Inside Swift's Memory Management

ARC, Runtime, and Compiler Behavior

Author: Erkan Demir

January 31, 2026

Contents

Inside Swift's Memory Management

ARC, Runtime, and Compiler Behavior

Erkan Demir

1 Introduction

Memory management is a critical aspect of software development that is often overlooked. When mishandled, its consequences are immediate and tangible: applications can crash, consume excessive resources, or behave inconsistently, frustrating both users and developers. Swift addresses these challenges through Automatic Reference Counting (ARC), which leverages the runtime's `HeapObject` structures, side tables, and compiler optimizations to manage memory safely and efficiently. A clear understanding of the Swift runtime and its reference-counting model is therefore essential for building performant and consistent applications.

2 Swift Runtime

Behind the scenes, Swift relies on its runtime to support fundamental execution processes. Among its responsibilities, the Swift runtime plays a central role in memory management [1] (`docs/Runtime.md`, *The Swift Runtime*).

2.1 Allocations

Swift manages memory by automatically allocating data to different memory regions based on data characteristics. In practice, Swift primarily uses two locations for allocation: the stack and the heap. The compiler decides where data should be stored without requiring explicit developer intervention [2] (`documentation/server/guides/allocations.html`, *Allocations / Heaps and stacks*).

The stack provides a simple and efficient mechanism for accessing data using a last-in, first-out (LIFO) model. Each thread maintains an independent stack. When a function is invoked, a stack frame containing local variables, parameters, and the return address is pushed onto the stack and removed once the function returns [3] (pp. 37–38, 113, 177–179).

In contrast, the heap is designed for dynamic memory allocation. Heap-allocated memory can be accessed across threads, but this flexibility comes at a cost. Heap allocation is generally slower than stack allocation due to its more complex structure and the synchronization required to ensure thread safety [4] (pp. 875, 880–881, 1029–1031).

2.2 Heap Object

To track and manage object lifetimes efficiently, Swift represents heap-allocated objects using a runtime structure known as `HeapObject`. This structure includes a header that stores metadata and reference-counting information [1] (`include/swift/Runtime/HeapObject.h`, L45–67; `stdlib/public/SwiftShims/swift/shims/HeapObject.h`, L45–58).

Swift also applies several compiler optimizations to reduce unnecessary heap allocations. When the compiler determines that an object's lifetime does not escape a function, it may apply *stack promotion*, allocating the object on the stack instead of the heap [1] (`SwiftCompilerSources/Sources/Optimizer/FunctionPasses/StackPromotion.swift`, L16–24). Similarly, closures capture local variables on the heap by default. However, if the compiler can prove that a captured variable is not mutated either within the closure or after its creation, it may perform *capture promotion*, converting a by-reference capture into a by-value copy [1] (`lib/SILOptimizer/Mandatory/CapturePromotion.cpp`, L18–43).

These optimizations reduce heap pressure and improve overall performance while preserving program semantics.

2.3 Side Table

The Swift runtime introduces an additional level of indirection to support weak references. Initially, strong and unowned reference counts are stored directly within the `HeapObject`. When a weak reference is created, or when the inline reference-count storage overflows, a side table is allocated. At this point, the strong, unowned, and weak reference counts are migrated to the side table [1] (`stdlib/public/SwiftShims/swift/shims/RefCount.h`, L63–96; `stdlib/public/SwiftShims/swift/shims/RefCount.h`, L556–564).

The side table maintains a reference to the associated `HeapObject`. Strong and unowned references continue to point directly to the object, whereas weak references point to the side table. This design enables weak references to safely observe object deallocation, at the cost of additional indirection.

