(.OptionA | .OptionB)
  object.invokeMethodWithOptions(nil)

  if options @ .OptionC == .OptionC {
    // .OptionC is set
  }
  ```

  Option sets support set literal syntax, and set-like methods such as contains:

  ```
  object.invokeMethodWithOptions([.OptionA, .OptionB])
  object.invokeMethodWithOptions([])

  if options.contains(.OptionC) {
    // .OptionC is set
  }
  ```

  A new option set type can be written in Swift as a struct that conforms to the `OptionSetType` protocol. If the type specifies a `rawValue` property and option constants as `static let` constants, the standard library will provide default implementations of the rest of the option set API:

  ```
  struct MyOptions: OptionSetType {
    let rawValue: Int

    static let TuringMachine  = MyOptions(rawValue: 1)
    static let LambdaCalculus = MyOptions(rawValue: 2)
    static let VonNeumann     = MyOptions(rawValue: 4)
  }

  let churchTuring: MyOptions = [.TuringMachine, .LambdaCalculus]
  ```

  (18069205)

- Type annotations are no longer allowed in patterns and are considered part of the outlying declaration. This means that code previously written as:

  ```
  var (a : Int, b : Float) = foo()
  ```

  needs to be written as:

  ```
  var (a, b) : (Int, Float) = foo()
  ```

  if an explicit type annotation is needed. The former syntax was ambiguous with tuple element labels. **(20167393)**

- The `do`/`while` loop is renamed to `repeat`/`while` to make it obvious whether a statement is a loop from its leading keyword.

In Swift 1.2:

```
do {
...
} while <condition>
```

In Swift 2.0:

```
repeat {
...
} while <condition>
```

(20336424)

- `forEach` has been added to `SequenceType`. This lets you iterate over elements of a sequence, calling a body closure on each. For example:

  ```
  (0..<10).forEach {
    print($0)
  }
  ```

  This is very similar to the following:

  ```
  for x in 0..<10 {
    print(x)
  }
  ```

  But take note of the following differences:

  - Unlike for-in loops, you can't use `break` or `continue` to exit the current call of the body closure or skip subsequent calls.
  - Also unlike for-in loops, using `return` in the body closure only exits from the current call to the closure, not any outer scope, and won't skip subsequent calls.

  (18231840)

- The `Word` and `UWord` types have been removed from the standard library; use `Int` and `UInt` instead. **(18693488)**
- Most standard library APIs that take closures or `@autoclosure` parameters now use `rethrows`. This allows the closure parameters to methods like `map` and `filter` to throw errors, and allows short-circuiting operators like `&&`, `||`, and `??` to work with expressions that may produce errors. **(21345565)**
- SIMD improvements: Integer vector types in the simd module now only support unchecked arithmetic with wraparound semantics using the `&+`, `&-`, and `&*` operators, in order to better support the machine model for vectors. The `+`, `-`, and `*` operators are unavailable on integer vectors, and Xcode automatically suggests replacing them with the wrapping operators.

Code generation for vector types in the simd module has been improved to better utilize vector hardware, leading to dramatically improved performance in many cases. **(21574425)**

- All `CollectionType` objects are now sliceable. `SequenceType` now has a notion of `SubSequence`, which is a type that represents only some of the values but in the same order. For example, the `ArraySubSequence` type is `ArraySlice`, which is an efficient view on the `Array` type's buffer that avoids copying as long as it uniquely references the `Array` from which it came.

  The following free Swift functions for splitting/slicing sequences have been removed and replaced by method requirements on the `SequenceType` protocol with default implementations in protocol extensions. `CollectionType` has specialized implementations, where possible, to take advantage of efficient access of its elements.

```
/// Returns the first `maxLength` elements of `self`,
/// or all the elements if `self` has fewer than `maxLength` elements.
prefix(maxLength: Int) -> SubSequence

/// Returns the last `maxLength` elements of `self`,
/// or all the elements if `self` has fewer than `maxLength` elements.
suffix(maxLength: Int) -> SubSequence

/// Returns all but the first `n` elements of `self`.
dropFirst(n: Int) -> SubSequence

/// Returns all but the last `n` elements of `self`.
dropLast(n: Int) -> SubSequence

/// Returns the maximal `SubSequence`s of `self`, in order, that
/// don't contain elements satisfying the predicate `isSeparator`.
split(maxSplits maxSplits: Int, allowEmptySlices: Bool, @noescape isSeparator: (Generat
```

  The following convenience extension is provided for `split`:

```
split(separator: Generator.Element, maxSplit: Int, allowEmptySlices: Bool) -> [SubSeque
```

  Also, new protocol requirements and default implementations on `CollectionType` are now available:

```
/// Returns `self[startIndex..<end]`
prefixUpTo(end: Index) -> SubSequence

/// Returns `self[start..<endIndex]`
suffixFrom(start: Index) -> SubSequence

/// Returns `prefixUpTo(position.successor())`
```

`prefixThrough(position: Index) -> SubSequence`

(21663830)

- The `print` and `debugPrint` functions are improved:
  - Both functions have become variadic, and you can print any number of items with a single call.
  - `separator: String = " "` was added so you can control how the items are separated.
  - `terminator: String = "\n"` replaced `appendNewline: bool = true.` With this change, `print(x, appendNewline: false)` is expressed as `print(x, terminator: "")`.
  - For the variants that take an output stream, the argument label `toStream` was added to the stream argument.

  The `println` function from Swift 1.2 has been removed. **(21788540)**

- For consistency and better composition of generic code, `ArraySlice` indices are no longer always zero-based but map directly onto the indices of the collection they are slicing and maintain that mapping even after mutations.

  Before:

```
var a = Array(0..<10)
var s = a[5..<10]
s.indices        // 0..<5
s[0] = 111
s                // [111, 6, 7, 8, 9]
s.removeFirst()
s.indices        // 1..<5
```

  After:

```
var a = Array(0..<10)
var s = a[5..<10]
s.indices        // 5..<10
s[5] = 99
s                // [99, 6, 7, 8, 9]
s.removeFirst()
s.indices        // 6..<10
```

  Rather than define variants of collection algorithms that take explicit subrange arguments, such as `a.sortSubrangeInPlace(3..<7)`, the Swift standard library provides “slicing,” which composes well with algorithms. This enables you to write `a[3..<7].sortInPlace()`, for example. With most collections, these algorithms compose naturally.

  For example, before this change was incorporated:

```
extension MyIntCollection {
  func prefixThroughFirstNegativeSubrange() -> SubSequence {
    // Find the first negative element
    let firstNegative = self.indexOf { $0 < 0 } ?? endIndex

    // Slice off non-negative prefix
    let startsWithNegative = self.suffixFrom(firstNegative)

    // Find the first non-negative position in the slice
    let end = startsWithNegative.indexOf { $0 >= 0 } ?? endIndex
    return self[startIndex..<end]
  }
}
```

The above code would work for any collection of `Int`s unless the collection is an `Array<Int>`. Unfortunately, when array slice indices are zero-based, the last two lines of the method need to change to:

```
let end = startsWithNegative.indexOf { $0 >= 0 }
  ?? startsWithNegative.endIndex
return self[startIndex..<end + firstNegative]
```

These differences made working with slices awkward, error-prone, and nongeneric.

After this change, Swift collections start to provide a guarantee that, at least until there is a mutation, slice indices are valid in the collection from which they were sliced, and refer to the same elements. **(21866825)**

- The method `RangeReplaceableCollectionType.extend()` was renamed to `appendContentsOf()`, and the `splice()` method was renamed to `insertContentsOf()`. **(21972324)**
- `find` has been renamed to `indexOf()`, sort has been renamed to `sortInPlace()`, and `sorted()` becomes `sort()`.
- `String.toInt()` has been renamed to a failable `Int(String)` initializer, since initialization syntax is the preferred style for type conversions.
- `String` no longer conforms to `SequenceType` in order to prevent non-Unicode correct sequence algorithms from being prominently available on String. To perform grapheme-cluster-based, UTF-8-based, or UTF-16-based algorithms, use the `.characters`, `.utf8`, and `.utf16` projections respectively.
- Generic functions that declare type parameters not used within the generic function's type produce a compiler error. For example:

```
func foo<T>() { } // error: generic parameter 'T' is not used in function signature
```

- The `Dictionary.removeAtIndex(_:)` method now returns the key-value pair being removed as a two-element tuple (rather than returning `Void`). Similarly, the `Set.removeAtIndex(_:)` method returns the element being removed. **(20299881)**
- Generic parameters on types in the Swift standard library have been renamed to reflect the role of the types in the API. For example, `Array<T>` became `Array<Element>`, `UnsafePointer<T>` became `UnsafePointer<Memory>`, and so forth. **(21429126)**
- The `SinkType` protocol and `SinkOf` struct have been removed from the standard library in favor of `(T) -> ()` closures. **(21663799)**

Swift 1.2

2015-04-08 (Xcode 6.3)

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 "`let`s 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.

  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 permitted
```

Note that these Cocoa types in Objective-C headers are still automatically bridged to their corresponding Swift type, which means that code is only affected if it is explicitly referencing (for example) `NSString` in a Swift source file. It is recommended you use the corresponding Swift types (for example, `String`) directly unless you are doing something advanced, like implementing a subclass in the class cluster. **(18311362)**

- The `@autoclosure` attribute is now an attribute on a parameter, not an attribute on the parameter's type.

  Where before you might have used:

  ```
  func assert(predicate : @autoclosure () -> Bool) {...}
  you now write this as:
  func assert(@autoclosure predicate : () -> Bool) {...}
  ```

  (15217242)

- The `@autoclosure` attribute on parameters now implies the new `@noescape` attribute.

- Curried function parameters can now specify argument labels.

  For example:

  ```
  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)
  ```

  (17237268)

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

  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'
}
```

(18391046, 18383574)

- The precedence of the Nil Coalescing Operator (`??`) has been raised to bind tighter than short-circuiting logical and comparison operators, but looser than as conversions and range operators. This provides more useful behavior for expressions like:

  ```
  if allowEmpty || items?.count ?? 0 > 0 {...}
  ```

- The `&/` and `&%` operators were removed, to simplify the language and improve consistency.

  Unlike the `&+`, `&-`, and `&*` operators, these operators did not provide two's-complement arithmetic behavior; they provided special case behavior for division, remainder by zero, and `Int.min/-1`. These tests should be written explicitly in the code as comparisons if needed. **(17926954)**

- Constructing a `UInt8` from an ASCII value now requires the ascii keyword parameter. Using non-ASCII unicode scalars will cause this initializer to trap. **(18509195)**

- The C `size_t` family of types are now imported into Swift as `Int`, since

Swift prefers sizes and counts to be represented as signed numbers, even if they are non-negative.

This change decreases the amount of explicit type conversion between `Int` and `UInt`, better aligns with `sizeof` returning `Int`, and provides safer arithmetic properties. **(18949559)**

- Classes that do not inherit from `NSObject` but do adopt an `@objc` protocol will need to explicitly mark those methods, properties, and initializers used to satisfy the protocol requirements as `@objc`.

  For example:

```
@objc protocol SomethingDelegate {
    func didSomething()
}

class MySomethingDelegate : SomethingDelegate {
    @objc func didSomething() { ... }
}
```

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.

  For example:

```
let animationCurve =
  unsafeBitCast(userInfo[UIKeyboardAnimationCurveUserInfoKey].integerValue,
  UIViewAnimationCurve.self)
can now be written instead as:
let animationCurve = UIViewAnimationCurve(rawValue:
  userInfo[UIKeyboardAnimationCurveUserInfoKey].integerValue)!
```

(19005771)

- 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.

  For example:

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

  (19321484)

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.

  The incremental build capability 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 afterwards should 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 (for instance, to avoid the optional unwrapping *"pyramid of doom"*).

  Further, `if-let` now also supports a single leading boolean condition along with optional binding `let` clauses. For example:

  ```
  if someValue > 42 && someOtherThing < 19, let a = getOptionalThing() where a > someValu
  }
  ```

  (19797158, 19382942)
- The `if-let` syntax has been extended to support a single leading boolean condition along with optional binding `let` clauses.

  For example:

  ```
  if someValue > 42 && someOtherThing < 19, let a = getOptionalThing() where a > someValu
  }
  ```

  (19797158)
- `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 such as:

  ```
  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.

  For example, this Swift declaration:

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

  imports into Objective-C as:

  ```
  // Objective-C
  typedef NS_ENUM(NSInteger, Bear) {
     BearBlack, BearGrizzly, BearPolar
  };
  ```

  (16967385)

- 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 items 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 that 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 `String`s. **(18018911)**

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

  ```
  toString(Int.self)           // prints "Swift.Int"
  println([Float].self)        // prints "Swift.Array&lt;Swift.Float&gt;"
  println((Int, String).self) // prints "(Swift.Int, Swift.String)"
  ```

  (18947381)

- 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()`.

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

  (16323038)

- 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)**
- The `@autoclosure` attribute has a second form, `@autoclosure(escaping)`, that provides the same caller-side syntax as `@autoclosure` but allows the resulting closure to escape in the implementation.

  For example:

  ```
  func lazyAssertion(@autoclosure(escaping) condition: () -> Bool,
                     message: String = "") {
    lazyAssertions.append(condition) // escapes
    }
  lazyAssertion(1 == 2, message: "fail eventually")
  ```

  (19499207)

Swift Performance

- A new compilation mode has been introduced for Swift called Whole Module Optimization. This option optimizes all of the files in a target together and enables better performance (at the cost of increased compile time). The new flag can be enabled in Xcode using the `Whole Module Optimization` build setting or by using the `swiftc` command line tool with the flag `-whole-module-optimization`. **(18603795)**

Swift Standard Library Enhancements and Changes

- `flatMap` was added to the standard library. `flatMap` is the function that maps a function over something and returns the result flattened one level. `flatMap` has many uses, such as to flatten an array:

  ```
  [[1,2],[3,4]].flatMap { $0 }
  ```

  or to chain optionals with functions:

  ```
  [[1,2], [3,4]].first.flatMap { find($0, 1) }
  ```

  (19881534)
- The function `zip` was added. It joins two sequences together into one sequence of tuples. **(17292393)**
- `utf16Count` is removed from `String`. Instead use count on the `UTF16` view of the `String`.

