# Writing High-Performance Swift Code

The following document is a gathering of various tips and tricks for writing high-performance Swift code. The intended audience of this document is compiler and standard library developers.

Some of the tips in this document can help improve the quality of your Swift program and make your code less error prone and more readable. Explicitly marking final-classes and class-protocols are two obvious examples. However some of the tips described in this document are unprincipled, twisted and come to solve a specific temporary limitation of the compiler or the language. Many of the recommendations in this document come with trade offs for things like program runtime, binary size, code readability, etc.

## Enabling Optimizations

The first thing one should always do is to enable optimization. Swift provides three different optimization levels:

- `-Onone`: This is meant for normal development. It performs minimal optimizations and preserves all debug info.
- `-O`: This is meant for most production code. The compiler performs aggressive optimizations that can drastically change the type and amount of emitted code. Debug information will be emitted but will be lossy.
- `-Osize`: This is a special optimization mode where the compiler prioritizes code size over performance.

In the Xcode UI, one can modify the current optimization level as follows:

In the Project Navigator, select the project icon to enter the Project Editor. In the project editor, select the icon under the "Project" header to enter the project settings editor. From there, an optimization setting can be applied to every target in the project by changing the "Optimization Level" field under the "Build Settings" header.

To apply a custom optimization level to a particular target, select that target under the "Targets" header in the Project Editor and override the "Optimization Level" field under its "Build Settings" header.

If a given optimization level is not available in the UI, its corresponding flag can be manually specified by selecting the `Other...` level in the "Optimization Level" dropdown.

## Whole Module Optimizations (WMO)

By default Swift compiles each file individually. This allows Xcode to compile multiple files in parallel very quickly. However, compiling each file separately prevents certain compiler optimizations. Swift can also compile the entire program as if it were one file and optimize the program as if it were a single compilation unit. This mode is enabled using the `swiftc` command line flag `-whole-module-optimization`. Programs that are compiled in this mode will most likely take longer to compile, but may run faster.

This mode can be enabled using the Xcode build setting 'Whole Module Optimization'.

NOTE: In sections below, for brevity purposes, we will refer to 'Whole Module Optimization' by the abbreviation 'WMO'.

## Reducing Dynamic Dispatch

Swift by default is a very dynamic language like Objective-C. Unlike Objective-C, Swift gives the programmer the ability to improve runtime performance when necessary by removing or reducing this dynamism. This section goes through several examples of language constructs that can be used to perform such an operation.

### Dynamic Dispatch

Classes use dynamic dispatch for methods and property accesses by default. Thus in the following code snippet, `a.aProperty`, `a.doSomething()` and `a.doSomethingElse()` will all be invoked via dynamic dispatch:

```
class A {
  var aProperty: [Int]
  func doSomething() { ... }
  dynamic doSomethingElse() { ... }
}

class B: A {
  override var aProperty {
    get { ... }
    set { ... }
  }

  override func doSomething() { ... }
}

func usingAnA(_ a: A) {
  a.doSomething()
  a.aProperty = ...
}
```

In Swift, dynamic dispatch defaults to indirect invocation through a vtable [1]. If one attaches the `dynamic` keyword to the declaration, Swift will emit calls via Objective-C message send instead. In both cases this is slower than a direct function call because it prevents many compiler optimizations [2] in addition to the overhead of performing the indirect call itself. In performance critical code, one often will want to restrict this dynamic behavior.

### Advice: Use 'final' when you know the declaration does not need to be overridden

The `final` keyword is a restriction on a declaration of a class, a method, or a property such that the declaration cannot be overridden. This implies that the compiler can emit direct function calls instead of indirect calls. For instance in the following `C.array1` and `D.array1` will be accessed directly [3]. In contrast, `D.array2` will be called via a vtable:

```
final class C {
  // No declarations in class 'C' can be overridden.
  var array1: [Int]
  func doSomething() { ... }
}

class D {
  final var array1: [Int] // 'array1' cannot be overridden by a computed property.
  var array2: [Int]      // 'array2' *can* be overridden by a computed property.
}

func usingC(_ c: C) {
   c.array1[i] = ... // Can directly access C.array without going through dynamic dispatch.
   c.doSomething() = ... // Can directly call C.doSomething without going through virtual dispatch.
}

func usingD(_ d: D) {
   d.array1[i] = ... // Can directly access D.array1 without going through dynamic dispatch.
   d.array2[i] = ... // Will access D.array2 through dynamic dispatch.
}
```