2.4 Existential Containers

In statically typed languages, representing heterogeneous values under a uniform abstraction presents a fundamental challenge. Swift addresses this problem through *existential containers* (Listing 1), which implement type erasure by introducing an additional level of indirection. As a result, the runtime must resolve the concrete type dynamically and dispatch protocol method calls through witness tables rather than via static dispatch (Listing 2). This abstraction provides flexibility but incurs performance overhead [5] (`opaquetypes`, *Opaque and Boxed Protocol Types / Boxed Protocol Types*).

```swift
protocol Shape {}

struct Triangle: Shape {}
struct Square: Shape {}

let array: [any Shape] = [Triangle(), Square()]
```

Listing 1: Each element of the `[any Shape]` array is stored in an existential container.

Swift defines two representations for these containers. `ClassExistentialContainer` directly references a `HeapObject` for class-bounded types, while `OpaqueExistentialContainer` provides a fixed-size inline buffer for value types that are not class-bounded [1] (`include/swift/Runtime/ExistentialContainer.h`). When a value fits within the inline buffer, it is stored directly in the container. Otherwise, the buffer holds a pointer to a heap-allocated `HeapObject` [1] (`stdlib/public/runtime/ExistentialContainer.cpp`, L27–40).

```swift
struct ClassExistentialContainer {
  HeapObject *value;
  WitnessTable *witnessTables[NUM_WITNESS_TABLES];
};

struct OpaqueExistentialContainer {
  void *fixedSizeBuffer[3];
  Metadata *type;
  WitnessTable *witnessTables[NUM_WITNESS_TABLES];
};
```

Listing 2: Memory layout of class-based and opaque existential containers in the Swift runtime [1] (`docs/ABI/TypeLayout.rst`, *Existential Container Layout*).

Where possible, the compiler mitigates existential overhead by converting existential parameters into generics and specializing the corresponding functions. This specialization allows static dispatch and avoids unnecessary runtime indirection [1] (`lib/SILOptimizer/FunctionSignatureTransforms/ExistentialSpecializer.cpp`, L14–14; `lib/SILOptimizer/Transforms/GenericSpecializer.cpp`, L13–14).

3 Reference Counting

Objective-C originally relied on manual retain–release (MRR), a memory management model in which developers explicitly controlled object lifetimes by sending `retain` and `release` messages [6] (`Articles/MemoryMgmt.html`, *About Memory Management / At a Glance / 1*; `Articles/mmPractical.html`, *Practical Memory Management / Ownership Policy Is Implemented Using Retain Counts*). While effective, MRR imposed significant cognitive and maintenance overhead.

Autorelease pools were introduced to reduce this burden by deferring and batching release operations. At the end of an autorelease pool's scope, all objects that received `autorelease`

messages within that scope are automatically released [6] (`Articles/mmAutoreleasePools.html`, *Using Autorelease Pool Blocks*).

Swift does not support MRR directly. Autorelease pools exist solely to enable interoperability with Objective-C. Instead, Swift employs Automatic Reference Counting (ARC). ARC uses the same underlying reference-counting mechanism as MRR but delegates the insertion of retain and release operations to the compiler [6] (`Articles/MemoryMgmt.html`, *About Memory Management / At a Glance / 2*), [5] (`automaticreferencecounting`, *Automatic Reference Counting*).

Under ARC, a `HeapObject` remains allocated as long as strong, unowned, or weak references exist (Figure 1). Once the object enters the deinitializing state, however, accessing it through an unowned reference results in a runtime crash (Appendix A: *Unowned Reference Crashes*). Weak references, by contrast, safely return `nil` under the same conditions (Appendix B: *Weak References Returning Nil*).

