2 Types & Mutation

Written by Ray Fix

What's a type? It's a logical grouping of data along with a set of operations you can perform with it. You use both standard types and custom types to build your programs in Swift. The Swift compiler often can guarantee correctness using type information in a process known as **type checking** before your program even runs.

Beginners, who are dealing with unfamiliar types such as `Optional` for the first time, might find type checking errors burdensome and slightly mysterious. But by type-checking your program, the type system makes sure you use the software correctly and enables many optimizations. Swift's type system is the key to safe and efficient code. As you become a more advanced practitioner, your programming perspective likely will become more type-centric.

Swift emphasizes **mutable value semantics** for its types. In this chapter, you'll review the important nominal types that Swift offers. Value types (enumerations and structures), reference types (classes) and the mutation rules all work together to allow mutable value semantics, which you can adopt in your own Swift types.

The fundamental types

The Swift type system consists of a small number of fundamental types. These types include the so-called **named types** (protocols, enumerations, structures and classes) as well as **compound types** (functions and tuples). Each of these types has a unique set of properties that make it useful for a particular situation.

As discussed in the previous chapter, it's pretty incredible that all the standard library types such as `Bool`, `Int`, `Double`, `String`, `Optional`, `Array` and `Dictionary` are clever compositions of these fundamental types. It speaks to the power of what you can do with them.

> **Note**: Protocols and generics also are amazing. This book has whole chapters

for protocol types and generic types because they're so powerful and important. You'll also briefly look at types of types, or metatypes, in Chapter 4, "Generics".

In this chapter, you'll explore the properties of named concrete types created from classes, structures and enumerations. Although some of this might be a review, it will provide a platform to explore some more advanced topics.

Modeling with types

Two-dimensional geometry is a great problem domain for exploring the type system because it's well defined mathematically and easy to visualize. Start by opening the **Geometry** starter playground and adding the following two definitions for a point type:

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

class ClassPoint {
  var x, y: Double
  init(x: Double, y: Double) { (self.x, self.y) = (x, y) }
}
```

COPY

Both of these types model a point in the x-y plane. But already there are five essential differences you should be aware of.

Difference 1: Automatic initialization

The first, most obvious difference is the need for an initializer in the class type. If you don't declare one, the compiler will declare an internal **member-wise initializer** for structures. This initializer simply assigns the member properties one by one. You must define an initializer for your class because `x` and `y` need to be initialized.

> **Note:** If you define an initializer for a structure, the compiler won't define the member-wise one for you. A common trick is to define your initializer in an extension if you want both the compiler-generated one and a custom one.

Difference 2: Copy semantics

The second major and probably most important difference is copy semantics. Classes have **reference semantics** and structures have **value semantics**.

Value semantics says that given two instances **A** and **B**, it's impossible to affect the value **B** by making changes to **A** and vice versa.

With reference semantics, you can affect one object from the other. Check out an example by adding this to the end of your playground and running it:

```
let structPointA = StructPoint(x: 0, y: 0)
var structPointB = structPointA
structPointB.x += 10
print(structPointA.x) // not affected, prints 0.0