### Advice: Use 'private' and 'fileprivate' when declaration does not need to be accessed outside of file

Applying the `private` or `fileprivate` keywords to a declaration restricts the visibility of the declaration to the file in which it is declared. This allows the compiler to be able to ascertain all other potentially overriding declarations. Thus the absence of any such declarations enables the compiler to infer the `final` keyword automatically and remove indirect calls for methods and field accesses accordingly. For instance in the following, `e.doSomething()` and `f.myPrivateVar`, will be able to be accessed directly assuming `E`, `F` do not have any overriding declarations in the same file:

```
private class E {
  func doSomething() { ... }
}
```

```
class F {
  fileprivate var myPrivateVar: Int
}

func usingE(_ e: E) {
  e.doSomething() // There is no sub class in the file that declares this class.
                  // The compiler can remove virtual calls to doSomething()
                  // and directly call E's doSomething method.
}

func usingF(_ f: F) -> Int {
  return f.myPrivateVar
}
```

### Advice: If WMO is enabled, use 'internal' when a declaration does not need to be accessed outside of module

WMO (see section above) causes the compiler to compile a module's sources all together at once. This allows the optimizer to have module wide visibility when compiling individual declarations. Since an internal declaration is not visible outside of the current module, the optimizer can then infer *final* by automatically discovering all potentially overridding declarations.

NOTE: Since in Swift the default access control level is `internal` anyways, by enabling Whole Module Optimization, one can gain additional devirtualization without any further work.

## Using Container Types Efficiently

An important feature provided by the Swift standard library are the generic containers Array and Dictionary. This section will explain how to use these types in a performant manner.

### Advice: Use value types in Array

In Swift, types can be divided into two different categories: value types (structs, enums, tuples) and reference types (classes). A key distinction is that value types cannot be included inside an NSArray. Thus when using value types, the optimizer can remove most of the overhead in Array that is necessary to handle the possibility of the array being backed an NSArray.

Additionally, in contrast to reference types, value types only need reference counting if they contain, recursively, a reference type. By using value types without reference types, one can avoid additional retain, release traffic inside Array.

```
// Don't use a class here.
struct PhonebookEntry {
  var name: String
  var number: [Int]
}

var a: [PhonebookEntry]
```

Keep in mind that there is a trade-off between using large value types and using reference types. In certain cases, the overhead of copying and moving around large value types will outweigh the cost of removing the bridging and retain/release overhead.

### Advice: Use ContiguousArray with reference types when NSArray bridging is unnecessary

If you need an array of reference types and the array does not need to be bridged to NSArray, use ContiguousArray instead of Array:

```
class C { ... }
var a: ContiguousArray<C> = [C(...), C(...), ..., C(...)]
```

### Advice: Use inplace mutation instead of object-reassignment

All standard library containers in Swift are value types that use COW (copy-on-write) [4] to perform copies instead of explicit copies. In many cases this allows the compiler to elide unnecessary copies by retaining the container instead of performing a deep copy. This is done by only copying the underlying container if the reference count of the container is greater than 1 and the container is mutated. For instance in the following, no copying will occur when `c` is assigned to `d`, but when `d` undergoes structural mutation by appending `2`, `d` will be copied and then `2` will be appended to `d`:

```
var c: [Int] = [ ... ]
var d = c        // No copy will occur here.
d.append(2)      // A copy *does* occur here.
```

Sometimes COW can introduce additional unexpected copies if the user is not careful. An example of this is attempting to perform mutation via object-reassignment in functions. In Swift, all parameters are passed in at +1, i.e. the parameters are retained before a callsite, and then are released at the end of the callee. This means that if one writes a function like the following:

```
func append_one(_ a: [Int]) -> [Int] {
  var a = a
  a.append(1)
  return a
}

var a = [1, 2, 3]
a = append_one(a)
```

`a` may be copied [5] despite the version of `a` without one appended to it has no uses after `append_one` due to the assignment. This can be avoided through the usage of `inout` parameters:

```
func append_one_in_place(a: inout [Int]) {
  a.append(1)
}

var a = [1, 2, 3]
append_one_in_place(&a)
```

## Wrapping operations

Swift eliminates integer overflow bugs by checking for overflow when performing normal arithmetic. These checks may not be appropriate in high performance code if one either knows that overflow cannot occur, or that the result of allowing the operation to wrap around is correct.

### Advice: Use wrapping integer arithmetic when you can prove that overflow cannot occur

In performance-critical code you can use wrapping arithmetic to avoid overflow checks if you know it is safe.

```
a: [Int]
b: [Int]
c: [Int]

// Precondition: for all a[i], b[i]: a[i] + b[i] either does not overflow,
// or the result of wrapping is desired.
for i in 0 ... n {
  c[i] = a[i] &+ b[i]
}
```

It's important to note that the behavior of the `&+`, `&-`, and `&*` operators is fully-defined; the result simply wraps around if it would overflow. Thus, `Int.max &+ 1` is guaranteed to be `Int.min` (unlike in C, where `INT_MAX + 1` is undefined behavior).