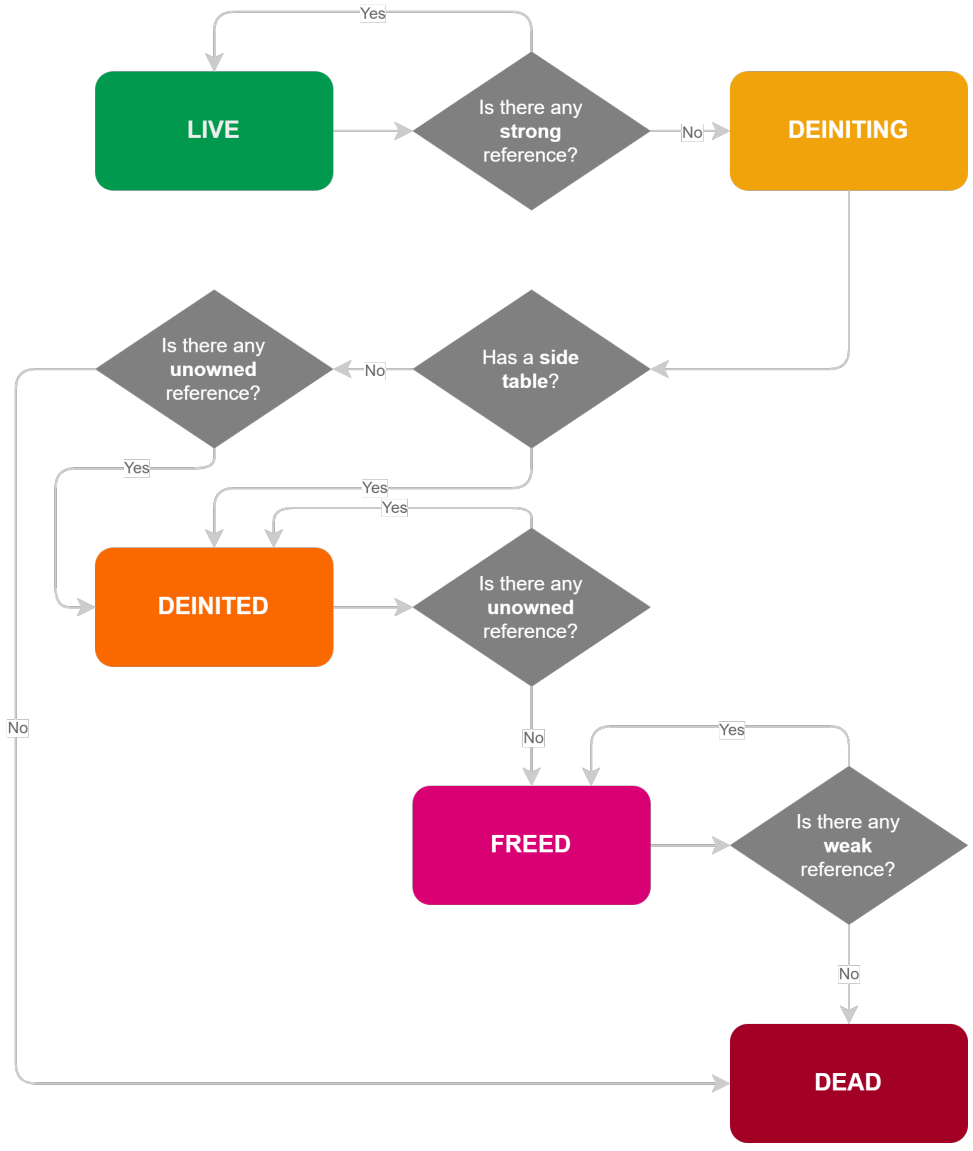

Figure 1: `HeapObject` lifecycle in the Swift runtime, based on the object lifecycle state machine [1] (`stdlib/public/SwiftShims/swift/shims/RefCount.h`, L112–176)

4 Performance

Two common sources of memory-related performance issues in Swift are memory leaks and excessive heap allocation. Both increase runtime overhead and can degrade application responsiveness.

4.1 Memory Leaks

Memory leaks occur when object lifetimes are mismanaged, reducing available heap memory and impairing allocation efficiency. In severe cases, leaks may lead to application termination due to memory exhaustion [2] (`documentation/server/guides/memory-leaks-and-usage.html`, *Debugging Memory Leaks and Usage / Overview*), [7] (`reducing-your-app-s-memory-use`, *Reducing Your App's Memory Use / Overview*).