  For example:

  ```
  count(string.utf16)
  ```

  (17627758)

Swift 1.1

2014-12-02 (Xcode 6.1.1)

- Class methods and initializers that satisfy protocol requirements now properly invoke subclass overrides when called in generic contexts. For example:

  ```
  protocol P {
    class func foo()
  }

  class C: P {
    class func foo() { println("C!") }
  }

  class D: C {
    override class func foo() { println("D!") }
  }

  func foo<T: P>(x: T) {
    x.dynamicType.foo()
  }

  foo(C()) // Prints "C!"
  foo(D()) // Used to incorrectly print "C!", now prints "D!"
  ```

  (18828217)

2014-10-09 (Xcode 6.1)

- Values of type `Any` can now contain values of function type. **(16406907)**
- Documentation for the standard library (displayed in quick help and in the synthesized header for the Swift module) is improved. **(16462500)**
- Class properties don't need to be marked final to avoid `O(n)` mutations on value semantic types. **(17416120)**
- Casts can now be performed between `CF` types (such as `CFString`, `CGImage`, and `SecIdentity`) and AnyObject. Such casts will always succeed at run-time. For example:

  ```
  var cfStr: CFString = ...
  var obj: AnyObject = cfStr as AnyObject
  var cfStr = obj as CFString
  ```

  (18088474)
- `HeapBuffer<Value, Element>`, `HeapBufferStorage<Value, Element>`, and `OnHeap<Value>` were never really useful, because their APIs

were insufficiently public. They have been replaced with a single class, `ManagedBuffer<Value, Element>`. See also the new function `isUniquelyReferenced(x)` which is often useful in conjunction with `ManagedBuffer`.

- The `Character` enum has been turned into a struct, to avoid exposing its internal implementation details.
- The `countElements` function has been renamed `count`, for better consistency with our naming conventions.
- Mixed-sign addition and subtraction operations, that were unintentionally allowed in previous versions, now cause a compilation error.
- OS X apps can now apply the `@NSApplicationMain` attribute to their app delegate class in order to generate an implicit `main` for the app. This works like the `@UIApplicationMain` attribute for iOS apps.
- Objective-C `init` and factory methods are now imported as failable initializers when they can return `nil`. In the absence of information about a potentially-`nil` result, an Objective-C `init` or factory method will be imported as `init!`.

  As part of this change, factory methods that have NSError** parameters, such as `+[NSString stringWithContentsOfFile:encoding:error:]`, will now be imported as (failable) initializers, e.g.,

  ```
  init?(contentsOfFile path: String,
        encoding: NSStringEncoding,
        error: NSErrorPointer)
  ```
- Nested classes explicitly marked `@objc` will now properly be included in a target's generated header as long as the containing context is also (implicitly or explicitly) `@objc`. Nested classes not explicitly marked `@objc` will never be printed in the generated header, even if they extend an Objective-C class.
- All of the `*LiteralConvertible` protocols, as well as `StringInterpolationConvertible`, now use initializers for their requirements rather than static methods starting with `convertFrom`. For example, `IntegerLiteralConvertible` now has the following initializer requirement:

  ```
  init(integerLiteral value: IntegerLiteralType)
  ```

  Any type that previously conformed to one of these protocols will need to replace its `convertFromXXX` static methods with the corresponding initializer.

Swift 1.0

2014-09-15 (Xcode 6.0)

- Initializers can now fail by returning `nil`. A failable initializer is declared with `init?` (to return an explicit optional) or `init!` (to return an implicitly-unwrapped optional). For example, you could implement `String.toInt` as a failable initializer of `Int` like this:

  ```swift
  extension Int {
    init?(fromString: String) {
      if let i = fromString.toInt() {
        // Initialize
        self = i
      } else {
        // Discard self and return 'nil'.
        return nil
      }
    }
  }
  ```

  The result of constructing a value using a failable initializer then becomes optional:

  ```swift
  if let twentytwo = Int(fromString: "22") {
    println("the number is \(twentytwo)")
  } else {
    println("not a number")
  }
  ```

  In the current implementation, struct and enum initializers can return `nil` at any point inside the initializer, but class initializers can only return `nil` after all of the stored properties of the object have been initialized and `self.init` or `super.init` has been called. If `self.init` or `super.init` is used to delegate to a failable initializer, then the `nil` return is implicitly propagated through the current initializer if the called initializer fails.

- The `RawRepresentable` protocol that enums with raw types implicitly conform to has been redefined to take advantage of failable initializers. The `fromRaw(RawValue)` static method has been replaced with an initializer `init?(rawValue: RawValue)`, and the `toRaw()` method has been replaced with a `rawValue` property. Enums with raw types can now be used like this:

  ```swift
  enum Foo: Int { case A = 0, B = 1, C = 2 }
  let foo = Foo(rawValue: 2)! // formerly 'Foo.fromRaw(2)!'
  println(foo.rawValue) // formerly 'foo.toRaw()'
  ```

2014-09-02

- Characters can no longer be concatenated using `+`. Use `String(c1) + String(c2)` instead.

2014-08-18

- When force-casting between arrays of class or `@objc` protocol types using `a as [C]`, type checking is now deferred until the moment each element is accessed. Because bridging conversions from NSArray are equivalent to force-casts from `[NSArray]`, this makes certain Array round-trips through Objective-C code `O(1)` instead of `O(N)`.

2014-08-04

- `RawOptionSetType` now implements `BitwiseOperationsType`, so imported `NS_OPTIONS` now support the bitwise assignment operators `|=`, `&=`, and `^=`. It also no longer implements `BooleanType`; to check if an option set is empty, compare it to `nil`.

- Types implementing `BitwiseOperationsType` now automatically support the bitwise assignment operators `|=`, `&=`, and `^=`.

- Optionals can now be coalesced with default values using the `??` operator. `??` is a short-circuiting operator that takes an optional on the left and a non-optional expression on the right. If the optional has a value, its value is returned as a non-optional; otherwise, the expression on the right is evaluated and returned:

  ```
  var sequence: [Int] = []
  sequence.first ?? 0 // produces 0, because sequence.first is nil
  sequence.append(22)
  sequence.first ?? 0 // produces 22, the value of sequence.first
  ```

- The optional chaining `?` operator can now be mutated through, like `!`. The assignment and the evaluation of the right-hand side of the operator are conditional on the presence of the optional value:

  ```
  var sequences = ["fibonacci": [1, 1, 2, 3, 4], "perfect": [6, 28, 496]]
  sequences["fibonacci"]?[4]++ // Increments element 4 of key "fibonacci"
  sequences["perfect"]?.append(8128) // Appends to key "perfect"

  sequences["cubes"]?[3] = 3*3*3 // Does nothing; no "cubes" key
  ```

  Note that optional chaining still flows to the right, so prefix increment operators are *not* included in the chain, so this won't type-check:

  ```
  ++sequences["fibonacci"]?[4] // Won't type check, can't '++' Int?
  ```

2014-07-28

- The swift command line interface is now divided into an interactive driver `swift`, and a batch compiler `swiftc`:

  ```
  swift [options] input-file [program-arguments]
    Runs the script 'input-file' immediately, passing any program-arguments
    to the script. Without any input files, invokes the repl.

  swiftc [options] input-filenames
    The familiar swift compiler interface: compiles the input-files according
    to the mode options like -emit-object, -emit-executable, etc.
  ```

- For greater clarity and explicitness when bypassing the type system, `reinterpretCast` has been renamed `unsafeBitCast`, and it has acquired a (required) explicit type parameter. So

  ```
  let x: T = reinterpretCast(y)
  ```

  becomes

  ```
  let x = unsafeBitCast(y, T.self)
  ```

- Because their semantics were unclear, the methods `asUnsigned` (on the signed integer types) and `asSigned` (on the unsigned integer types) have been replaced. The new idiom is explicit construction of the target type using the `bitPattern:` argument label. So,

  ```
  myInt.asUnsigned()
  ```

  has become

  ```
  UInt(bitPattern: myInt)
  ```

- To better follow Cocoa naming conventions and to encourage immutability, The following pointer types were renamed:

Old Name	New Name
`UnsafePointer<T>`	`UnsafeMutablePointer<T>`
`ConstUnsafePointer<T>`	`UnsafePointer<T>`
`AutoreleasingUnsafePointer<T>`	`AutoreleasingUnsafeMutablePointer<T>`

  Note that the meaning of `UnsafePointer` has changed from mutable to immutable. As a result, some of your code may fail to compile when assigning to an `UnsafePointer.memory` property. The fix is to change your `UnsafePointer<T>` into an `UnsafeMutablePointer<T>`.

- The optional unwrapping operator `x!` can now be assigned through, and mutating methods and operators can be applied through it:

```
var x: Int! = 0
x! = 2
x!++

// Nested dictionaries can now be mutated directly:
var sequences = ["fibonacci": [1, 1, 2, 3, 0]]
sequences["fibonacci"]![4] = 5
sequences["fibonacci"]!.append(8)
```

- The `@auto_closure` attribute has been renamed to `@autoclosure`.
- There is a new `dynamic` declaration modifier. When applied to a method, property, subscript, or initializer, it guarantees that references to the declaration are always dynamically dispatched and never inlined or devirtualized, and that the method binding can be reliably changed at runtime. The implementation currently relies on the Objective-C runtime, so `dynamic` can only be applied to `@objc-compatible` declarations for now. `@objc` now only makes a declaration visible to Objective-C; the compiler may now use vtable lookup or direct access to access (non-dynamic) `@objc` declarations.

  ```
  class Foo {
    // Always accessed by objc_msgSend
    dynamic var x: Int

    // Accessed by objc_msgSend from ObjC; may be accessed by vtable
    // or by static reference in Swift
    @objc var y: Int

    // Not exposed to ObjC (unless Foo inherits NSObject)
    var z: Int
  }
  ```

  `dynamic` enables KVO, proxying, and other advanced Cocoa features to work reliably with Swift declarations.
- Clang submodules can now be imported:

  ```
  import UIKit.UIGestureRecognizerSubclass
  ```
- The numeric optimization levels `-O[0-3]` have been removed in favor of the named levels `-Onone` and `-O`.
- The `-Ofast` optimization flag has been renamed to `-Ounchecked`. We will accept both names for now and remove `-Ofast` in a later build.
- An initializer that overrides a designated initializer from its superclass must be marked with the `override` keyword, so that all overrides in the language consistently require the use of `override`. For example:

```
class A {
  init() { }
}

class B : A {
  override init() { super.init() }
}
```

- Required initializers are now more prominent in several ways. First, a (non-final) class that conforms to a protocol that contains an initializer requirement must provide a required initializer to satisfy that requirement. This ensures that subclasses will also conform to the protocol, and will be most visible with classes that conform to `NSCoding`:

  ```
  class MyClass : NSObject, NSCoding {
    required init(coder aDecoder: NSCoder!) { /*... */ }
    func encodeWithCoder(aCoder: NSCoder!) { /* ... */ }
  }
  ```

  Second, because `required` places a significant requirement on all subclasses, the `required` keyword must be placed on overrides of a required initializer:

  ```
  class MySubClass : MyClass {
    var title: String = "Untitled"

    required init(coder aDecoder: NSCoder!) { /*... */ }
    override func encodeWithCoder(aCoder: NSCoder!) { /* ... */ }
  }
  ```

  Finally, required initializers can now be inherited like any other initializer:

  ```
  class MySimpleSubClass : MyClass { } // inherits the required init(coder:).
  ```

2014-07-21

- Access control has been implemented.

  - `public` declarations can be accessed from any module.
  - `internal` declarations (the default) can be accessed from within the current module.
  - `private` declarations can be accessed only from within the current file.

  There are still details to iron out here, but the model is in place. The general principle is that an entity cannot be defined in terms of another entity with less accessibility.

  Along with this, the generated header for a framework will only include public declarations. Generated headers for applications will include public and internal declarations.

- `CGFloat` is now a distinct floating-point type that wraps either a `Float` (on 32-bit architectures) or a `Double` (on 64-bit architectures). It provides all of the same comparison and arithmetic operations of Float and Double, and can be created using numeric literals.
- The immediate mode `swift -i` now works for writing `#!` scripts that take command line arguments. The `-i` option to the swift driver must now come at the end of the compiler arguments, directly before the input filename. Any arguments that come after `-i` and the input filename are treated as arguments to the interpreted file and forwarded to `Process.arguments`.
- Type inference for `for..in` loops has been improved to consider the sequence along with the element pattern. For example, this accepts the following loops that were previously rejected:

  ```
  for i: Int8 in 0..<10 { }
  for i: Float in 0.0...10.0 { }
  ```

- Introduced the new `BooleanLiteralConvertible` protocol, which allows user-defined types to support Boolean literals. `true` and `false` are now `Boolean` constants and keywords.
- The `@final`, `@lazy`, `@required` and `@optional` attributes are now considered to be declaration modifiers - they no longer require (or allow) an `@` sign.
- The `@prefix`, `@infix`, and `@postfix` attributes have been changed to declaration modifiers, so they are no longer spelled with an `@` sign. Operator declarations have been rearranged from `operator prefix +` to `prefix operator +` for consistency.

2014-07-03

- C function pointer types are now imported as `CFunctionPointer<T>`, where `T` is a Swift function type. `CFunctionPointer` and `COpaquePointer` can be explicitly constructed from one another, but they do not freely convert, nor is `CFunctionPointer` compatible with Swift closures.

  Example: `int (*)(void)` becomes `CFunctionPointer<(Int) -> Void>`.
- The interop model for pointers in C APIs has been simplified. Most code that calls C functions by passing arrays, UnsafePointers, or the addresses of variables with `&x` does not need to change. However, the `CConstPointer` and `CMutablePointer` bridging types have been removed, and functions and methods are now imported as and overridden by taking UnsafePointer and `ConstUnsafePointer` directly. `Void` pointers are now imported as `(Const)UnsafePointer<Void>`; `COpaquePointer` is only imported for opaque types now.

- `Array` types are now spelled with the brackets surrounding the element type. For example, an array of `Int` is written as:

  ```
  var array: [Int]
  ```

- `Dictionary` types can now be spelled with the syntax `[K : V]`, where `K` is the key type and `V` is the value type. For example:

  ```
  var dict: [String : Int] = ["Hello" : 1, "World" : 2]
  ```

  The type `[K : V]` is syntactic sugar for `Dictionary<K, V>`; nothing else has changed.