## Generics

Swift provides a very powerful abstraction mechanism through the use of generic types. The Swift compiler emits one block of concrete code that can perform `MySwiftFunc<T>` for any `T`. The generated code takes a table of function pointers and a box containing `T` as additional parameters. Any differences in behavior between `MySwiftFunc<Int>` and `MySwiftFunc<String>` are accounted for by passing a different table of function pointers and the size abstraction provided by the box. An example of generics:

```
class MySwiftFunc<T> { ... }

MySwiftFunc<Int> X    // Will emit code that works with Int...
MySwiftFunc<String> Y // ... as well as String.
```

When optimizations are enabled, the Swift compiler looks at each invocation of such code and attempts to ascertain the concrete (i.e. non-generic type) used in the invocation. If the generic function's definition is visible to the optimizer and the concrete type is known, the Swift compiler will emit a version of the generic function specialized to the specific type. This process, called *specialization*, enables the removal of the overhead associated with generics. Some more examples of generics:

```
class MyStack<T> {
  func push(_ element: T) { ... }
  func pop() -> T { ... }
}

func myAlgorithm<T>(_ a: [T], length: Int) { ... }

// The compiler can specialize code of MyStack<Int>
var stackOfInts: MyStack<Int>
// Use stack of ints.
for i in ... {
  stack.push(...)
  stack.pop(...)
}

var arrayOfInts: [Int]
// The compiler can emit a specialized version of 'myAlgorithm' targeted for
// [Int]' types.
```

```
myAlgorithm(arrayOfInts, arrayOfInts.length)
```

### Advice: Put generic declarations in the same module where they are used

The optimizer can only perform specialization if the definition of the generic declaration is visible in the current Module. This can only occur if the declaration is in the same file as the invocation of the generic, unless the `-whole-module-optimization` flag is used. *NOTE* The standard library is a special case. Definitions in the standard library are visible in all modules and available for specialization.

## The cost of large Swift values

In Swift, values keep a unique copy of their data. There are several advantages to using value-types, like ensuring that values have independent state. When we copy values (the effect of assignment, initialization, and argument passing) the program will create a new copy of the value. For some large values these copies could be time consuming and hurt the performance of the program.

Consider the example below that defines a tree using "value" nodes. The tree nodes contain other nodes using a protocol. In computer graphics scenes are often composed from different entities and transformations that can be represented as values, so this example is somewhat realistic.

```
protocol P {}
struct Node: P {
  var left, right: P?
}

struct Tree {
  var node: P?
  init() { ... }
}
```

When a tree is copied (passed as an argument, initialized or assigned) the whole tree needs to be copied. In the case of our tree this is an expensive operation that requires many calls to malloc/free and a significant reference counting overhead.

However, we don't really care if the value is copied in memory as long as the semantics of the value remains.

### Advice: Use copy-on-write semantics for large values

To eliminate the cost of copying large values adopt copy-on-write behavior. The easiest way to implement copy-on-write is to compose existing copy-on-write data structures, such as Array. Swift arrays are values, but the content of the array is not copied around every time the array is passed as an argument because it features copy-on-write traits.

In our Tree example we eliminate the cost of copying the content of the tree by wrapping it in an array. This simple change has a major impact on the performance of our tree data structure, and the cost of passing the array as an argument drops from being O(n), depending on the size of the tree to O(1).

```
struct Tree: P {
  var node: [P?]
  init() {
    node = [thing]
  }
}
```

There are two obvious disadvantages of using Array for COW semantics. The first problem is that Array exposes methods like "append" and "count" that don't make any sense in the context of a value wrapper. These methods can make the use of the reference wrapper awkward. It is possible to work around this problem by creating a wrapper struct that will hide the unused APIs and the optimizer will remove this overhead, but this wrapper will not solve the second problem. The Second problem is that Array has code for ensuring program safety and interaction with Objective-C. Swift checks if indexed accesses fall within the array bounds and when storing a value if the array storage needs to be extended. These runtime checks can slow things down.

An alternative to using Array is to implement a dedicated copy-on-write data structure to replace Array as the value wrapper. The example below shows how to construct such a data structure:

```
final class Ref<T> {
  var val: T
  init(_ v: T) {val = v}
}

struct Box<T> {
    var ref: Ref<T>
    init(_ x: T) { ref = Ref(x) }

    var value: T {
        get { return ref.val }
        set {
          if !isKnownUniquelyReferenced(&ref) {
            ref = Ref(newValue)
```

```
                return
            }
            ref.val = newValue
        }
    }
}
```

The type `Box` can replace the array in the code sample above.