The most frequent cause of memory leaks in Swift is a retain cycle, in which two or more objects hold strong references to one another, preventing their reference counts from reaching zero [5] (`automaticreferencecounting`, *Automatic Reference Counting / Strong Reference Cycles Between Class Instances*). Additionally, tight loops that generate large numbers of Objective-C objects without an explicit autorelease pool can cause rapid memory accumulation, producing behavior similar to a leak [6] (`Articles/mmAutoreleasePools.html`, *Using Autorelease Pool Blocks*).

Objects retained by weak or unowned references may also contribute to heap pressure by prolonging object lifetime beyond its logical usefulness (Section 3: *Reference Counting*).

4.2 Excessive Heap Allocations

As discussed earlier (Section 2.1: *Allocations*), heap allocations are slower than stack allocations due to their complexity and synchronization requirements. Excessive heap usage can therefore have a significant impact on performance.

Accessing unowned references introduces additional overhead, as the runtime must verify object validity (Appendix A: *Unowned Reference Crashes*). Weak references add further indirection through the side table, increasing access latency (Appendix B: *Weak References Returning Nil*). Existential containers likewise increase heap usage when values exceed the size of the inline buffer.

Together, these factors demonstrate that careful attention to allocation patterns and reference types is essential for maintaining optimal performance in Swift applications.

5 Consistency

Heap-allocated objects must be managed carefully to ensure consistent program behavior. Improper handling of object lifetimes or concurrent access can result in runtime failures, nondeterministic outcomes, or violations of program invariants.

5.1 Object Lifetime

Unowned references provide a non-owning way to access objects without incrementing their reference count. However, if an object is deallocated while an unowned reference still exists,

any subsequent access results in a runtime crash (Appendix A: *Unowned Reference Crashes*). This form of unexpected deallocation undermines assumptions about object availability and compromises program consistency.

Weak references avoid crashes by automatically returning `nil` once the referenced object is deallocated (Appendix B: *Weak References Returning Nil*). While safer, this behavior can still lead to logical errors if the absence of a value is not properly handled. Selecting the appropriate reference type and accounting for deallocation behavior are therefore critical for preserving predictable program semantics.

5.2 Race Conditions

In multithreaded environments, unsynchronized access to heap-allocated objects can lead to race conditions (Listing 3). In such cases, program behavior depends on the relative timing of thread execution, producing nondeterministic, difficult-to-reproduce results [4] (pp. 1031–1034).

Unlike unowned reference crashes, which are deterministic and triggered by specific access patterns, race conditions manifest only under concurrent execution. Avoiding these issues requires proper synchronization mechanisms and disciplined concurrency design.

6 Conclusion

Swift's memory management model, built on Automatic Reference Counting, provides a robust framework for maintaining safe, predictable object lifetimes. By combining runtime structures such as `HeapObject` and side tables with compiler optimizations like stack and capture promotion, Swift balances performance with safety.

Developers who understand the behavior of strong, weak, and unowned references—as well as heap allocation patterns and concurrency concerns—are better equipped to prevent retain cycles, excessive memory usage, and runtime crashes. Thoughtful management of object lifetimes and synchronization is therefore not merely good practice; it is central to writing reliable, efficient, and maintainable Swift code.

```swift
import Foundation

final class Counter: @unchecked Sendable {
    private var value: Int = 0

    func get() -> Int { value }
    func increment() { value += 1 }
}

final class SafeCounter: @unchecked Sendable {
    private var value: Int = 0
    private let queue = DispatchQueue(label: "Counter", attributes: .concurrent)

    func get() -> Int { queue.sync { value } }
    func increment() {
        queue.async(flags: .barrier) {
            self.value += 1
        }
    }
}

func count() {
    let counter = Counter()
    let safeCounter = SafeCounter()

    DispatchQueue.concurrentPerform(iterations: 1000) { _ in
        counter.increment()
        safeCounter.increment()
    }

    print("SafeCounter: \(safeCounter.get()), Counter: \(counter.get())")
}

count(); count(); count();

// Output
// ---------------------------------------
// SafeCounter: 1000, Counter: 995
// SafeCounter: 1000, Counter: 993
// SafeCounter: 1000, Counter: 999
```

Listing 3: Race conditions lead to nondeterministic results across program executions.