- The `@IBOutlet` attribute no longer implicitly (and invisibly) changes the type of the declaration it is attached to. It no longer implicitly makes variables be an implicitly unwrapped optional and no longer defaults them to weak.

- The `\x`, `\u` and `\U` escape sequences in string literals have been consolidated into a single and less error prone `\u{123456}` syntax.

2014-06-23

- The half-open range operator has been renamed from `..` to `..<` to reduce confusion. The `..<` operator is precedented in Groovy (among other languages) and makes it much more clear that it doesn't include the endpoint.

- Class objects such as `NSObject.self` can now be converted to `AnyObject` and used as object values.

- Objective-C protocol objects such as `NSCopying.self` can now be used as instances of the `Protocol` class, such as in APIs such as XPC.

- Arrays now have full value semantics: both assignment and initialization create a logically-distinct object

- The `sort` function and array method modify the target in-place. A new `sorted` function and array method are non-mutating, creating and returning a new collection.

2014-05-19

- `sort`, `map`, `filter`, and `reduce` methods on `Array`s accept trailing closures:

  ```
  let a = [5, 6, 1, 3, 9]
  a.sort{ $0 > $1 }
  println(a)                                 // [9, 6, 5, 3, 1]
  println(a.map{ $0 * 2 })                   // [18, 12, 10, 6, 2]
  println(a.map{ $0 * 2 }.filter{ $0 < 10})  // [6, 2]
  println(a.reduce(1000){ $0 + $1 })         // 1024 (no kidding)
  ```

- A lazy `map()` function in the standard library works on any `Sequence`. Example:

  ```
  class X {
    var value: Int

    init(_ value: Int) {
      self.value = value
      println("created X(\(value))")
    }
  }

  // logically, this sequence is X(0), X(1), X(2), ... X(50)
  let lazyXs = map(0..50){ X($0) }

  // Prints "created X(...)" 4 times
  for x in lazyXs {
    if x.value == 4 {
      break
    }
  }
  ```

- There's a similar lazy `filter()` function:

  ```
  // 0, 10, 20, 30, 40
  let tens = filter(0..50) { $0 % 10 == 0 }
  let tenX = map(tens){ X($0) }    // 5 lazy Xs
  let tenXarray = Array(tenX)      // Actually creates those Xs
  ```

- Weak pointers of classbound protocol type work now.

- `IBOutlets` now default to weak pointers with implicit optional type (`T!`).

- `NSArray*` parameters and result types of Objective-C APIs are now imported as `AnyObject[]!`, i.e., an implicitly unwrapped optional array storing `AnyObject` values. For example, `NSView`'s constraints property

  ```
  @property (readonly) NSArray *constraints;
  ```

  is now imported as

  ```
  var constraints: AnyObject[]!
  ```

  Note that one can implicitly convert between an `AnyObject[]` and an `NSArray` (in both directions), so (for example) one can still explicitly use `NSArray` if desired:

  ```
  var array: NSArray = view.constraints
  ```

  Swift arrays bridge to `NSArray` similarly to the way Swift strings bridge to `NSString`.

- `ObjCMutablePointer` has been renamed `AutoreleasingUnsafePointer`.
- `UnsafePointer` (and `AutoreleasingUnsafePointer`)'s `set()` and `get()` have been replaced with a property called `memory`.
  - Previously you would write:

    ```
    val = p.get()
    p.set(val)
    ```

  - Now you write:

    ```
    val = p.memory
    p.memory = val
    ```

- Removed shorthand `x as T!`; instead use `(x as T)!`
  - `x as T!` now means "x as implicitly unwrapped optional".
- Range operators `..` and `...` have been switched.
  - `1..3` now means 1,2
  - `1...3` now means 1,2,3
- The pound sign (`#`) is now used instead of the back-tick (‘) to mark an argument name as a keyword argument, e.g.,

  ```
  func moveTo(#x: Int, #y: Int) { ... }
  moveTo(x: 5, y: 7)
  ```

- Objective-C factory methods are now imported as initializers. For example, `NSColor`'s `+colorWithRed:green:blue:alpha` becomes

  ```
  init(red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)
  ```

  which allows an `NSColor` to be created as, e.g.,

  ```
  NSColor(red: 0.5, green: 0.25, blue: 0.25, alpha: 0.5)
  ```

  Factory methods are identified by their kind (class methods), name (starts with words that match the words that end the class name), and result type (`instancetype` or the class type).
- Objective-C properties of some `CF` type are no longer imported as `Unmanaged`.
- REPL mode now uses LLDB, for a greatly-expanded set of features. The colon prefix now treats the rest of the line as a command for LLDB, and entering a single colon will drop you into the debugging command prompt. Most importantly, crashes in the REPL will now drop you into debugging mode to see what went wrong.

  If you do have a need for the previous REPL, pass `-integrated-repl`.

- In a UIKit-based application, you can now eliminate your 'main.swift' file and instead apply the `@UIApplicationMain` attribute to your `UIApplicationDelegate` class. This will cause the `main` entry point to the application to be automatically generated as follows:

  ```
  UIApplicationMain(argc, argv, nil,
                    NSStringFromClass(YourApplicationDelegate.self))
  ```

  If you need nontrivial logic in your application entry point, you can still write out a `main.swift`. Note that `@UIApplicationMain` and `main.swift` are mutually exclusive.

2014-05-13

- weak pointers now work with implicitly unchecked optionals, enabling usecases where you don't want to `!` every use of a weak pointer. For example:

  ```
  weak var myView : NSView!
  ```

  of course, they still work with explicitly checked optionals like `NSView?`

- Dictionary subscripting now takes/returns an optional type. This allows querying a dictionary via subscripting to gracefully fail. It also enables the idiom of removing values from a dictionary using `dict[key] = nil`. As part of this, `deleteKey` is no longer available.

- Stored properties may now be marked with the `@lazy` attribute, which causes their initializer to be evaluated the first time the property is touched instead of when the enclosing type is initialized. For example:

  ```
  func myInitializer() -> Int { println("hello\n"); return 42 }
  class MyClass {
    @lazy var aProperty = myInitializer()
  }

  var c = MyClass()     // doesn't print hello
  var tmp = c.aProperty // prints hello on first access
  tmp = c.aProperty     // doesn't print on subsequent loads.

  c = MyClass()         // doesn't print hello
  c.aProperty = 57      // overwriting the value prevents it from ever running
  ```

  Because lazy properties inherently rely on mutation of the property, they cannot be `let`s. They are currently also limited to being members of structs and classes (they aren't allowed as local or global variables yet) and cannot be observed with `willSet`/`didSet` yet.

- Closures can now specify a capture list to indicate with what strength they want to capture a value, and to bind a particular field value if they want to.

Closure capture lists are square-bracket delimited and specified before the (optional) argument list in a closure. Each entry may be specified as `weak` or `unowned` to capture the value with a weak or unowned pointer, and may contain an explicit expression if desired. Some examples:

```
takeClosure { print(self.title) }                    // strong capture
takeClosure { [weak self] in print(self!.title) }    // weak capture
takeClosure { [unowned self] in print(self.title) }  // unowned capture
```

You can also bind arbitrary expression to named values in the capture list. The expression is evaluated when the closure is formed, and captured with the specified strength. For example:

```
// weak capture of "self.parent"
takeClosure { [weak tmp = self.parent] in print(tmp!.title) }
```

The full form of a closure can take a signature (an argument list and optionally a return type) if needed. To use either the capture list or the signature, you must specify the context sensitive `in` keyword. Here is a (weird because there is no need for `unowned`) example of a closure with both:

```
myNSSet.enumerateObjectsUsingBlock { [unowned self] (obj, stop) in
  self.considerWorkingWith(obj)
}
```

- The word `with` is now removed from the first keyword argument name if an initialized imported from Objective-C. For example, instead of building `UIColor` as:

  ```
  UIColor(withRed: r, green: g, blue: b, alpha: a)
  ```

  it will now be:

  ```
  UIColor(red: r, green: g, blue: b, alpha: a)
  ```

- `Dictionary` can be bridged to `NSDictionary` and vice versa:

  – `NSDictionary` has an implicit conversion to `Dictionary<NSObject, AnyObject>`. It bridges in O(1), without memory allocation.

  – `Dictionary<K, V>` has an implicit conversion to `NSDictionary`. `Dictionary<K, V>` bridges to `NSDictionary` iff both `K` and `V` are bridged. Otherwise, a runtime error is raised.

    Depending on `K` and `V` the operation can be `O(1)` without memory allocation, or `O(N)` with memory allocation.

- Single-quoted literals are no longer recognized. Use double-quoted literals and an explicit type annotation to define `Characters` and `UnicodeScalars`:

```
var ch: Character = "a"
var us: UnicodeScalar = "a"
```

2014-05-09

- The use of keyword arguments is now strictly enforced at the call site. For example, consider this method along with a call to it:

  ```
  class MyColor {
    func mixColorWithRed(red: Float, green: Float, blue: Float) { /* ... */ }
  }

  func mix(color: MyColor, r: Float, g: Float, b: Float) {
    color.mixColorWithRed(r, g, b)
  }
  ```

  The compiler will now complain about the missing `green:` and `blue:` labels, with a Fix-It to correct the code:

  ```
  color.swift:6:24: error: missing argument labels 'green:blue:' in call
    color.mixColorWithRed(r, g, b)
                         ^
                             green:  blue:
  ```

  The compiler handles missing, extraneous, and incorrectly-typed argument labels in the same manner. Recall that one can make a parameter a keyword argument with the back-tick or remove a keyword argument with the underscore.

  ```
  class MyColor {
    func mixColor(`red: Float, green: Float, blue: Float) { /* ... */ }
    func mixColorGuess(red: Float, _ green: Float, _ blue: Float) { /* ... */ }
  }

  func mix(color: MyColor, r: Float, g: Float, b: Float) {
    color.mixColor(red: r, green: g, blue: b) // okay: all keyword arguments
    color.mixColorGuess(r, g, b) // okay: no keyword arguments
  }
  ```

  Arguments cannot be re-ordered unless the corresponding parameters have default arguments. For example, given:

  ```
  func printNumber(`number: Int, radix: Int = 10, separator: String = ",") { }
  ```

  The following three calls are acceptable because only the arguments for defaulted parameters are re-ordered relative to each other:

  ```
  printNumber(number: 256, radix: 16, separator: "_")
  printNumber(number: 256, separator: "_")
  printNumber(number: 256, separator: ",", radix: 16)
  ```

However, this call:

```
printNumber(separator: ",", radix: 16, number: 256)
```

results in an error due to the re-ordering:

```
printnum.swift:7:40: error: argument 'number' must precede argument 'separator'
printNumber(separator: ",", radix: 16, number: 256)
            ~~~~~~~~~~~~~~             ^       ~~~
```

- `;` can no longer be used to demarcate an empty case in a switch statement, use `break` instead.

2014-05-07

- The compiler's ability to diagnose many common kinds of type check errors has improved. (`expression does not type-check` has been retired.)
- Ranges can be formed with floating point numbers, e.g. `0.0 .. 100.0`.
- Convenience initializers are now spelled as `convenience init` instead of with the `-> Self` syntax. For example:

  ```
  class Foo {
    init(x : Int) {}  // designated initializer

    convenience init() { self.init(42) } // convenience initializer
  }
  ```

  You still cannot declare designated initializers in extensions, only convenience initializers are allowed.

- Reference types using the CoreFoundation runtime are now imported as class types. This means that Swift will automatically manage the lifetime of a `CFStringRef` the same way that it manages the lifetime of an `NSString`.

  In many common cases, this will just work. Unfortunately, values are returned from `CF`-style APIs in a wide variety of ways, and unlike Objective-C methods, there simply isn't enough consistency for Swift to be able to safely apply the documented conventions universally. The framework teams have already audited many of the most important `CF`-style APIs, and those APIs should be imported without a hitch into Swift. For all the APIs which haven't yet been audited, we must import return types using the `Unmanaged` type. This type allows the programmer to control exactly how the object is passed.

  For example:

  ```
  // CFBundleGetAllBundles() returns an Unmanaged<CFArrayRef>.
  // From the documentation, we know that it returns a +0 value.
  let bundles = CFBundleGetAllBundles().takeUnretainedValue()
  ```

```
// CFRunLoopCopyAllModes() returns an Unmanaged<CFArrayRef>.
// From the documentation, we know that it returns a +1 value.
let modes = CFRunLoopCopyAllModes(CFRunLoopGetMain()).takeRetainedValue()
```

You can also use `Unmanaged` types to pass and return objects indirectly, as well as to generate unbalanced retains and releases if you really require them.

The API of the Unmanaged type is still in flux, and your feedback would be greatly appreciated.

2014-05-03

- The `@NSManaged` attribute can be applied to the properties of an `NSManagedObject` subclass to indicate that they should be handled by CoreData:

  ```
  class Employee : NSManagedObject {
    @NSManaged var name: String
    @NSManaged var department: Department
  }
  ```

- The `@weak` and `@unowned` attributes have become context sensitive keywords instead of attributes. To declare a `weak` or `unowned` pointer, use:

  ```
  weak var someOtherWindow : NSWindow?
  unowned var someWindow : NSWindow
  ```

  . . . with no `@` on the `weak`/`unowned`.

2014-04-30

- Swift now supports a `#elseif` form for build configurations, e.g.:

  ```
  #if os(OSX)
    typealias SKColor = NSColor
  #elseif os(iOS)
    typealias SKColor = UIColor
  #else
    typealias SKColor = Green
  #endif
  ```

- You can now use the `true` and `false` constants in build configurations, allowing you to emulate the C idioms of `#if 0` (but spelled `#if false`).

- `break` now breaks out of switch statements.

- It is no longer possible to specify `@mutating` as an attribute, you may only use it as a keyword, e.g.:

  ```
  struct Pair {
    var x, y : Int
  ```

```
  mutating func nuke() { x = 0; y = 0 }
}
```

The former `@!mutating` syntax used to mark setters as non-mutating is now spelled with the `nonmutating` keyword. Both mutating and nonmutating are context sensitive keywords.