let classPointA = ClassPoint(x: 0, y: 0)
let classPointB = classPointA
classPointB.x += 10
print(classPointA.x) // affected, prints 10.0
```

With reference semantics, changing `classPointB` affects `classPointA` because both variables point to the same underlying memory. This phenomenon is not the case with the structure in which `structPointA` and `structPointB` are independent copies with value semantics.

Difference 3: Scope of mutation

Swift supports an **instance-level** mutation model. This means that by using the introducer keyword `let` instead of `var`, you can lock down an instance from mutation. This is why you must declare `structPointB` in the code above with a `var`. If you didn't, you wouldn't be able to add 10 to the `x` coordinate. The compiler would prevent this with an error.

Notice that you can modify the `x` coordinate with the class version even though `let` introduces `classPointB`. The mutation control applies to the reference itself, not the underlying property data.

Value semantics through immutability

You know from the example above that classes are reference types with reference semantics. Is it possible to give a class value semantics? The answer is yes, and the easiest way is through immutability. Simply make all the properties immutable by declaring them with `let`. Because you can't modify anything from anywhere, this satisfies the definition for value semantics. Functional languages often use strict immutability at the cost of performance to achieve value semantics.

> **Note**: Objective-C uses a **type-level** mutation model. For example, `NSString` is immutable. But `NSMutableString`, which derives from `NSString`, adds mutability. However, if you have a pointer to an `NSString`, you can't be 100

percent sure it doesn't point to an `NSMutableString` that another client could modify. Defensive copies become necessary, making this a less efficient, less safe and more error-prone programming model.

The beautiful thing about declaring the `x` and `y` properties with `var` in `StructPoint` is that they can be mutable if you declare the instance with `var` and immutable with `let`. That is why you usually want to declare properties with `var` for structures, because you can control mutability for each instance at the point of use.

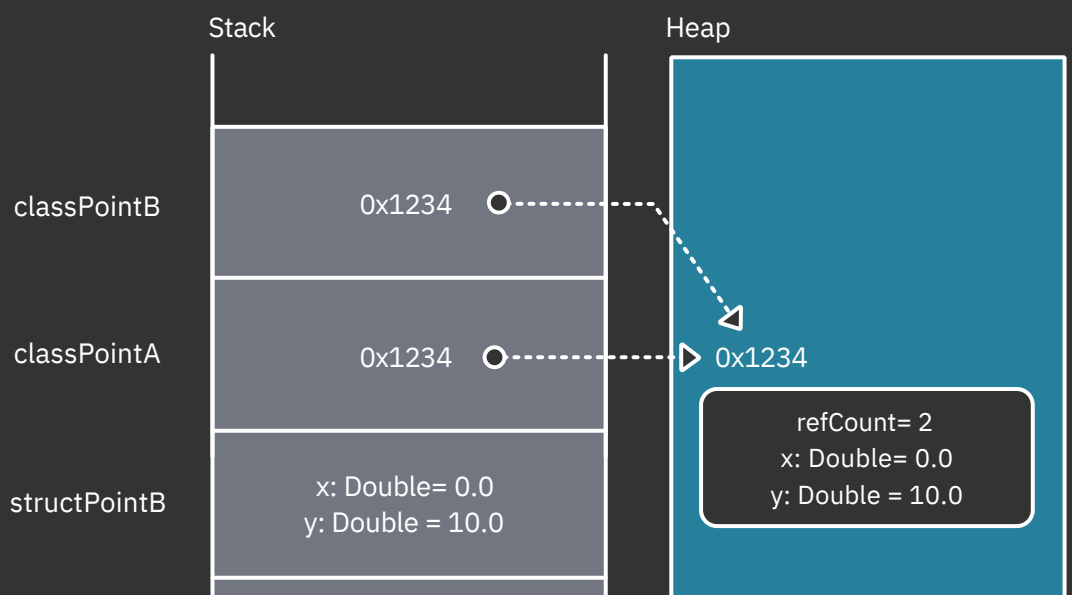
Difference 4: Heap versus stack

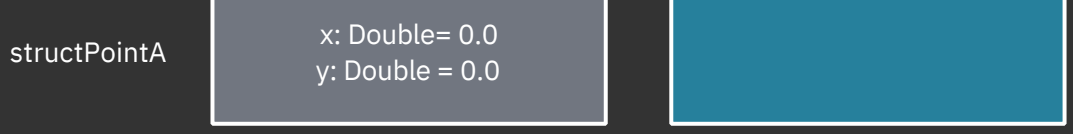
A general rule of thumb is that classes use heap memory but structures and enumerations use stack memory. Because stack allocations are orders of magnitude faster than heap allocations, this is where value types get their fast reputation.

Each thread of execution has its own stack, and stacks only change by modifying the top-most element. As a result, allocating and deallocating onto a stack doesn't require expensive concurrency locks or fancy allocation strategies. Allocation and deallocation can be performed with a single add or subtract instruction in a single clock tick.

The heap, by contrast, is shared by multiple threads and needs to be protected by concurrency locks. The operating system must protect against **heap fragmentation**, which can happen if you allocate and deallocate different size memory blocks. As a result, even though heap allocation has been highly optimized, it's ultimately non-deterministic and could require thousands or even millions of instructions to perform.

Here's a diagram of how the above code might look like allocated in memory:





Instances on the stack and heap

The structures are put on the stack, whereas the classes are put on both the stack and the heap. The reference count in heap memory keeps track of the object's lifetime because reference types are shared. Only when the reference count drops to zero does `deinit` get called and the memory deallocated.

> **Note**: Heap for classes vs. stack for structures and enumerations is just a general rule of thumb. As you saw in the previous chapter, the Swift compiler starts by allocating everything on the heap and then reasons about the object's lifetime to determine whether it can be allocated on the stack. For example, an escaping closure that closes over a local structure will need to put that object on the heap to extend the structure's lifetime beyond its scope. On the other hand, a class that is created, that performs some action and then goes out of scope might be optimized away entirely and just include the instructions necessary to complete the operation.

Difference 5: Lifetime and identity

Value types, such as structures and enumerations, generally live on the stack and are cheap to copy. Values don't have the notion of a lifetime or intrinsic identity. References do have lifetimes, and because of that, you can define a `deinit` function for them. They also automatically have an identity because they reside at a specific place in memory you can use to identify them.

> **Note**: It's possible to give a value type identity by specifying a unique property attribute. The `Identifiable` protocol, which adds a `Hashable` (and `Equatable`) `id` property, does this. The SwiftUI framework defines property wrappers, such as `@State`, which among other things imbue lifetime into simple value types.

More differences

There are other differences between classes and structures that this simple example doesn't illuminate. The most glaring one is inheritance, which classes use to realize runtime polymorphism. Classes dispatch their methods dynamically, whereas this doesn't happen for structures unless you're using a protocol. Dispatch happens statically for structure methods not part of a

protocol. You'll learn more about protocol dispatch in the next chapter.

> **Note**: You can mark methods in classes as `final`, which can have a side effect of devirtualizing them and making them run faster. The compiler can use hints from access control and whole module optimization to prove that a method can't be overridden and optimize it.

Defining a `Point`

Given the differences above, having a lightweight value representing your `Point` is likely a good choice. Go with that design. Add this to the playground:

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

struct Size: Equatable {
  var width, height: Double
}

struct Rectangle: Equatable {
  var origin: Point
  var size: Size
}
```