References

[1] The Swift Project Contributers. *The Swift Programming Language*. Version swift-6.2.3-RELEASE. 2025. URL: https://github.com/swiftlang/swift/tree/swift-6.2.3-RELEASE (visited on 01/31/2026).

[2] Apple Inc. *Swift.org*. URL: https://swift.org (visited on 01/31/2026).

[3] Abraham Silberschatz, Peter B. Galvin and Gagne Greg. *Operating system concepts*. 10th, Global ed. Wiley, 2019.

[4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd, Global ed. Pearson, 2016.

[5] Apple Inc. and the Swift project authors. *The Swift Programming Language Documentation*. Version 6.2.3. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language (visited on 01/31/2026).

[6] Apple Inc. *Advanced Memory Management Programming Guide*. URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt (visited on 01/31/2026).

[7] Apple Inc. *Xcode Documentation*. URL: https://developer.apple.com/documentation/xcode (visited on 01/31/2026).

A Unowned Reference Crashes

```
481 /// Increment the strong retain count of an object, aborting if it has
482 /// been deallocated.
483 SWIFT_RUNTIME_EXPORT
484 HeapObject *swift_unownedRetainStrong(HeapObject *value);
```

Listing 4: Declaration of `swift_unownedRetainStrong`, which increments the strong reference count for unowned references [1] (`include/swift/Runtime/HeapObject.h`, L481–484).

```
687 HeapObject *swift::swift_unownedRetainStrong(HeapObject *object) {
688 #ifdef SWIFT_THREADING_NONE
689   return swift_nonatomic_unownedRetainStrong(object);
690 #else
691   SWIFT_RT_TRACK_INVOCATION(object, swift_unownedRetainStrong);
692   if (!isValidPointerForNativeRetain(object))
693     return object;
694   assert(object->refCounts.getUnownedCount() &&
695          "object is not currently unowned-retained");
696
697   if (! object->refCounts.tryIncrement())
698     swift::swift_abortRetainUnowned(object);
699   return object;
700 #endif
701 }
```

Listing 5: Implementation of `swift_unownedRetainStrong`, which increments the strong reference count for unowned references [1] (`stdlib/public/runtime/HeapObject.cpp`, L687–701).

```
861   // Increment the reference count, unless the object is deiniting.
862   SWIFT_ALWAYS_INLINE
863   bool tryIncrement() {
864     auto oldbits = refCounts.load(SWIFT_MEMORY_ORDER_CONSUME);
865     RefCountBits newbits;
866     do {
867       if (!oldbits.hasSideTable() && oldbits.getIsDeiniting())
868         return false;
869
870       newbits = oldbits;
871       bool fast = newbits.incrementStrongExtraRefCount(1);
872       if (SWIFT_UNLIKELY(!fast)) {
873         if (oldbits.isImmortal(false))
874           return true;
875         return tryIncrementSlow(oldbits);
876       }
877     } while (!refCounts.compare_exchange_weak(oldbits, newbits,
878                                               std::memory_order_relaxed));
879     return true;
880   }
```