- `NSLog` is now available from Swift code.
- The parser now correctly handles expressions like `var x = Int[]()` to create an empty array of integers. Previously you'd have to use syntax like `Array<Int>()` to get this. Now that this is all working, please prefer to use `Int[]` consistently instead of `Array<Int>`.
- `Character` is the new character literal type:

  ```
  var x = 'a' // Infers 'Character' type
  ```

  You can force inference of `UnicodeScalar` like this:

  ```
  var scalar: UnicodeScalar = 'a'
  ```

  `Character` type represents a Unicode extended grapheme cluster (to put it simply, a grapheme cluster is what users think of as a character: a base plus any combining marks, or other cases explained in Unicode Standard Annex #29).

2014-04-22

- Loops and switch statements can now carry labels, and you can `break`/`continue` to those labels. These use conventional C-style label syntax, and should be dedented relative to the code they are in. An example:

  ```
  func breakContinue(x : Int) -> Int {
  Outer:
    for a in 0..1000 {

    Switch:
      switch x {
      case 42: break Outer
      case 97: continue Outer
      case 102: break Switch
      case 13: continue // continue always works on loops.
      case 139: break   // break will break out of the switch (but see below)
      }
    }
  }
  ```

- We are changing the behavior of `break` to provide C-style semantics, to allow breaking out of a switch statement. Previously, break completely ignored switches so that it would break out of the nearest loop. In the example above, `case 139` would break out of the `Outer` loop, not the `Switch`.

In order to avoid breaking existing code, we're making this a compile time error instead of a silent behavior change. If you need a solution for the previous behavior, use labeled break.

This error will be removed in a week or two.

- Cocoa methods and properties that are annotated with the `NS_RETURNS_INNER_POINTER` attribute, including `-[NSData bytes]` and `-[{NS,UI}Color CGColor]`, are now safe to use and follow the same lifetime extension semantics as ARC.

2014-04-18

- Enabling/disabling of asserts

  ```
  assert(condition, msg)
  ```

  is enabled/disabled dependent on the optimization level. In debug mode at `-O0` asserts are enabled. At higher optimization levels asserts are disabled and no code is generated for them. However, asserts are always type checked even at higher optimization levels.

  Alternatively, assertions can be disabled/enabled by using the frontend flag `-assert-config Debug`, or `-assert-config Release`.

- Added optimization flag `-Ofast`. It disables all assertions (`assert`), and runtime overflow and type checks.

- The "selector-style" function and initializer declaration syntax is being phased out. For example, this:

  ```
  init withRed(red: CGFloat) green(CGFloat) blue(CGFloat) alpha(CGFloat)
  ```

  will now be written as:

  ```
  init(withRed red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)
  ```

  For each parameter, one can have both an argument API name (i.e., `withRed`, which comes first and is used at the call site) and an internal parameter name that follows it (i.e. `red`, which comes second and is used in the implementation). When the two names are the same, one can simply write the name once and it will be used for both roles (as with `green`, `blue`, and `alpha` above). The underscore (`_`) can be used to mean "no name", as when the following function/method:

  ```
  func murderInRoom(room:String) withWeapon(weapon: String)
  ```

  is translated to:

```
func murderInRoom(_ room: String, withWeapon weapon: String)
```

The compiler now complains when it sees the selector-style syntax and will provide Fix-Its to rewrite to the newer syntax.

Note that the final form of selector syntax is still being hammered out, but only having one declaration syntax, which will be very close to this, is a known.

- Stored properties can now be marked with the `@NSCopying` attribute, which causes their setter to be synthesized with a copy to `copyWithZone:`. This may only be used with types that conform to the `NSCopying` protocol, or option types thereof. For example:

  ```
  @NSCopying var myURL : NSURL
  ```

  This fills the same niche as the (`copy`) attribute on Objective-C properties.

2014-04-16

- Optional variables and properties are now default-initialized to `nil`:

  ```
  class MyClass {
    var cachedTitle: String?       // "= nil" is implied
  }
  ```

- `@IBOutlet` has been improved in a few ways:

  - `IBOutlets` can now be `@unchecked` optional.
  - An `IBOutlet` declared as non-optional, i.e.,

    ```
    @IBOutlet var button: NSButton
    ```

    will be treated as an `@unchecked` optional. This is considered to be the best practice way to write an outlet, unless you want to explicitly handle the null case - in which case, use `NSButton?` as the type. Either way, the `= nil` that was formerly required is now implicit.

- The precedence of `is` and `as` is now higher than comparisons, allowing the following sorts of things to be written without parens:

  ```
  if x is NSButton && y is NSButtonCell { ... }

  if 3/4 as Float == 6/8 as Float { ... }
  ```

- Objective-C blocks are now transparently bridged to Swift closures. You never have to write `@objc_block` when writing Objective-C-compatible methods anymore. Block parameters are now imported as unchecked optional closure types, allowing `nil` to be passed.

2014-04-09

- `Dictionary` changes:
  - `Elements` are now tuples, so you can write

    ```
    for (k, v) in d {
      // ...
    }
    ```

  - `keys` and `values` properties, which are `Collections` projecting the corresponding aspect of each element. `Dictionary` indices are usable with their `keys` and `values` properties, so:

    ```
    for i in indices(d) {
      let (k, v) = d[i]
      assert(k == d.keys[i])
      assert(v == d.values[i])
    }
    ```

- Semicolon can be used as a single no-op statement in otherwise empty cases in `switch` statements:

  ```
  switch x {
  case 1, 2, 3:
    print("x is 1, 2 or 3")
  default:
    ;
  }
  ```

- `override` is now a context sensitive keyword, instead of an attribute:

  ```
  class Base {
    var property: Int { return 0 }
    func instanceFunc() {}
    class func classFunc() {}
  }
  class Derived : Base {
    override var property: Int { return 1 }
    override func instanceFunc() {}
    override class func classFunc() {}
  }
  ```

2014-04-02

- Prefix splitting for imported enums has been revised again due to feedback:
  - If stripping off a prefix would leave an invalid identifier (like `10_4`), leave one more word in the result than would otherwise be there (`Behavior10_4`).

    - If all enumerators have a `k` prefix (for `constant`) and the enum doesn't, the `k` should not be considered when finding the common prefix.
    - If the enum name is a plural (like `NSSomethingOptions`) and the enumerator names use the singular form (`NSSomethingOptionMagic`), this is considered a matching prefix (but only if nothing follows the plural).

- Cocoa APIs that take pointers to plain C types as arguments now get imported as taking the new `CMutablePointer<T>` and `CConstPointer<T>` types instead of `UnsafePointer<T>`. These new types allow implicit conversions from Swift `inout` parameters and from Swift arrays:

  ```
  let rgb = CGColorSpaceCreateDeviceRGB()
  // CGColorRef CGColorCreate(CGColorSpaceRef, const CGFloat*);
  let white = CGColorCreate(rgb, [1.0, 1.0, 1.0])

  var s = 0.0, c = 0.0
  // void sincos(double, double*, double*);
  sincos(M_PI/2, &s, &c)
  ```

  Pointers to pointers to ObjC classes, such as `NSError**`, get imported as `ObjCMutablePointer<NSError?>`. This type doesn't work with arrays, but accepts inouts or `nil`:

  ```
  var error: NSError?
  let words = NSString.stringWithContentsOfFile("/usr/share/dict/words",
    encoding: .UTF8StringEncoding,
    error: &error)
  ```

  `Void` pointer parameters can be passed an array or inout of any type:

  ```
  // + (NSData*)dataWithBytes:(const void*)bytes length:(NSUInteger)length;
  let data = NSData.dataWithBytes([1.5, 2.25, 3.125],
                                  length: sizeof(Double.self) * 3)
  var fromData = [0.0, 0.0, 0.0]
  // - (void)getBytes:(void*)bytes length:(NSUInteger)length;
  data.getBytes(&fromData, length: sizeof(Double.self) * 3)
  ```

  Note that we don't know whether an API reads or writes the C pointer, so you need to explicitly initialize values (like `s` and `c` above) even if you know that the API overwrites them.

  This pointer bridging only applies to arguments, and only works with well-behaved C and ObjC APIs that don't keep the pointers they receive as arguments around or do other dirty pointer tricks. Nonstandard use of pointer arguments still requires `UnsafePointer`.

- Objective-C pointer types now get imported by default as the `@unchecked T?` optional type. Swift class types no longer implicitly include `nil`.

A value of `@unchecked T?` can be implicitly used as a value of `T`. Swift will implicitly cause a reliable failure if the value is `nil`, rather than introducing undefined behavior (as in Objective-C ivar accesses or everything in C/C++) or silently ignoring the operation (as in Objective-C message sends).

A value of `@unchecked T?` can also be implicitly used as a value of `T?`, allowing you explicitly handle the case of a `nil` value. For example, if you would like to just silently ignore a message send a la Objective-C, you can use the postfix `?` operator like so:

```
fieldsForKeys[kHeroFieldKey]?.setEditable(true)
```

This design allows you to isolate and handle `nil` values in Swift code without requiring excessive "bookkeeping" boilerplate to use values that you expect to be non-`nil`.

For now, we will continue to import C pointers as non-optional `UnsafePointer` and `C*Pointer` types; that will be evaluated separately.

We intend to provide attributes for Clang to allow APIs to opt in to importing specific parameters, return types, etc. as either the explicit optional type `T?` or the simple non-optional type `T`.

- The "separated" call syntax, i.e.,

  ```
  NSColor.colorWithRed(r) green(g) blue(b) alpha(a)
  UIColor.init withRed(r) green(g) blue(b) alpha(a)
  ```

  is being removed. The compiler will now produce an error and provide Fix-Its to rewrite calls to the "keyword-argument" syntax:

  ```
  NSColor.colorWithRed(r, green: g, blue: b, alpha: a)
  UIColor(withRed: r, green:g, blue:b, alpha: a)
  ```

- The `objc` attribute now optionally accepts a name, which can be used to provide the name for an entity as seen in Objective-C. For example:

  ```
  class MyType {
    var enabled: Bool {
      @objc(isEnabled) get {
        // ...
      }
    }
  }
  ```

  The `@objc` attribute can be used to name initializers, methods, getters, setters, classes, and protocols.

- Methods, properties and subscripts in classes can now be marked with the `@final` attribute. This attribute prevents overriding the declaration in any subclass, and provides better performance (since dynamic dispatch is avoided in many cases).

2014-03-26

- Attributes on declarations are no longer comma separated.

  Old syntax:

  ```
  @_silgen_name("foo"), @objc func bar() {}
  ```

  New syntax:

  ```
  @_silgen_name("foo") @objc
  ```

  The `,` was vestigial when the attribute syntax consisted of bracket lists.

- `switch` now always requires a statement after a `case` or `default`.

  Old syntax:

  ```
  switch x {
  case .A:
  case .B(1):
    println(".A or .B(1)")
  default:
    // Ignore it.
  }
  ```

  New syntax:

  ```
  switch x {
  case .A, .B(1):
    println(".A or .B(1)")
  default:
    () // Ignore it.
  }
  ```

  The following syntax can be used to introduce guard expressions for patterns inside the `case`:

  ```
  switch x {
  case .A where isFoo(),
       .B(1) where isBar():
    ...
  }
  ```

- Observing properties can now `@override` properties in a base class, so you can observe changes that happen to them.

  ```
  class MyAwesomeView : SomeBasicView {
   @override
   var enabled : Bool {
  ```

```
    didSet {
      println("Something changed")
    }
  }
  ...
}
```

Observing properties still invoke the base class getter/setter (or storage) when accessed.

- An `as` cast can now be forced using the postfix `!` operator without using parens:

```
class B {}
class D {}

let b: B = D()

// Before
let d1: D = (b as D)!
// After
let d2: D = b as D!
```

  Casts can also be chained without parens:

```
// Before
let b2: B = (((D() as B) as D)!) as B
// After
let b3: B = D() as B as D! as B
```

- `as` can now be used in `switch` cases to match the result of a checked cast:

```
func printHand(hand: Any) {
  switch hand {
  case 1 as Int:
    print("ace")
  case 11 as Int:
    print("jack")
  case 12 as Int:
    print("queen")
  case 13 as Int:
    print("king")
  case let numberCard as Int:
    print("\(numberCard)")
  case let (a, b) as (Int, Int) where a == b:
    print("two ")
    printHand(a)
    print("s")
  case let (a, b) as (Int, Int):
    printHand(a)
    print(" and a ")
    printHand(b)
  case let (a, b, c) as (Int, Int, Int) where a == b && b == c:
    print("three ")
    printHand(a)
    print("s")
  case let (a, b, c) as (Int, Int, Int):
    printHand(a)
    print(", ")
    printHand(b)
    print(", and a ")
    printHand(c)
  default:
    print("unknown hand")
  }
}
printHand(1, 1, 1) // prints "three aces"
printHand(12, 13) // prints "queen and a king"
```

- Enums and option sets imported from C/Objective-C still strip common prefixes, but the name of the enum itself is now taken into consideration as well. This keeps us from dropping important parts of a name that happen to be shared by all members.

  ```
  // NSFileManager.h
  typedef NS_OPTIONS(NSUInteger, NSDirectoryEnumerationOptions) {
      NSDirectoryEnumerationSkipsSubdirectoryDescendants = 1UL << 0,
      NSDirectoryEnumerationSkipsPackageDescendants      = 1UL << 1,
      NSDirectoryEnumerationSkipsHiddenFiles             = 1UL << 2
  } NS_ENUM_AVAILABLE(10_6, 4_0);
  ```

  ```
  // Swift
  let opts: NSDirectoryEnumerationOptions = .SkipsPackageDescendants
  ```

- `init` methods in Objective-C protocols are now imported as initializers. To conform to `NSCoding`, you will now need to provide

  ```
  init withCoder(aDecoder: NSCoder) { ... }
  ```

  rather than

  ```
  func initWithCoder(aDecoder: NSCoder) { ... }
  ```

2014-03-19

- When a class provides no initializers of its own but has default values for all of its stored properties, it will automatically inherit all of the initializers of its superclass. For example:

```
class Document {
  var title: String

  init() -> Self {
    self.init(withTitle: "Default title")
  }

  init withTitle(title: String) {
    self.title = title
  }
}

class VersionedDocument : Document {
  var version = 0

  // inherits 'init' and 'init withTitle:' from Document
}
```

When one does provide a designated initializer in a subclass, as in the following example:

```
class SecureDocument : Document {
  var key: CryptoKey

  init withKey(key: CryptoKey) -> Self {
    self.init(withKey: key, title: "Default title")
  }

  init withKey(key: CryptoKey) title(String) {
    self.key = key
    super.init(withTitle: title)
  }
}
```

the compiler emits Objective-C method stubs for all of the designated initializers of the parent class that will abort at runtime if called, and which indicate which initializer needs to be implemented. This provides memory safety for cases where an Objective-C initializer (such as `-[Document init]` in this example) appears to be inherited, but isn't actually implemented.