This defines `Point`, `Size` and `Rectangle` with `Equatable` conformance. For value types, the compiler will generate the required `==` method for you if the stored properties are also `Equatable` types. Reference types (aka classes) require you to write `==` for `Equatable` and `hash(into:)` for `Hashable` yourself.

Another essential characteristic of value semantics is that they compose. `Rectangle` has value semantics because it's a value type and both `Point` and `Size` have value semantics. Further, because Swift arrays have value semantics, an array of `Rectangle` will also have value semantics.

> **Note**: Code synthesis happens during the type-checking phase of the compiler. When you adopt a protocol, the compiler checks to see whether the type fulfills (witnesses) the protocol. If it doesn't, it typically emits an error. In the special cases of `Equatable`, if the type is a value type, it will attempt to synthesize `==` if all the stored properties are also `Equatable`. A similar process happens for `Hashable`, `Codable` and `CaseIterable`. Unlike the others, `Codable` synthesizes code for both value types and reference types.

Functions and methods

The custom types so far only have data in the form of stored properties. But things get interesting when you add operations. To warm up, add a couple of methods to the `Point` type:

```
// 1st draft version
extension Point {
  func flipped() -> Self {
    Point(x: self.y, y: self.x)
  }

  mutating func flip() {
    let temp = self
    self.x = temp.y
    self.y = temp.x
  }
}
```

COPY

Here are two simple methods for exchanging the `x` and `y` coordinates of a point. The names of the methods follow the "fluent" usage of mutating and non-mutating pairs described by the Swift API Design Guidelines (https://swift.org/documentation/api-design-guidelines/).

The function `flipped()` uses `self`, and the function `flip` both uses and modifies `self`. For that reason, you need to declare it `mutating`. Both functions contain the exchanging logic, which is repetitious.

Clean things up by replacing the code above with this version:

```
extension Point {
  func flipped() -> Self {
    Point(x: y, y: x)
  }

  mutating func flip() {
    self = flipped()
  }
}
```

COPY

The unnecessary references to `self` are gone, and the exchanging logic is *only* in `flipped`. In this case, the implementation is trivial, so the duplication wasn't a big deal. But when you have non-mutating and mutating function pairs that are more complicated, you'll appreciate this pattern.

Mutating and self

With a type method, the Swift compiler passes `self: Self` as an invisible parameter. That's why you can use it in the function body. With a `mutating`

method, Swift passes an invisible `self: inout Self`. If you recall, the semantics of `inout` make a copy on entry into the function and then make a copy on exit. This timing corresponds to the property observers `willSet` and `didSet` getting called. Further, `inout` effectively makes an input and extra return value from your function.