Listing 6: Implementation of `tryIncrement`, which attempts to increment the strong reference count, checking for deinitialization [1] (`stdlib/public/SwiftShims/swift/shims/RefCount.h`, L861–880).

```
503 // Crash due to retain of a dead unowned reference.
504 // FIXME: can't pass the object's address from InlineRefCounts without hacks
505 void swift::swift_abortRetainUnowned(const void *object) {
506   if (object) {
507     swift::fatalError(FatalErrorFlags::ReportBacktrace,
508                       "Fatal error: Attempted to read an unowned reference but "
509                       "object %p was already deallocated\n", object);
510   } else {
511     swift::fatalError(FatalErrorFlags::ReportBacktrace,
512                       "Fatal error: Attempted to read an unowned reference but "
513                       "the object was already deallocated\n");
514   }
515 }
```

Listing 7: Implementation of `swift_abortRetainUnowned`, which triggers a runtime crash when retaining a dead unowned reference [1] (`stdlib/public/runtime/Errors.cpp`, L503–515).

B Weak References Returning Nil

```
599 /// Load a value from a weak reference.  If the current value is a
600 /// non-null object that has begun deallocation, returns null;
601 /// otherwise, retains the object before returning.
602 ///
603 /// \param ref - never null
604 /// \return can be null
605 SWIFT_RUNTIME_EXPORT
606 HeapObject *swift_weakLoadStrong(WeakReference *ref);
```

Listing 8: Declaration of `swift_weakLoadStrong`, which loads and retains a strong reference from a weak reference if valid [1] (`include/swift/Runtime/HeapObject.h`, L599–606).

```
1013 HeapObject *swift::swift_weakLoadStrong(WeakReference *ref) {
1014   return ref->nativeLoadStrong();
1015 }
```

Listing 9: Implementation of `swift_weakLoadStrong`, which loads and retains a strong reference from a weak reference if valid [1] (`stdlib/public/runtime/HeapObject.cpp`, L1013–1015).

```
233   HeapObject *nativeLoadStrong() {
234     auto bits = nativeValue.load(std::memory_order_relaxed);
235     return nativeLoadStrongFromBits(bits);
236   }
```

Listing 10: Implementation of `nativeLoadStrong`, which loads a strong reference from weak reference bits [1] (`stdlib/public/runtime/WeakReference.h`, L233–236).

```
170   HeapObject *nativeLoadStrongFromBits(WeakReferenceBits bits) {
171     auto side = bits.getNativeOrNull();
172     return side ? side->tryRetain() : nullptr;
173   }
```

Listing 11: Implementation of `nativeLoadStrongFromBits`, which loads a strong reference from the side table [1] (`stdlib/public/runtime/WeakReference.h`, L170–173).

```
1364  HeapObject* tryRetain() {
1365    if (refCounts.tryIncrement())
1366      return object.load(std::memory_order_relaxed);
1367    else
1368      return nullptr;
1369  }
```

Listing 12: Implementation of `tryRetain`, which attempts to load the object and increment its strong reference count, returning `nullptr` on failure [1] (`stdlib/public/SwiftShims/swift/shims/RefCount.h`, L1364–1369).

```
861  // Increment the reference count, unless the object is deiniting.
862  SWIFT_ALWAYS_INLINE
863  bool tryIncrement() {
864    auto oldbits = refCounts.load(SWIFT_MEMORY_ORDER_CONSUME);
865    RefCountBits newbits;
866    do {
867      if (!oldbits.hasSideTable() && oldbits.getIsDeiniting())
868        return false;
869
870      newbits = oldbits;
871      bool fast = newbits.incrementStrongExtraRefCount(1);
872      if (SWIFT_UNLIKELY(!fast)) {
873        if (oldbits.isImmortal(false))
874          return true;
875        return tryIncrementSlow(oldbits);
876      }
877    } while (!refCounts.compare_exchange_weak(oldbits, newbits,
878                                              std::memory_order_relaxed));
879    return true;
880  }
```

Listing 13: Implementation of `tryIncrement`, which attempts to increment the strong reference count, checking for deinitialization [1] (`stdlib/public/SwiftShims/swift/shims/RefCount.h`, L861–880).