- `nil` may now be used as a Selector value. This allows calls to Cocoa methods that accept `nil` selectors.
- `[]` and `[:]` can now be used as the empty array and dictionary literal, respectively. Because these carry no information about their element types, they may only be used in a context that provides this information through type inference (e.g. when passing a function argument).

- Properties defined in classes are now dynamically dispatched and can be overridden with `@override`. Currently `@override` only works with computed properties overriding other computed properties, but this will be enhanced in coming weeks.

2014-03-12

- The `didSet` accessor of an observing property now gets passed in the old value, so you can easily implement an action for when a property changes value. For example:

  ```
  class MyAwesomeView : UIView {
    var enabled : Bool = false {
    didSet(oldValue):
      if oldValue != enabled {
        self.needsDisplay = true
      }
    }
    ...
  }
  ```

- The implicit argument name for set and willSet property specifiers has been renamed from `(value)` to `(newValue)`. For example:

  ```
  var i : Int {
    get {
      return 42
    }
    set {  // defaults to (newValue) instead of (value)
      print(newValue)
    }
  }
  ```

- The magic identifier `__FUNCTION__` can now be used to get the name of the current function as a string. Like `__FILE__` and `__LINE__`, if `__FUNCTION__` is used as a default argument, the function name of the caller is passed as the argument.

  ```
  func malkovich() {
    println(__FUNCTION__)
  }
  malkovich() // prints "malkovich"

  func nameCaller(name: String = __FUNCTION__) -> String {
    return name
  }

  func foo() {
  ```

```
  println(nameCaller()) // prints "foo"
}

func foo(x: Int) bar(y: Int) {
  println(nameCaller()) // prints "foo:bar:"
}
```

At top level, `__FUNCTION__` gives the module name:

```
println(nameCaller()) // prints your module name
```

- Selector-style methods can now be referenced without applying arguments using member syntax `foo.bar:bas:`, for instance, to test for the availability of an optional protocol method:

  ```
  func getFrameOfObjectValueForColumn(ds: NSTableViewDataSource,
                                      tableView: NSTableView,
                                      column: NSTableColumn,
                                      row: Int) -> AnyObject? {
    if let getObjectValue = ds.tableView:objectValueForTableColumn:row: {
      return getObjectValue(tableView, column, row)
    }
    return nil
  }
  ```

- The compiler now warns about cases where a variable is inferred to have `AnyObject`, `AnyClass`, or `()` type, since type inference can turn a simple mistake (e.g. failing to cast an `AnyObject` when you meant to) into something with ripple effects. Here is a simple example:

  ```
  t.swift:4:5: warning: variable 'fn' inferred to have type '()', which may be unexpected
  var fn = abort()
      ^
  t.swift:4:5: note: add an explicit type annotation to silence this warning
  var fn = abort()
      ^
        : ()
  ```

  If you actually did intend to declare a variable of one of these types, you can silence this warning by adding an explicit type (indicated by the Fixit). See **rdar://15263687 and rdar://16252090** for more rationale.

- `x.type` has been renamed to `x.dynamicType`, and you can use `type` as a regular identifier again.

2014-03-05

- C macros that expand to a single constant string are now imported as global constants. Normal string literals are imported as `CString`; `NSString` literals are imported as `String`.

- All values now have a `self` property, exactly equivalent to the value itself:

  ```
  let x = 0
  let x2 = x.self
  ```

  Types also have a `self` property that is the type object for that type:

  ```
  let theClass = NSObject.self
  let theObj = theClass()
  ```

  References to type names are now disallowed outside of a constructor call or member reference; to get a type object as a value, `T.self` is required. This prevents the mistake of intending to construct an instance of a class but forgetting the parens and ending up with the class object instead:

  ```
  let x = MyObject // oops, I meant MyObject()...
  return x.description() // ...and I accidentally called +description
                         //    instead of -description
  ```

- Initializers are now classified as **designated initializers**, which are responsible for initializing the current class object and chaining via `super.init`, and **convenience initializers**, which delegate to another initializer and can be inherited. For example:

  ```
  class A {
    var str: String

    init() -> Self { // convenience initializer
      self.init(withString: "hello")
    }

    init withString(str: String) { // designated initializer
      self.str = str
    }
  }
  ```

  When a subclass overrides all of its superclass's designated initializers, the convenience initializers are inherited:

  ```
  class B {
    init withString(str: String) { // designated initializer
      super.init(withString: str)
    }

    // inherits A.init()
  }
  ```

  Objective-C classes that provide `NS_DESIGNATED_INITIALIZER` annotations will have their init methods mapped to designated initializers or convenience initializers as appropriate; Objective-C classes without `NS_DESIGNATED_INITIALIZER` annotations have all of their `init` methods imported as designated initializers, which is safe (but can be verbose for subclasses). Note that the syntax and terminology is still somewhat in flux.

- Initializers can now be marked as `required` with an attribute, meaning that every subclass is required to provide that initializer either directly or by inheriting it from a superclass. To construct

```
class View {
  @required init withFrame(frame: CGRect) { ... }
}

func buildView(subclassObj: View.Type, frame: CGRect) -> View {
  return subclassObj(withFrame: frame)
}

class MyView : View {
  @required init withFrame(frame: CGRect) {
    super.init(withFrame: frame)
  }
}

class MyOtherView : View {
  // error: must override init withFrame(CGRect).
}
```

- Properties in Objective-C protocols are now correctly imported as properties. (Previously the getter and setter were imported as methods.)
- Simple enums with no payloads, including `NS_ENUM`s imported from Cocoa, now implicitly conform to the Equatable and Hashable protocols. This means they can be compared with the `==` and `!=` operators and can be used as `Dictionary` keys:

```
enum Flavor {
  case Lemon, Banana, Cherry
}

assert(Flavor.Lemon == .Lemon)
assert(Flavor.Banana != .Lemon)

struct Profile {
  var sweet, sour: Bool
}

let flavorProfiles: Dictionary<Flavor, Profile> = [
  .Lemon:  Profile(sweet: false, sour: true ),
  .Banana: Profile(sweet: true,  sour: false),
  .Cherry: Profile(sweet: true,  sour: true ),
]
assert(flavorProfiles[.Lemon].sour)
```

- `val` has been removed. Long live `let`!
- Values whose names clash with Swift keywords, such as Cocoa methods or properties named `class`, `protocol`, `type`, etc., can now be defined and accessed by wrapping reserved keywords in backticks to suppress their builtin meaning:

```
let `class` = 0
let `type` = 1
let `protocol` = 2
println(`class`)
println(`type`)
println(`protocol`)

func foo(Int) `class`(Int) {}
foo(0, `class`: 1)
```

2014-02-26

- The `override` attribute is now required when overriding a method, property, or subscript from a superclass. For example:

```
class A {
  func foo() { }
}

class B : A {
  @override func foo() { } // 'override' is required here
}
```

- We're renaming `val` back to `let`. The compiler accepts both for this week, next week it will just accept `let`. Please migrate your code this week, sorry for the back and forth on this.
- Swift now supports `#if`, `#else` and `#endif` blocks, along with target configuration expressions, to allow for conditional compilation within declaration and statement contexts.

  Target configurations represent certain static information about the compile-time build environment. They are implicit, hard-wired into the compiler, and can only be referenced within the conditional expression of an `#if` block.

  Target configurations are tested against their values via a pseudo-function invocation expression, taking a single argument expressed as an identifier. The argument represents certain static build-time information.

There are currently two supported target configurations: `os`, which can have the values `OSX` or `iOS` `arch`, which can have the values `i386`, `x86_64`, `arm` and `arm64`

Within the context of an `#if` block's conditional expression, a target configuration expression can evaluate to either `true` or `false`.

For example:

```
#if arch(x86_64)
  println("Building for x86_64")
#else
  println("Not building for x86_64")
#endif

class C {
#if os(OSX)
  func foo() {
    // OSX stuff goes here
  }
#else
  func foo() {
    // non-OSX stuff goes here
  }
#endif
}
```

The conditional expression of an `#if` block can be composed of one or more of the following expression types:

- A unary expression, using `!`
- A binary expression, using `&&` or `||`
- A parenthesized expression
- A target configuration expression

For example:

```
#if os(iOS) && !arch(I386)
...
#endif
```

Note that `#if`/`#else`/`#endif` blocks do not constitute a preprocessor, and must form valid and complete expressions or statements. Hence, the following produces a parser error:

```
class C {

#if os(iOS)
  func foo() {}
}
```

```
#else
  func bar() {}
  func baz() {}
}
#endif
```

Also note that "active" code will be parsed, typechecked and emitted, while "inactive" code will only be parsed. This is why code in an inactive `#if` or `#else` block will produce parser errors for malformed code. This allows the compiler to detect basic errors in inactive regions.

This is the first step to getting functionality parity with the important subset of the C preprocessor. Further refinements are planned for later.

- Swift now has both fully-closed ranges, which include their endpoint, and half-open ranges, which don't.

```
(swift) for x in 0...5 { print(x) } ; print('\n') // half-open range
01234
(swift) for x in 0..5 { print(x) } ; print('\n')  // fully-closed range
012345
```

- Property accessors have a new brace-based syntax, instead of using the former "label like" syntax. The new syntax is:

```
var computedProperty: Int {
  get {
    return _storage
  }
  set {
    _storage = value
  }
}

var implicitGet: Int {    // This form still works.
  return 42
}

var storedPropertyWithObservingAccessors: Int = 0 {
  willSet { ... }
  didSet { ... }
}
```

- Properties and subscripts now work in protocols, allowing you to do things like:

```
protocol Subscriptable {
  subscript(idx1: Int, idx2: Int) -> Int { get set }
  var prop: Int { get }
}
```

```
func foo(s: Subscriptable) {
  return s.prop + s[42, 19]
}
```

These can be used for generic algorithms now as well.

- The syntax for referring to the type of a type, `T.metatype`, has been changed to `T.Type`. The syntax for getting the type of a value, `typeof(x)`, has been changed to `x.type`.
- `DynamicSelf` is now called `Self`; the semantics are unchanged.
- `destructor` has been replaced with `deinit`, to emphasize that it is related to `init`. We will refer to these as `deinitializers`. We've also dropped the parentheses, i.e.:

```
class MyClass {
  deinit {
    // release any resources we might have acquired, etc.
  }
}
```

- Class methods defined within extensions of Objective-C classes can now refer to `self`, including using `instancetype` methods. As a result, `NSMutableString`, `NSMutableArray`, and `NSMutableDictionary` objects can now be created with their respective literals, i.e.,

```
var dict: NSMutableDictionary = ["a" : 1, "b" : 2]
```

2014-02-19

- The `Stream` protocol has been renamed back to `Generator,` which is precedented in other languages and causes less confusion with I/O streaming.
- The `type` keyword was split into two: `static` and `class`. One can define static functions and static properties in structs and enums like this:

```
struct S {
  static func foo() {}
  static var bar: Int = 0
}
enum E {
  static func foo() {}
}
```

`class` keyword allows one to define class properties and class methods in classes and protocols:

```
class C {
  class func foo() {}
  class var bar: Int = 0
}
protocol P {
  class func foo() {}
  class var bar: Int = 0
}
```

When using `class` and `static` in the extension, the choice of keyword depends on the type being extended:

```
extension S {
  static func baz() {}
}
extension C {
  class func baz() {}
}
```

- The `let` keyword is no longer recognized. Please move to `val`.
- The standard library has been renamed to `Swift` (instead of `swift`) to be more consistent with other modules on our platforms.
- `NSInteger` and other types that are layout-compatible with Swift standard library types are now imported directly as those standard library types.
- Optional types now support a convenience method named “cache” to cache the result of a closure. For example:

```
class Foo {
  var _lazyProperty: Int?
  var property: Int {
    return _lazyProperty.cache { computeLazyProperty() }
  }
}
```

2014-02-12

- We are experimenting with a new message send syntax. For example:

```
SKAction.colorizeWithColor(SKColor.whiteColor()) colorBlendFactor(1.0) duration(0.0)
```

When the message send is too long to fit on a single line, subsequent lines must be indented from the start of the statement or declaration. For example, this is a single message send:

```
SKAction.colorizeWithColor(SKColor.whiteColor())
         colorBlendFactor(1.0)
         duration(0.0)
```

while this is a message send to colorizeWithColor: followed by calls to `colorBlendFactor` and `duration` (on self or to a global function):

```
SKAction.colorizeWithColor(SKColor.whiteColor())
colorBlendFactor(1.0) // call to 'colorBlendFactor'
duration(0.0) // call to 'duration'
```

- We are renaming the `let` keyword to `val`. The `let` keyword didn't work out primarily because it is not a noun, so "defining a let" never sounded right. We chose `val` over `const` and other options because `var` and `val` have similar semantics (making syntactic similarity useful), because `const` has varied and sordid connotations in C that we don't want to bring over, and because we don't want to punish the "preferred" case with a longer keyword.

  For migration purposes, the compiler now accepts `let` and `val` as synonyms, `let` will be removed next week.

- Selector arguments in function arguments with only a type are now implicitly named after the selector chunk that contains them. For example, instead of:

  ```
  func addIntsWithFirst(first : Int) second(second : Int) -> Int {
    return first+second
  }
  ```

  you can now write:

  ```
  func addIntsWithFirst(first : Int) second(Int) -> Int {
    return first+second
  }
  ```

  if you want to explicitly want to ignore an argument, it is recommended that you continue to use the `_` to discard it, as in:

  ```
  func addIntsWithFirst(first : Int) second(_ : Int) -> Int {...}
  ```

- The `@inout` attribute in argument lists has been promoted to a context-sensitive keyword. Where before you might have written:

  ```
  func swap<T>(a : @inout T, b : @inout T) {
    (a, b) = (b, a)
  }
  ```

  You are now required to write:

  ```
  func swap<T>(inout a : T, inout b : T) {
    (a, b) = (b, a)
  }
  ```

  We made this change because `inout` is a fundamental part of the type system, which attributes are a poor match for. The inout keyword is also orthogonal to the `var` and `let` keywords (which may be specified in the same place), so it fits naturally there.