> **Note**: Methods on classes (i.e., reference types) don't use `inout`. If you think about what `self: inout Self` does, that makes sense. `inout` on a reference type would only prevent the entire instance from being reassigned to another instance.

Static methods and properties

Add a static property and method to your point type using an extension by adding this to the playground:

```
extension Point {
  static var zero: Point {
    Point(x: 0, y: 0)
  }

  static func random(inRadius radius: Double) -> Point {
    guard radius >= 0 else {
      return .zero
    }

    let x = Double.random(in: -radius ... radius)
    let maxY = (radius * radius - x * x).squareRoot()
    let y = Double.random(in: -maxY ... maxY)
    return Point(x: x, y: y)
  }
}
```

This code creates a static property `zero`, which is just a point at the origin. The static method `random` creates a random point bounded by the specified radius. The `x` value is first pinned down, and you use the Pythagorean theorem to determine the maximum bounds of allowed `y` values so it stays in the circle.

Going deterministic

Swift's default `Double.random(in:)` uses `SystemRandomNumberGenerator()`, which is cryptographically secure. This choice is a great default because it prevents would-be attackers from guessing your random numbers.

Sometimes, you want your random values to be deterministic and repeatable. This importance is especially true for continuous integration tests. You want

these types of tests to fail in response to a code change (bad merge or refactoring), not because of a new, untried input value. Fortunately, the Swift standard library supports your own generators with the overloaded method `Double.random(in:using:)`, where the `using` parameter takes a pseudo-random number generator of your choice.

Although the standard library doesn't include one of these seedable pseudo-random sources, it's easy to make one yourself. There's a lot of research about making "good" random generators on the web. Here is a decent one from Wikipedia. The Permuted Congruential Generator (https://en.wikipedia.org/wiki/Permuted_congruential_generator) can be translated to Swift from the listed C code. Add this to your playground:

```
struct PermutedCongruential: RandomNumberGenerator {
  private var state: UInt64
  private let multiplier: UInt64 = 6364136223846793005
  private let increment: UInt64 = 1442695040888963407

  private func rotr32(x: UInt32, r: UInt32) -> UInt32 {
    (x &>> r) | x &<< ((~r &+ 1) & 31)
  }

  private mutating func next32() -> UInt32 {
    var x = state
    let count = UInt32(x &>> 59)
    state = x &* multiplier &+ increment
    x ^= x &>> 18
    return rotr32(x: UInt32(truncatingIfNeeded: x &>> 27),
                  r: count)
  }

  mutating func next() -> UInt64 {
    UInt64(next32()) << 32 | UInt64(next32())
  }

  init(seed: UInt64) {
    state = seed &+ increment
    _ = next()
  }
}
```

This code contains some math details that aren't important for this book. (However, you will see more about C-style unsafe binary arithmetic such as `&>>`, `&*` and `&+` in Chapter 5, "Numerics & Ranges".) The critical thing to notice is how you can mark the internal details and state as private. As a user of this type, you only need to know that it's seeded with a 64-bit integer and that it produces a deterministic stream of pseudo-random 64-bit integers. This hiding is **encapsulation** in action; it tames complexity and makes the type easy to use and reason about. You'll see encapsulation used throughout this book and discussed further in Chapter 14, "API Design Tips & Tricks".

To use this pseudo-random source, create an overload of `Point.random`. Add this to your playground:

```
extension Point {
  static func random(inRadius radius: Double,
                     using randomSource:
                       inout PermutedCongruential) -> Point {
    guard radius >= 0 else {
      return .zero
    }

    let x = Double.random(in: -radius...radius,
                          using: &randomSource)
    let maxY = (radius * radius - x * x).squareRoot()
    let y = Double.random(in: -maxY...maxY,
                          using: &randomSource)
    return Point(x: x, y: y)
  }
}
```

It's quite like the previous version that uses the system random number generator. As a static method, `random(in:using:)` also doesn't touch an instance of `Point`. But notice how mutable state can flow through the function because `randomSource` is an `inout` parameter. This way to handle a side-effect via parameters is a much better design than, say, using a global variable to track the pseudo-random state. It explicitly surfaces the side-effect to the user, allowing it to be controlled.