## Unsafe code

Swift classes are always reference counted. The Swift compiler inserts code that increments the reference count every time the object is accessed. For example, consider the problem of scanning a linked list that's implemented using classes. Scanning the list is done by moving a reference from one node to the next: `elem = elem.next`. Every time we move the reference Swift will increment the reference count of the `next` object and decrement the reference count of the previous object. These reference count operations are expensive and unavoidable when using Swift classes.

```
final class Node {
 var next: Node?
 var data: Int
 ...
}
```

### Advice: Use unmanaged references to avoid reference counting overhead

Note, `Unmanaged<T>._withUnsafeGuaranteedRef` is not a public API and will go away in the future. Therefore, don't use it in code that you can not change in the future.

In performance-critical code you can choose to use unmanaged references. The `Unmanaged<T>` structure allows developers to disable automatic reference counting for a specific reference.

When you do this, you need to make sure that there exists another reference to instance held by the `Unmanaged` struct instance for the duration of the use of `Unmanaged` (see Unmanaged.swift for more details) that keeps the instance alive.

```
// The call to ``withExtendedLifetime(Head)`` makes sure that the lifetime of
// Head is guaranteed to extend over the region of code that uses Unmanaged
// references. Because there exists a reference to Head for the duration
// of the scope and we don't modify the list of ``Node``s there also exist a
// reference through the chain of ``Head.next``, ``Head.next.next``, ...
// instances.

withExtendedLifetime(Head) {

  // Create an Unmanaged reference.
  var Ref: Unmanaged<Node> = Unmanaged.passUnretained(Head)

  // Use the unmanaged reference in a call/variable access. The use of
  // _withUnsafeGuaranteedRef allows the compiler to remove the ultimate
  // retain/release across the call/access.

  while let Next = Ref._withUnsafeGuaranteedRef { $0.next } {
    ...
    Ref = Unmanaged.passUnretained(Next)
  }
}
```

## Protocols

### Advice: Mark protocols that are only satisfied by classes as class-protocols

Swift can limit protocols adoption to classes only. One advantage of marking protocols as class-only is that the compiler can optimize the program based on the knowledge that only classes satisfy a protocol. For example, the ARC memory management system can easily retain (increase the reference count of an object) if it knows that it is dealing with a class. Without this knowledge the compiler has to assume that a struct may satisfy the protocol and it needs to be prepared to retain or release non-trivial structures, which can be expensive.

If it makes sense to limit the adoption of protocols to classes then mark protocols as class-only protocols to get better runtime performance.

```
protocol Pingable: AnyObject { func ping() -> Int }
```

## The Cost of Let/Var when Captured by Escaping Closures

While one may think that the distinction in between let/var is just about language semantics, there are also performance considerations. Remember that any time one creates a binding for a closure, one is forcing the compiler to emit an escaping closure, e.x.:

```
let f: () -> () = { ... } // Escaping closure
// Contrasted with:
({ ... })() // Non Escaping closure
x.map { ... } // Non Escaping closure
```

When a var is captured by an escaping closure, the compiler must allocate a heap box to store the var so that both the closure creator/closure can read/write to the value. This even includes situations where the underlying type of the captured binding is trivial! In contrast, when captured a *let* is captured by value. As such, the compiler stores a copy of the value directly into the closure's storage without needing a box.

### Advice: Pass var as an *inout* if closure not actually escaping

If one is using an escaping closure for expressivity purposes, but is actually using a closure locally, pass vars as inout parameters instead of by using captures. The inout will ensure that a heap box is not allocated for the variables and avoid any retain/release traffic from the heap box being passed around.

## Unsupported Optimization Attributes

Some underscored type attributes function as optimizer directives. Developers are welcome to experiment with these attributes and send back bug reports and other feedback, including meta bug reports on the following incomplete documentation: :ref:`UnsupportedOptimizationAttributes`. These attributes are not supported language features. They have not been reviewed by Swift Evolution and are likely to change between compiler releases.

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main) (docs)OptimizationTips.rst`, line 612);** ***backlink***

Unknown interpreted text role "ref".

## Footnotes

[1] A virtual method table or 'vtable' is a type specific table referenced by instances that contains the addresses of the type's methods. Dynamic dispatch proceeds by first looking up the table from the object and then looking up the method in the table.

[2] This is due to the compiler not knowing the exact function being called.

[3] i.e. a direct load of a class's field or a direct call to a function.

[4] An optimization technique in which a copy will be made if and only if a modification happens to the original copy, otherwise a pointer will be given.

[5] In certain cases the optimizer is able to via inlining and ARC optimization remove the retain, release causing no copy to occur.