- The `@mutating` attribute (which can be used on functions in structs, enums, and protocols) has been promoted to a context-sensitive keyword. Mutating struct methods are now written as:

  ```
  struct SomeStruct {
    mutating func f() {}
  }
  ```

- Half-open ranges (those that don't include their endpoint) are now spelled with three `.`s instead of two, for consistency with Ruby.

  ```
  (swift) for x in 0...5 { print(x) } ; print('\n') // new syntax
  01234
  ```

  Next week, we'll introduce a fully-closed range which does include its endpoint. This will provide:

  ```
  (swift) for x in 0..5 { print(x) } ; print('\n')  // coming soon
  012345
  ```

  These changes are being released separately so that users have a chance to update their code before its semantics changes.

- Objective-C properties with custom getters/setters are now imported into Swift as properties. For example, the Objective-C property

  ```
  @property (getter=isEnabled) BOOL enabled;
  ```

  was previously imported as getter (`isEnabled`) and setter (`setEnabled`) methods. Now, it is imported as a property (`enabled`).

- `didSet`/`willSet` properties may now have an initial value specified:

  ```
  class MyAwesomeView : UIView {
    var enabled : Bool = false {       // Initial value.
    didSet: self.needsDisplay = true
    }
    ...
  }
  ```

  they can also be used as non-member properties now, e.g. as a global variable or a local variable in a function.

- Objective-C instancetype methods are now imported as methods that return Swift's `DynamicSelf` type. While `DynamicSelf` is not generally useful for defining methods in Swift, importing to it eliminates the need for casting with the numerous `instancetype` APIs, e.g.,

  ```
  let tileNode: SKSpriteNode = SKSpriteNode.spriteNodeWithTexture(tileAtlas.textureNamed(
  ```

  becomes

  ```
  let tileNode = SKSpriteNode.spriteNodeWithTexture(tileAtlas.textureNamed("tile\(tileNum
  ```

`DynamicSelf` will become more interesting in the coming weeks.

2014-02-05

- `if` and `while` statements can now conditionally bind variables. If the condition of an `if` or `while` statement is a `let` declaration, then the right-hand expression is evaluated as an `Optional` value, and control flow proceeds by considering the binding to be `true` if the `Optional` contains a value, or `false` if it is empty, and the variables are available in the true branch. This allows for elegant testing of dynamic types, methods, nullable pointers, and other Optional things:

  ```
  class B : NSObject {}
  class D : B {
    func foo() { println("we have a D") }
  }
  var b: B = D()
  if let d = b as D {
    d.foo()
  }
  var id: AnyObject = D()
  if let foo = id.foo {
    foo()
  }
  ```

- When referring to a member of an `AnyObject` (or `AnyClass`) object and using it directly (such as calling it, subscripting, or accessing a property on it), one no longer has to write the `?` or `!`. The run-time check will be performed implicitly. For example:

  ```
  func doSomethingOnViews(views: NSArray) {
    for view in views {
        view.updateLayer() // no '!' needed
    }
  }
  ```

  Note that one can still test whether the member is available at runtime using `?`, testing the optional result, or conditionally binding a variable to the resulting member.

- The `swift` command line tool can now create executables and libraries directly, just like Clang. Use `swift main.swift` to create an executable and `swift -emit-library -o foo.dylib foo.swift` to create a library.

- Object files emitted by Swift are not debuggable on their own, even if you compiled them with the `-g` option. This was already true if you had multiple files in your project. To produce a debuggable Swift binary from the command line, you must compile and link in a single step with `swift`, or pass object files AND swiftmodule files back into `swift` after compilation. (Or use Xcode.)

- `import` will no longer import other source files, only built modules.
- The current directory is no longer implicitly an import path. Use `-I .` if you have modules in your current directory.

2014-01-29

- Properties in structs and classes may now have `willSet:` and `didSet:` observing accessors defined on them:

  For example, where before you may have written something like this in a class:

  ```
  class MyAwesomeView : UIView {
    var _enabled : Bool  // storage
    var enabled : Bool { // computed property
    get:
      return _enabled
    set:
      _enabled = value
      self.needDisplay = true
    }
    ...
  }
  ```

  you can now simply write:

  ```
  class MyAwesomeView : UIView {
    var enabled : Bool {  // Has storage & observing methods
    didSet: self.needDisplay = true
    }
    ...
  }
  ```

  Similarly, if you want notification before the value is stored, you can use `willSet`, which gets the incoming value before it is stored:

  ```
  var x : Int {
  willSet(value):  // value is the default and may be elided, as with set:
    println("changing from \(x) to \(value)")
  didSet:
    println("we've got a value of \(x) now.\n")
  }
  ```

  The `willSet`/`didSet` observers are triggered on any store to the property, except stores from `init()`, destructors, or from within the observers themselves.

Overall, a property now may either be "stored" (the default), "computed" (have a `get:` and optionally a `set:` specifier), or an observed (`willSet`/`didSet`) property. It is not possible to have a custom getter or setter on an observed property, since they have storage.

Two known-missing bits are:

- **(rdar://problem/15920332) didSet/willSet variables need to allow initializers**
- **(rdar://problem/15922884) support non-member didset/willset properties**

Because of the first one, for now, you need to explicitly store an initial value to the property in your `init()` method.

- Objective-C properties with custom getter or setter names are (temporarily) not imported into Swift; the getter and setter will be imported individually as methods instead. Previously, they would appear as properties within the Objective-C class, but attempting to use the accessor with the customized name would result in a crash.

  The long-term fix is tracked as **(rdar://problem/15877160)**.

- Computed 'type' properties (that is, properties of types, rather than of values of the type) are now permitted on classes, on generic structs and enums, and in extensions. Stored 'type' properties in these contexts remain unimplemented.

  The implementation of stored 'type' properties is tracked as **(rdar://problem/15915785)** (for classes) and **(rdar://problem/15915867)** (for generic types).

- The following command-line flags have been deprecated in favor of new spellings. The old spellings will be removed in the following week's build:

Old Spelling	New Spelling
`-emit-llvm`	`-emit-ir`
`-triple`	`-target`
`-serialize-diagnostics`	`-serialize-diagnostics-path`

- Imported `NS_OPTIONS` types now have a default initializer which produces a value with no options set. They can also be initialized to the empty set with `nil`. These are equivalent:

```
var x = NSMatchingOptions()
var y: NSMatchingOptions = nil
```

2014-01-22

- The swift binary no longer has an SDK set by default. Instead, you must do one of the following:
  - pass an explicit `-sdk /path/to/sdk`
  - set `SDKROOT` in your environment
  - run `swift` through `xcrun`, which sets `SDKROOT` for you
- `let` declarations can now be used as struct/class properties. A `let` property is mutable within `init()`, and immutable everywhere else.

  ```
  class C {
    let x = 42
    let y : Int
    init(y : Int) {
      self.y = y   // ok, self.y is mutable in init()
    }

    func test() {
      y = 42       // error: 'y' isn't mutable
    }
  }
  ```

- The immutability model for structs and enums is complete, and arguments are immutable by default. This allows the compiler to reject mutations of temporary objects, catching common bugs. For example, this is rejected:

  ```
  func setTo4(a : Double[]) {
    a[10] = 4.0     // error: 'a' isn't mutable
  }
  ...
  setTo4(someArray)
  ```

  since `a` is semantically a copy of the array passed into the function. The proper fix in this case is to mark the argument is `@inout`, so the effect is visible in the caller:

  ```
  func setTo4(a : @inout Double[]) {
    a[10] = 4.0     // ok: 'a' is a mutable reference
  }
  ...
  setTo4(&someArray)
  ```

  Alternatively, if you really just want a local copy of the argument, you can mark it `var`. The effects aren't visible in the caller, but this can be convenient in some cases:

  ```
  func doStringStuff(var s : String) {
    s += "foo"
  ```

```
  print(s)
}
```

- Objective-C instance variables are no longer imported from headers written in Objective-C. Previously, they would appear as properties within the Objective-C class, but trying to access them would result in a crash. Additionally, their names can conflict with property names, which confuses the Swift compiler, and there are no patterns in our frameworks that expect you to access a parent or other class's instance variables directly. Use properties instead.
- The `NSObject` protocol is now imported under the name `NSObjectProtocol` (rather than `NSObjectProto`).

2014-01-15

- Improved deallocation of Swift classes that inherit from Objective-C classes: Swift destructors are implemented as `-dealloc` methods that automatically call the superclass's `-dealloc`. Stored properties are released right before the object is deallocated (using the same mechanism as ARC), allowing properties to be safely used in destructors.
- Subclasses of `NSManagedObject` are now required to provide initial values for each of their stored properties. This permits initialization of these stored properties directly after +alloc to provide memory safety with CoreData's dynamic subclassing scheme.
- `let` declarations are continuing to make slow progress. Curried and selector-style arguments are now immutable by default, and `let` declarations now get proper debug information.

2014-01-08

- The `static` keyword changed to `type`. One can now define "type functions" and "type variables" which are functions and variables defined on a type (rather than on an instance of the type), e.g.,

  ```
  class X {
    type func factory() -> X { ... }

    type var version: Int
  }
  ```

  The use of `static` was actively misleading, since type methods on classes are dynamically dispatched (the same as Objective-C `+` methods).

  Note that `type` is a context-sensitive keyword; it can still be used as an identifier.

- Strings have a new native UTF-16 representation that can be converted back and forth to `NSString` at minimal cost. String literals are emitted as UTF-16 for string types that support it (including Swift's `String`).
- Initializers can now delegate to other initializers within the same class by calling `self.init`. For example:

  ```
  class A { }

  class B : A {
    var title: String

    init() {
      // note: cannot access self before delegating
      self.init(withTitle: "My Title")
    }

    init withTitle(title: String) {
      self.title = title
      super.init()
    }
  }
  ```

- Objective-C protocols no longer have the `Proto` suffix unless there is a collision with a class name. For example, `UITableViewDelegate` is now imported as `UITableViewDelegate` rather than `UITableViewDelegateProto`. Where there is a conflict with a class, the protocol will be suffixed with `Proto`, as in `NSObject` (the class) and `NSObjectProto` (the protocol).

2014-01-01

- Happy New Year
- Division and remainder arithmetic now trap on overflow. Like with the other operators, one can use the "masking" alternatives to get non-trapping behavior. The behavior of the non-trapping masking operators is defined:

  ```
  x &/ 0 == 0
  x &% 0 == 0
  SIGNED_MIN_FOR_TYPE &/ -1 == -1 // i.e. Int8: -0x80 / -1 == -0x80
  SIGNED_MIN_FOR_TYPE &% -1 == 0
  ```

- Protocol conformance checking for `@mutating` methods is now implemented: an `@mutating` struct method only fulfills a protocol requirement if the protocol method was itself marked `@mutating`:

  ```
  protocol P {
    func nonmutating()
    @mutating
  ```

```
  func mutating()
}

struct S : P {
  // Error, @mutating method cannot implement non-@mutating requirement.
  @mutating
  func nonmutating() {}

  // Ok, mutating allowed, but not required.
  func mutating() {}
}
```

As before, class methods never need to be marked `@mutating` (and indeed, they aren't allowed to be marked as such).

2013-12-25

- Merry Christmas
- The setters of properties on value types (structs/enums) are now `@mutating` by default. To mark a setter non-mutating, use the `@!mutating` attribute.
- Compiler inserts calls to `super.init()` into the class initializers that do not call any initializers explicitly.
- A `map` method with the semantics of Haskell's `fmap` was added to `Array<T>`. Map applies a function `f: T->U` to the values stored in the array and returns an `Array<U>`. So,

  ```
  (swift) func names(x: Int[]) -> String[] {
            return x.map { "<" + String($0) + ">" }
          }
  (swift) names(Array<Int>())
  // r0 : String[] = []
  (swift) names([3, 5, 7, 9])
  // r1 : String[] = ["<3>", "<5>", "<7>", "<9>"]
  ```

2013-12-18

- Global variables and static properties are now lazily initialized on first use. Where you would use `dispatch_once` to lazily initialize a singleton object in Objective-C, you can simply declare a global variable with an initializer in Swift. Like `dispatch_once`, this lazy initialization is thread safe.

  Unlike C++ global variable constructors, Swift global variables and static properties now never emit static constructors (and thereby don't raise build warnings). Also unlike C++, lazy initialization naturally follows dependency order, so global variable initializers that cross module boundaries don't have undefined behavior or fragile link order dependencies.

- Swift has the start of an immutability model for value types. As part of this, you can now declare immutable value bindings with a new `let` declaration, which is semantically similar to defining a get-only property:

  ```
  let x = foo()
  print(x)        // ok
  x = bar()       // error: cannot modify an immutable value
  swap(&x, &y)    // error: cannot pass an immutable value as @inout parameter
  x.clear()       // error: cannot call mutating method on immutable value
  getX().clear()  // error: cannot mutate a temporary
  ```

  In the case of bindings of class type, the bound object itself is still mutable, but you cannot change the binding.

  ```
  let r = Rocket()
  r.blastOff()    // Ok, your rocket is mutable.
  r = Rocket()    // error: cannot modify an immutable binding.
  ```

  In addition to the `let` declaration itself, `self` on classes, and a few other minor things have switched to immutable bindings.

  A pivotal part of this is that methods of value types (structs and enums) need to indicate whether they can mutate self - mutating methods need to be disallowed on let values (and get-only property results, temporaries, etc) but non-mutating methods need to be allowed. The default for a method is that it does not mutate `self`, though you can opt into mutating behavior with a new `@mutating` attribute:

  ```
  struct MyWeirdCounter {
    var count : Int

    func empty() -> Bool { return count == 0 }

    @mutating
    func reset() {
      count = 0
    }
    ...
  }

  let x = MyWeirdCounter()
  x.empty()   // ok
  x.reset()   // error, cannot mutate immutable 'let' value
  ```

  One missing piece is that the compiler does not yet reject mutations of self in a method that isn't marked `@mutating`. That will be coming soon. Related to methods are properties. Getters and setters can be marked mutating as well:

  ```
  extension MyWeirdCounter {
  ```

```
  var myproperty : Int {
  get:
    return 42

  @mutating
  set:
    count = value*2
  }
}
```

The intention is for setters to default to mutating, but this has not been implemented yet. There is more to come here.