> **Note**: This random function is unfortunately specific to the concrete type `PermutedCongruential`. In Chapter 4, "Generics", you'll see the techniques for working with any type conforming to `RandomNumberGenerator`, including `SystemRandomNumberGenerator()`. If you want to see this function written generically and without logic duplication, check out the playground **RandomPointGeneric** in this chapter's final resources folder.

Test deterministic random numbers with this code in your playground:

```
var pcg = PermutedCongruential(seed: 1234)
for _ in 1...10 {
  print(Point.random(inRadius: 1, using: &pcg))
}
```

These look like random numbers but are reproducible. The tenth random point will always be `Point(x: 0.43091531644250813, y: 0.3236366519677818)` given a starting seed of `1234`.

Enumerations

Swift enumerations are another powerful value type that lets you model a finite set of states. Add this to your playground:

```
enum Quadrant: CaseIterable, Hashable {
  case i, ii, iii, iv

  init?(_ point: Point) {
    guard !point.x.isZero && !point.y.isZero else {
      return nil
    }

    switch (point.x.sign, point.y.sign) {
    case (.plus, .plus):
      self = .i
    case (.minus, .plus):
      self = .ii
    case (.minus, .minus):
      self = .iii
    case (.plus, .minus):
      self = .iv
    }
  }
}
```

COPY

This code creates an abstraction for quadrants in the two-dimensional plane. The `CaseIterable` conformance lets you access an array, `allCases`. `Hashable` means you can use it as an element of a `Set` or key of as the key of a `Dictionary`. You can make the initializer failable because points on the x- or y-axis aren't defined to be in a quadrant. An optional initializer lets you document this possibility naturally.

Try it out with this:

```
Quadrant(Point(x: 10, y: -3)) // evaluates to .iv
Quadrant(.zero) // evaluates to nil
```

COPY

Types as documentation

Types can serve as documentation. For example, if you have a function that returns an `Int`, you don't need to worry if the function will return 3.14159 or "Giraffe". It just can't happen. In a sense, the compiler rules out all those crazy possibilities.

Historical note: One of the more famous software engineering failures came in 1999 with the Mars Climate Orbiter. Engineers at the Jet Propulsion Lab in California wrote their functions with metric impulse values measured in newton-seconds. In contrast, engineers at Lockheed Martin Astronautics in Colorado wrote their functions with English units of pound-seconds. Imagine

if the two groups had made units explicit with a type. Doing so might have prevented the costly ($125M+) error that caused the space probe to skip off (or burn up in) the Mars atmosphere.

Foundation has an extensible set of types for dealing with common physical units, such as length, temperature and angle. Consider angles, which can be expressed in a variety of units. Add this to the playground:

```
let a = Measurement(value: .pi/2,
                    unit: UnitAngle.radians)

let b = Measurement(value: 90,
                    unit: UnitAngle.degrees)

a + b  // 180 degrees
```

The variable `a` is a right angle expressed in radians, and `b` is a right angle expressed in degrees. You can add them together to see that they're 180 degrees. The `+` operator converts them to a base unit before adding the values.

Of course, Swift lets you define overloads of standard math functions. You can make a type-safe version of `cos()` and `sin()`.

```
func cos(_ angle: Measurement<UnitAngle>) -> Double {
  cos(angle.converted(to: .radians).value)
}

func sin(_ angle: Measurement<UnitAngle>) -> Double {
  sin(angle.converted(to: .radians).value)
}

cos(a)  // 0
cos(b)  // 0
sin(a)  // 1
sin(b)  // 1
```

The function takes an angle and converts it explicitly to radians before passing it to the standard transcendental `cos()` and `sin()` functions. With this new API, the compiler can check to make sure you're passing angle types instead of something nonsensical.

Note: Several popular frameworks take care of angle types. In addition to the Foundation's `Measurement` type, SwiftUI also defines an `Angle` that explicitly initializes with degrees or radians. A generic version that abstracts across all the different floating-point types is proposed for the official Swift numerics package.