- A `map` method with the semantics of Haskell's `fmap` was added to `Optional<T>`. Map applies a function `f: T->U` to any value stored in an `Optional<T>`, and returns an `Optional<U>`. So,

```
(swift) func nameOf(x: Int?) -> String? {
          return x.map { "<" + String($0) + ">" }
        }
(swift)
(swift) var no = nameOf(.None) // Empty optional in...
// no : String? = <unprintable value>
(swift) no ? "yes" : "no"      // ...empty optional out
// r0 : String = "no"
(swift)
(swift) nameOf(.Some(42))      // Non-empty in
// r1 : String? = <unprintable value>
(swift) nameOf(.Some(42))!     // Non-empty out
// r2 : String = "<42>"
```

- Cocoa types declared with the `NS_OPTIONS` macro are now available in Swift. Like `NS_ENUM` types, their values are automatically shortened based on the common prefix of the value names in Objective-C, and the name can be elided when type context provides it. They can be used in `if` statements using the `&`, `|`, `^`, and `~` operators as in C:

```
var options: NSJSONWritingOptions = .PrettyPrinted
if options & .PrettyPrinted {
  println("pretty-printing enabled")
}
```

We haven't yet designed a convenient way to author `NS_OPTIONS`-like types in Swift.

2013-12-11

- Objective-C `id` is now imported as `AnyObject` (formerly known as `DynamicLookup`), Objective-C `Class` is imported as `AnyClass`.

- The casting syntax `x as T` now permits both implicit conversions (in which case it produces a value of type `T`) and for runtime-checked casts (in which case it produces a value of type `T?` that will be `.Some(casted x)` on success and `.None` on failure). An example:

  ```
  func f(x: AnyObject, y: NSControl) {
    var view = y as NSView                    // has type 'NSView'
    var maybeView = x as NSView               // has type NSView?
  }
  ```

- The precedence levels of binary operators has been redefined, with a much simpler model than C's. This is with a goal to define away classes of bugs such as those caught by Clang's `-Wparentheses` warnings, and to make it actually possible for normal humans to reason about the precedence relationships without having to look them up.

  We ended up with 6 levels, from tightest binding to loosest: `exponentiative: <<, >>   multiplicative: *, /, %, &   additive: +, -, |, ^   comparative: ==, !=, <, <=, >=, >   conjunctive: &&   disjunctive: ||`

- The `Enumerable` protocol has been renamed `Sequence`.

- The `Char` type has been renamed `UnicodeScalar`. The preferred unit of string fragments for users is called `Character`.

- Initialization semantics for classes, structs and enums init methods are now properly diagnosed by the compiler. Instance variables now follow the same initialization rules as local variables: they must be defined before use. The initialization model requires that all properties with storage in the current class be initialized before `super.init` is called (or, in a root class, before any method is called on `self,` and before the final return).

  For example, this will yield an error:

  ```
  class SomeClass : SomeBase {
    var x : Int

    init() {
      // error: property 'self.x' not initialized at super.init call
      super.init()
    }
  }
  ```

  A simple fix for this is to change the property definition to `var x = 0`, or to explicitly assign to it before calling `super.init()`.

- Relatedly, the compiler now diagnoses incorrect calls to `super.init()`. It validates that any path through an initializer calls `super.init()` exactly once, that all ivars are defined before the call to super.init, and that any uses which require the entire object to be initialized come after the `super.init` call.

- Type checker performance has improved considerably (but we still have much work to do here).

2013-12-04

- The "slice" versus "array" subtlety is now dead. `Slice<T>` has been folded into `Array<T>` and `T[]` is just sugar for `Array<T>`.

2013-11-20

- Unreachable code warning has been added:

  ```
  var y: Int = 1
  if y == 1 { // note: condition always evaluates to true
    return y
  }
  return 1 // warning: will never be executed
  ```

- Overflows on integer type conversions are now detected at runtime and, when dealing with constants, at compile time:

  ```
  var i: Int = -129
  var i8 = Int8(i)
  // error: integer overflows when converted from 'Int' to 'Int8'

  var si = Int8(-1)
  var ui = UInt8(si)
  // error: negative integer cannot be converted to unsigned type 'UInt8'
  ```

- `def` keyword was changed back to `func`.

2013-11-13

- Objective-C-compatible protocols can now contain optional requirements, indicated by the `@optional` attribute:

  ```
  @class_protocol @objc protocol NSWobbling {
    @optional def wobble()
  }
  ```

  A class that conforms to the `NSWobbling` protocol above can (but does not have to) implement `wobble`. When referring to the `wobble` method for a value of type `NSWobbling` (or a value of generic type that is bounded by `NSWobbling`), the result is an optional value indicating whether the underlying object actually responds to the given selector, using the same mechanism as messaging `id`. One can use `!` to assume that the method is always there, `?` to chain the optional, or conditional branches to handle each case distinctly:

```
def tryToWobble(w : NSWobbling) {
  w.wobble()   // error: cannot call a value of optional type
  w.wobble!()  // okay: calls -wobble, but fails at runtime if not there
  w.wobble?()  // okay: calls -wobble only if it's there, otherwise no-op
  if w.wobble {
    // okay: we know -wobble is there
  } else {
    // okay: we know -wobble is not there
  }
}
```

- Enums from Cocoa that are declared with the `NS_ENUM` macro are now imported into Swift as Swift enums. Like all Swift enums, the constants of the Cocoa enum are scoped as members of the enum type, so the importer strips off the common prefix of all of the constant names in the enum when forming the Swift interface. For example, this Objective-C declaration:

```
typedef NS_ENUM(NSInteger, NSComparisonResult) {
  NSOrderedAscending,
  NSOrderedSame,
  NSOrderedDescending,
};
```

shows up in Swift as:

```
enum NSComparisonResult : Int {
  case Ascending, Same, Descending
}
```

The `enum` cases can then take advantage of type inference from context. In Objective-C, you would write:

```
NSNumber *foo = [NSNumber numberWithInt: 1];
NSNumber *bar = [NSNumber numberWithInt: 2];

switch ([foo compare: bar]) {
case NSOrderedAscending:
  NSLog(@"ascending\n");
  break;
case NSOrderedSame:
  NSLog(@"same\n");
  break;
case NSOrderedDescending:
  NSLog(@"descending\n");
  break;
}
```

In Swift, this becomes:

```
var foo: NSNumber = 1
var bar: NSNumber = 2

switch foo.compare(bar) {
case .Ascending:
  println("ascending")
case .Same:
  println("same")
case .Descending:
  println("descending")
}
```

- Work has begun on implementing static properties. Currently they are supported for nongeneric structs and enums.

```
struct Foo {
  static var foo: Int = 2
}
enum Bar {
  static var bar: Int = 3
}
println(Foo.foo)
println(Bar.bar)
```

2013-11-06

- `func` keyword was changed to `def`.
- Implicit conversions are now allowed from an optional type `T?` to another optional type `U?` if `T` is implicitly convertible to `U`. For example, optional subclasses convert to their optional base classes:

```
class Base {}
class Derived : Base {}

var d: Derived? = Derived()
var b: Base? = d
```

2013-10-30

- Type inference for variables has been improved, allowing any variable to have its type inferred from its initializer, including global and instance variables:

```
class MyClass {
  var size = 0 // inferred to Int
}
```

```
var name = "Swift"
```

Additionally, the arguments of a generic type can also be inferred from the initializer:

```
// infers Dictionary<String, Int>
var dict: Dictionary = ["Hello": 1, "World": 2]
```

2013-10-23

- Missing return statement from a non-`Void` function is diagnosed as an error.
- `Vector<T>` has been replaced with `Array<T>`. This is a complete rewrite to use value-semantics and copy-on-write behavior. The former means that you never need to defensively copy again (or remember to attribute a property as "copy") and the latter yields better performance than defensive copying. `Dictionary<T>` is next.
- `switch` can now pattern-match into structs and classes, using the syntax `case Type(property1: pattern1, property2: pattern2, ...):`.

  ```
  struct Point { var x, y: Double }
  struct Size { var w, h: Double }
  struct Rect { var origin: Point; var size: Size }

  var square = Rect(Point(0, 0), Size(10, 10))

  switch square {
  case Rect(size: Size(w: var w, h: var h)) where w == h:
    println("square")
  case Rect(size: Size(w: var w, h: var h)) where w > h:
    println("long rectangle")
  default:
    println("tall rectangle")
  }
  ```

  Currently only stored properties ("ivars" in ObjC terminology) are supported by the implementation.
- Array and dictionary literals allow an optional trailing comma:

  ```
  var a = [1, 2,]
  var d = ["a": 1, "b": 2,]
  ```

2013-10-16

- Unlike in Objective-C, objects of type `id` in Swift do not implicitly convert to any class type. For example, the following code is ill-formed:

```
func getContentViewBounds(window : NSWindow) -> NSRect {
  var view : NSView = window.contentView() // error: 'id' doesn't implicitly convert to
 return view.bounds()
}
```

because `contentView()` returns an `id`. One can now use the postfix `!` operator to allow an object of type `id` to convert to any class type, e.g.,

```
func getContentViewBounds(window : NSWindow) -> NSRect {
  var view : NSView = window.contentView()! // ok: checked conversion to NSView
 return view.bounds()
}
```

The conversion is checked at run-time, and the program will fail if the object is not an NSView. This is shorthand for

```
var view : NSView = (window.contentView() as NSView)!
```

which checks whether the content view is an `NSView` (via the `as NSView`). That operation returns an optional `NSView` (written `NSView?`) and the `!` operation assumes that the cast succeeded, i.e., that the optional has a value in it.

- The unconditional checked cast syntax `x as! T` has been removed. Many cases where conversion from `id` is necessary can now be handled by postfix `!` (see above). Fully general unconditional casts can still be expressed using `as` and postfix `!` together, `(x as T)!`.
- The old "square bracket" attribute syntax has been removed.
- Overflows on construction of integer and floating point values from integer literals that are too large to fit the type are now reported by the compiler. Here are some examples:

```
var x = Int8(-129)
// error: integer literal overflows when stored into 'Int8'

var y: Int = 0xFFFF_FFFF_FFFF_FFFF_F
// error: integer literal overflows when stored into 'Int'
```

  Overflows in constant integer expressions are also reported by the compiler.

```
var x: Int8 = 125
var y: Int8 = x + 125
// error: arithmetic operation '125 + 125' (on type 'Int8') results in
//        an overflow
```

- Division by zero in constant expressions is now detected by the compiler:

```
var z: Int = 0
var x = 5 / z  // error: division by zero
```

- Generic structs with type parameters as field types are now fully supported.

```
struct Pair<T, U> {
  var first: T
  var second: U
}
```

2013-10-09

- Autorelease pools can now be created using the `autoreleasepool` function.

  ```
  autoreleasepool {
    // code
  }
  ```

  Note that the wrapped code is a closure, so constructs like `break` and `continue` and `return` do not behave as they would inside an Objective-C `@autoreleasepool` statement.

- Enums can now declare a "raw type", and cases can declare "raw values", similar to the integer underlying type of C enums:

  ```
  // Declare the underlying type as in Objective-C or C++11, with
  // ': Type'
  enum AreaCode : Int {
    // Assign explicit values to cases with '='
    case SanFrancisco = 415
    case EastBay = 510
    case Peninsula = 650
    case SanJose = 408
    // Values are also assignable by implicit auto-increment
    case Galveston // = 409
    case Baltimore // = 410
  }
  ```

  This introduces `fromRaw` and `toRaw` methods on the enum to perform conversions from and to the raw type:

  ```
  /* As if declared:
      extension AreaCode {
        // Take a raw value, and produce the corresponding enum value,
        // or None if there is no corresponding enum value
        static func fromRaw(raw:Int) -> AreaCode?

        // Return the corresponding raw value for 'self'
        func toRaw() -> Int
      }
   */

  AreaCode.fromRaw(415) // => .Some(.SanFrancisco)
  ```

```
AreaCode.fromRaw(111) // => .None
AreaCode.SanJose.toRaw() // => 408
```

Raw types are not limited to integer types–they can additionally be character, floating-point, or string values:

```
enum State : String {
  case CA = "California"
  case OR = "Oregon"
  case WA = "Washington"
}

enum SquareRootOfInteger : Float {
  case One = 1.0
  case Two = 1.414
  case Three = 1.732
  case Four = 2.0
}
```

Raw types are currently limited to simple C-like enums with no payload cases. The raw values are currently restricted to simple literal values; expressions such as `1 + 1` or references to other enum cases are not yet supported. Raw values are also currently required to be unique for each case in an enum.

Enums with raw types implicitly conform to the `RawRepresentable` protocol, which exposes the fromRaw and toRaw methods to generics:

```
protocol RawRepresentable {
  typealias RawType
  static func fromRaw(raw: RawType) -> Self?
  func toRaw() -> RawType
}
```

- Attribute syntax has been redesigned (see **(rdar://10700853)** and **(rdar://14462729)**) so that attributes now precede the declaration and use the `@` character to signify them. Where before you might have written:

  ```
  func [someattribute=42] foo(a : Int) {}
  ```

  you now write:

  ```
  @someattribute=42
  func foo(a : Int) {}
  ```

  This flows a lot better (attributes don't push the name for declarations away), and means that square brackets are only used for array types, collection literals, and subscripting operations.

- The `for` loop now uses the Generator protocol instead of the `Enumerator` protocol to iterate a sequence. This protocol looks like this:

```
protocol Generator {
  typealias Element
  func next() -> Element?
}
```

The single method `next()` advances the generator and returns an Optional, which is either `.Some(value)`, wrapping the next value out of the underlying sequence, or `.None` to signal that there are no more elements. This is an improvement over the previous Enumerator protocol because it eliminates the separate `isEmpty()` query and better reflects the semantics of ephemeral sequences like un-buffered input streams.

2013-10-02

- The `[byref]` attribute has been renamed to `[inout]`. When applied to a logical property, the getter is invoked before a call and the setter is applied to write back the result. `inout` conveys this better and aligns with existing Objective-C practice better.
- `[inout]` arguments can now be captured into closures. The semantics of a inout capture are that the captured variable is an independent local variable of the callee, and the inout is updated to contain the value of that local variable at function exit.

  In the common case, most closure arguments do not outlive the duration of their callee, and the observable behavior is unchanged. However, if the captured variable outlives the function, you can observe this. For example, this code:

  ```
  func foo(x : [inout] Int) -> () -> Int {
    func bar() -> Int {
      x += 1
      return x
    }
    // Call 'bar' once while the inout is active.
    bar()
    return bar
  }

  var x = 219
  var f = foo(&x)
  // x is updated to the value of foo's local x at function exit.
  println("global x = \(x)")
  // These calls only update the captured local 'x', which is now independent
  // of the inout parameter.
  println("local x = \(f())")
  println("local x = \(f())")
  println("local x = \(f())")
  ```

```
println("global x = \(x)")
```

will print:

```
global x = 220
local x = 221
local x = 222
local x = 223
global x = 220
```

In no case will you end up with a dangling pointer or other unsafe construct.

- `x as T` now performs a checked cast to `T?`, producing `.Some(t)` if the cast succeeds, or `.None` if the cast fails.
- The ternary expression (`x ? y : z`) now requires whitespace between the first expression and the question mark. This permits `?` to be used as a postfix operator.
- A significant new piece of syntactic sugar has been added to ease working with optional values. The `?` postfix operator is analogous to `!`, but instead of asserting on None, it causes all the following postfix operators to get skipped and return `None`.

  In a sense, this generalizes (and makes explicit) the Objective-C behavior where message sends to `nil` silently produce the zero value of the result.

  For example, this code

  ```
  object?.parent.notifyChildEvent?(object!, .didExplode)
  ```

  first checks whether `object` has a value; if so, it drills to its parent and checks whether that object implements the `notifyChildEvent` method; if so, it calls that method. (Note that we do not yet have generalized optional methods.)

  This code:

  ```
  var titleLength = object?.title.length
  ```

  checks whether `object` has a value and, if so, asks for the length of its title. `titleLength` will have type `Int?`, and if `object` was missing, the variable will be initialized to None.

- Objects with type `id` can now be used as the receiver of property accesses and subscript operations to get (but not set) values. The result is of optional type. For example, for a variable `obj` of type `id`, the expression

  ```
  obj[0]
  ```

  will produce a value of type `id`, which will either contain the result of the message send objectAtIndexedSubscript(0) (wrapped in an optional type) or, if the object does not respond to `objectAtIndexedSubscript:`, an empty optional. The same approach applies to property accesses.

- `_` can now be used not only in `var` bindings, but in assignments as well, to ignore elements of a tuple assignment, or to explicitly ignore values.

```
var a = (1, 2.0, 3)
var x = 0, y = 0
_ = a           // explicitly load and discard 'a'
(x, _, y) = a   // assign a.0 to x and a.2 to y
```

2013-09-24

- The `union` keyword has been replaced with `enum`. Unions and enums are semantically identical in swift (the former just has data associated with its discriminators) and `enum` is the vastly more common case. For more rationale, please see docs/proposals/Enums.rst

- The Optional type `T?` is now represented as an `enum`:

```
enum Optional<T> {
  case None
  case Some(T)
}
```

This means that, in addition to the existing Optional APIs, it can be pattern-matched with switch:

```
var x : X?, y : Y?
switch (x, y) {
// Both are present
case (.Some(var a), .Some(var b)):
  println("both")

// One is present
case (.Some, .None):
case (.None, .Some):
  println("one")

// Neither is present
case (.None, .None):
  println("neither")
}
```

- Enums now allow multiple cases to be declared in a comma-separated list in a single `case` declaration:

```
enum Color {
  case Red, Green, Blue
}
```

- The Objective-C `id` and `Class` types now support referring to methods declared in any class or protocol without a downcast. For example, given a variable `sender` of type `id`, one can refer to `-isEqual: with:`

  ```
  sender.isEqual
  ```

  The actual object may or may not respond to `-isEqual`, so this expression returns result of optional type whose value is determined via a compiler-generated `-respondsToSelector` send. When it succeeds, the optional contains the method; when it fails, the optional is empty.

  To safely test the optional, one can use, e.g.,

  ```
  var senderIsEqual = sender.isEqual
  if senderIsEqual {
    // this will never trigger an "unrecognized selector" failure
    var equal = senderIsEqual!(other)
  } else {
    // sender does not respond to -isEqual:
  }
  ```

  When you *know* that the method is there, you can use postfix `!` to force unwrapping of the optional, e.g.,

  ```
  sender.isEqual!(other)
  ```

  This will fail at runtime if in fact sender does not respond to `-isEqual:`. We have some additional syntactic optimizations planned for testing an optional value and handling both the success and failure cases concisely. Watch this space.

- Weak references now always have optional type. If a weak variable has an explicit type, it must be an optional type:

  ```
  var [weak] x : NSObject?
  ```

  If the variable is not explicitly typed, its type will still be inferred to be an optional type.

- There is now an implicit conversion from `T` to `T?`.

2013-09-17

- Constructor syntax has been improved to align better with Objective-C's `init` methods. The `constructor` keyword has been replaced with `init`, and the selector style of declaration used for func declarations is now supported. For example:

  ```
  class Y : NSObject {
    init withInt(i : Int) string(s : String) {
      super.init() // call superclass initializer
  ```

```
  }
}
```

One can use this constructor to create a `Y` object with, e.g.,

```
Y(withInt:17, string:"Hello")
```

Additionally, the rules regarding the selector corresponding to such a declaration have been revised. The selector for the above initializer is `initWithInt:string:`; the specific rules are described in the documentation.

Finally, Swift initializers now introduce Objective-C entry points, so a declaration such as:

```
class X : NSObject {
  init() {
    super.init()
  }
}
```

Overrides `NSObject`'s `-init` method (which it calls first) as well as introducing the 'allocating' entry point so that one can create a new `X` instance with the syntax `X()`.

- Variables in top-level code (i.e. scripts, but not global variables in libraries) that lack an initializer now work just like local variables: they must be explicitly assigned-to sometime before any use, instead of being default constructed. Instance variables are still on the TODO list.

- Generic unions with a single payload case and any number of empty cases are now implemented, for example:

```
union Maybe<T> {
  case Some(T)
  case None
}

union Tristate<T> {
  case Initialized(T)
  case Initializing
  case Uninitialized
}
```

Generic unions with multiple payload cases are still not yet implemented.

2013-09-11

- The implementation now supports partial application of class and struct methods:

```
(swift) class B { func foo() { println("B") } }
(swift) class D : B { func foo() { println("D") } }
(swift) var foo = B().foo
// foo : () -> () = <unprintable value>
(swift) foo()
B
(swift) foo = D().foo
(swift) foo()
D
```

Support for partial application of Objective-C class methods and methods in generic contexts is still incomplete.

2013-09-04

- Local variable declarations without an initializer are no longer implicitly constructed. The compiler now verifies that they are initialized on all paths leading to a use of the variable. This means that constructs like this are now allowed:

  ```
  var p : SomeProtocol
  if whatever {
    p = foo()
  } else {
    p = bar()
  }
  ```

  where before, the compiler would reject the definition of `p` saying that it needed an initializer expression.

  Since all local variables must be initialized before use, simple things like this are now rejected as well:

  ```
  var x : Int
  print(x)
  ```

  The fix is to initialize the value on all paths, or to explicitly default initialize the value in the declaration, e.g. with `var x = 0` or with `var x = Int()` (which works for any default-constructible type).

- The implementation now supports unions containing protocol types and weak reference types.

- The type annotation syntax, `x as T`, has been removed from the language. The checked cast operations `x as! T` and `x is T` still remain.

2013-08-28

- `this` has been renamed to `self`. Similarly, `This` has been renamed to `Self`.

- Swift now supports unions. Unlike C unions, Swift's `union` is type-safe and always knows what type it contains at runtime. Union members are labeled using `case` declarations; each case may have a different set of types or no type:

  ```
  union MaybeInt {
    case Some(Int)
    case None
  }

  union HTMLTag {
    case A(href:String)
    case IMG(src:String, alt:String)
    case BR
  }
  ```

  Each `case` with a type defines a static constructor function for the union type. `case` declarations without types become static members:

  ```
  var br = HTMLTag.BR
  var a = HTMLTag.A(href:"http://www.apple.com/")
  // 'HTMLTag' scope deduced for '.IMG' from context
  var img : HTMLTag = .IMG(src:"http://www.apple.com/mac-pro.png",
                           alt:"The new Mac Pro")
  ```

  Cases can be pattern-matched using `switch`:

  ```
  switch tag {
  case .BR:
    println("<br>")
  case .IMG(var src, var alt):
    println("<img src=\"\(escape(src))\" alt=\"\(escape(alt))\">")
  case .A(var href):
    println("<a href=\"\(escape(href))\">")
  }
  ```

  Due to implementation limitations, recursive unions are not yet supported.

- Swift now supports autolinking, so importing frameworks or Swift libraries should no longer require adding linker flags or modifying your project file.

2013-08-14

- Swift now supports weak references by applying the `[weak]` attribute to a variable declaration.

  ```
  (swift) var x = NSObject()
  // x : NSObject = <NSObject: 0x7f95d5804690>
  (swift) var [weak] w = x
  // w : NSObject = <NSObject: 0x7f95d5804690>
  ```

```
(swift) w == nil
// r2 : Bool = false
(swift) x = NSObject()
(swift) w == nil
// r3 : Bool = true
```

Swift also supports a special form of weak reference, called `[unowned]`, for references that should never be `nil` but are required to be weak to break cycles, such as parent or sibling references. Accessing an `[unowned]` reference asserts that the reference is still valid and implicitly promotes the loaded reference to a strong reference, so it does not need to be loaded and checked for nullness before use like a true `[weak]` reference.

```
class Parent {
  var children : Array<Child>

  func addChild(c:Child) {
    c.parent = this
    children.append(c)
  }
}

class Child {
  var [unowned] parent : Parent
}
```

2013-07-31

- Numeric literals can now use underscores as separators. For example:

  ```
  var billion = 1_000_000_000
  var crore = 1_00_00_000
  var MAXINT = 0x7FFF_FFFF_FFFF_FFFF
  var SMALLEST_DENORM = 0x0.0000_0000_0000_1p-1022
  ```

- Types conforming to protocols now must always declare the conformance in their inheritance clause.

- The build process now produces serialized modules for the standard library, greatly improving build times.

2013-07-24

- Arithmetic operators `+`, `-`, `*`, and `/` on integer types now do overflow checking and trap on overflow. A parallel set of masking operators, `&+`, `&-`, `&*`, and `&/`, are defined to perform two's complement wrapping arithmetic for all signed and unsigned integer types.

- Debugger support. Swift has a -g command line switch that turns on debug info for the compiled output. Using the standard lldb debugger this will allow single-stepping through Swift programs, printing backtraces, and navigating through stack frames; all in sync with the corresponding Swift source code. An unmodified lldb cannot inspect any variables.

  Example session:

  ```
  $ echo 'println("Hello World")' >hello.swift
  $ swift hello.swift -c -g -o hello.o
  $ ld hello.o "-dynamic" "-arch" "x86_64" "-macosx_version_min" "10.9.0" \
       -framework Foundation lib/swift/libswift_stdlib_core.dylib \
       lib/swift/libswift_stdlib_posix.dylib -lSystem -o hello
  $ lldb hello
  Current executable set to 'hello' (x86_64).
  (lldb) b top_level_code
  Breakpoint 1: where = hello`top_level_code + 26 at hello.swift:1, addre...
  (lldb) r
  Process 38592 launched: 'hello' (x86_64)
  Process 38592 stopped
  * thread #1: tid = 0x1599fb, 0x0000000100000f2a hello`top_level_code + ...
      frame #0: 0x0000000100000f2a hello`top_level_code + 26 at hello.shi...
  -> 1         println("Hello World")
  (lldb) bt
  * thread #1: tid = 0x1599fb, 0x0000000100000f2a hello`top_level_code + ...
      frame #0: 0x0000000100000f2a hello`top_level_code + 26 at hello.shi...
      frame #1: 0x0000000100000f5c hello`main + 28
      frame #2: 0x00007fff918605fd libdyld.dylib`start + 1
      frame #3: 0x00007fff918605fd libdyld.dylib`start + 1
  ```

  Also try s, n, up, down.

2013-07-17

- Swift now has a switch statement, supporting pattern matching of multiple values with variable bindings, guard expressions, and range comparisons. For example:

  ```
  func classifyPoint(point:(Int, Int)) {
    switch point {
    case (0, 0):
      println("origin")

    case (_, 0):
      println("on the x axis")

    case (0, _):
      println("on the y axis")
  ```

```
    case (var x, var y) where x == y:
      println("on the y = x diagonal")

    case (var x, var y) where -x == y:
      println("on the y = -x diagonal")

    case (-10..10, -10..10):
      println("close to the origin")

    case (var x, var y):
      println("length \(sqrt(x*x + y*y))")
    }
  }
```

2013-07-10

- Swift has a new closure syntax. The new syntax eliminates the use of pipes. Instead, the closure signature is written the same way as a function type and is separated from the body by the `in` keyword. For example:

  ```
  sort(fruits) { (lhs : String, rhs : String) -> Bool in
    return lhs > rhs
  }
  ```

  When the types are omitted, one can also omit the parentheses, e.g.,

  ```
  sort(fruits) { lhs, rhs in lhs > rhs }
  ```

  Closures with no parameters or that use the anonymous parameters (`$0`, `$1`, etc.) don't need the `in`, e.g.,

  ```
  sort(fruits) { $0 > $1 }
  ```

- `nil` can now be used without explicit casting. Previously, `nil` had type `NSObject`, so one would have to write (e.g.) `nil as! NSArray` to create a `nil` `NSArray`. Now, `nil` picks up the type of its context.

- `POSIX.EnvironmentVariables` and `swift.CommandLineArguments` global variables were merged into a `swift.Process` variable. Now you can access command line arguments with `Process.arguments`. In order to access environment variables add `import POSIX` and use `Process.environmentVariables`.