Improving type ergonomics

One of the great things about Swift is how you can extend the functionality and interoperability of existing types that you don't even have the source for. For example, suppose you wanted your program to deal with polar coordinates. You'll use angles a lot, so add this:

```
typealias Angle = Measurement<UnitAngle>

extension Angle {
  init(radians: Double) {
    self = Angle(value: radians, unit: .radians)
  }
  init(degrees: Double) {
    self = Angle(value: degrees, unit: .degrees)
  }
  var radians: Double {
    converted(to: .radians).value
  }
  var degrees: Double {
    converted(to: .degrees).value
  }
}
```

`typealias` gives you a shorter, descriptive spelling for angles. You can now go back and improve your `sin` and `cos` implementations like this:

```
func cos(_ angle: Angle) -> Double {
  cos(angle.radians)
}
func sin(_ angle: Angle) -> Double {
  sin(angle.radians)
}
```

You'll probably agree that those look much nicer. Now, you can define a polar coordinate type:

```
struct Polar: Equatable {
  var angle: Angle
  var distance: Double
}
```

Because you'll want to flip between xy coordinates and polar coordinates easily, you can add type converting initializers for those:

```
// Convert polar-coordinates to xy-coordinates
extension Point {
  init(_ polar: Polar) {
    self.init(x: polar.distance * cos(polar.angle),
              y: polar.distance * sin(polar.angle))
```

```
  }
}

// Convert xy-coordinates to polar coordinates
extension Polar {
  init(_ point: Point) {
    self.init(angle: Angle(radians: atan2(point.y, point.x)),
              distance: hypot(point.x, point.y))
  }
}
```

Notice how your abstractions build on one another, making an even more powerful environment to work with. Your types are letting you hide complexity layer-by-layer.

Now, you can easily go from xy coordinates to polar coordinates and vice-versa like this:

```
let coord = Point(x: 4, y: 3)
Polar(coord).angle.degrees // 36.87
Polar(coord).distance      // 5
```

Strong types mean you can't accidentally mix up polar coordinates and xy coordinates, but you can still easily switch between the two when that is what you intend.

Associated values

Enumerations in Swift are quite powerful because they let you associate information with a particular case. For example, you can create a fixed set of shapes:

```
enum Shape {
  case point(Point)
  case segment(start: Point, end: Point)
  case circle(center: Point, radius: Double)
  case rectangle(Rectangle)
}
```

As you can see, it's easy to compose types. You can succinctly model the valid states in your app and even prevent invalid states from being representable and thus compiling.

Using `RawRepresentable`

There's another essential tool for your tool chest. You have probably used `RawRepresentable` for enumerations without realizing it. Open the starter playground **RawRepresentable** and add the following:

```
enum Coin {
  case penny, nickel, dime, quarter
}
```

When you back the enumeration with an integer, character or string, it becomes `RawRepresentable` thanks to compiler magic. Replace the previous definition with this:

```
enum Coin: Int {
  case penny = 1, nickel = 5, dime = 10, quarter = 25
}
```

Being `RawRepresentable` means you can create and get the raw value. It also means that the type is `Equatable`, `Hashable` and `Codable` for free.

```
let lucky = Coin(rawValue: 1)
lucky?.rawValue  // returns Optional(1)
let notSoMuch = Coin(rawValue: 2)
```

You can use `RawRepresentable` directly to create simple checked types. Consider this example:

```
struct Email: RawRepresentable {
  var rawValue: String

  init?(rawValue: String) {
    guard rawValue.contains("@") else {
      return nil
    }
    self.rawValue = rawValue
  }
}
```

This simple type provides a form of documentation. Consider the signature of a function that uses it:

```
func send(message: String, to recipient: Email) throws {
  // some implementation
}
```

This function's easier to use because the parameter labels make it clear where `message` and `recipient` go and hard to misuse because of the specific types that the compiler can check. The type for `Email` means that it's only possible to pass around well-formed email addresses. (For this example, the check simply looks to make sure there is an `@` in the address, but you can make it arbitrarily strict.)

Rather than having a property like `isValid`, it's better if you can make your custom type's initializer failable either by returning nil or throwing a more specific error when a valid instance can't be created. This explicit failure mode allows you to set up your code so the compiler forces you to check for errors. The reward is this: When you write a function that uses a type, you don't have to worry about half-baked instances that might not be valid. This pattern pushes data validation and error handling to your software stack's upper layers and lets the lower levels run efficiently without extra checks.

Exercises

Here are a few quick exercises to check your understanding. You can use the starter playground **Exercises** to get you started. It imports the types you've made so far. As always, it's best to give it an honest try before looking at the answers in the final version.

1. Generate 100 random points in the unit circle. How many does the second quadrant contain? Demonstrate the solution with code. Use `PermutedCongruential` with the seed 4321 to make your answer repeatable.
2. How many cups are in 1.5 liters? Use Foundation's `Measurement` types to figure it out.
3. Create an initializer for `Quadrant` that takes a polar coordinate.

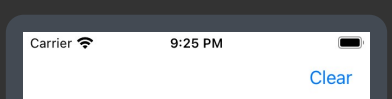
QuadTree

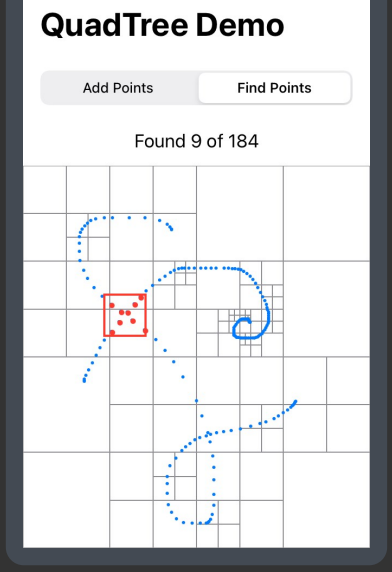
Now, you'll get a little more experience with types by building a `QuadTree` type for `CGPoints`. This example will show you how to implement mutable value semantics when using class types for storage.

A `QuadTree` is an efficient tree data structure for finding points in a region. Instead of requiring **O(n)** to find points in a matching region, it only takes **O(log n)**. It does this by putting the points in nodes (or buckets). When a node reaches maximum capacity and overflows, it creates new child nodes that split the space into four equal parts. When it comes time to find points, you can binary search these nodes efficiently.

The final demo app will let you add some points and then find points in a region by dragging your finger around.

It will look like this:

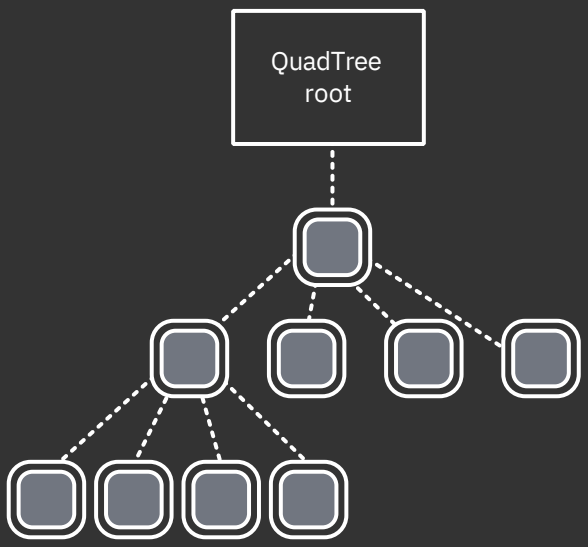




QuadTree demo app

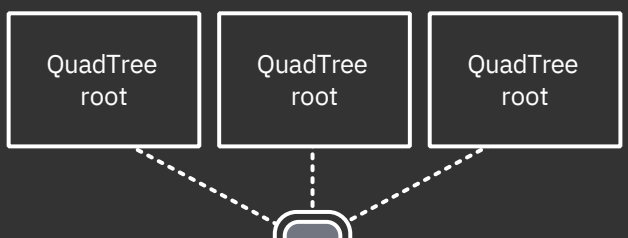
The dots are points in `QuadTree`, and the rectangles show how the tree partitions the space. The heavier square and larger dots are a set of found points that you can move around.

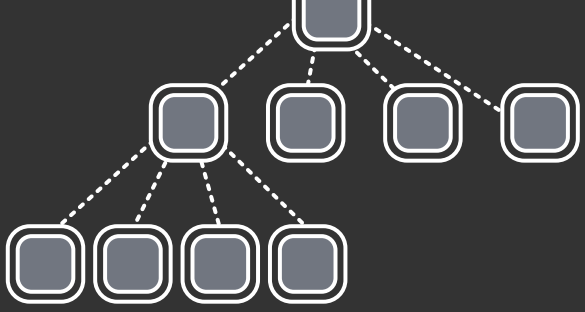
When drawn as a traditional tree, the `QuadTree` data structure looks something like this:



Single instance of a QuadTree

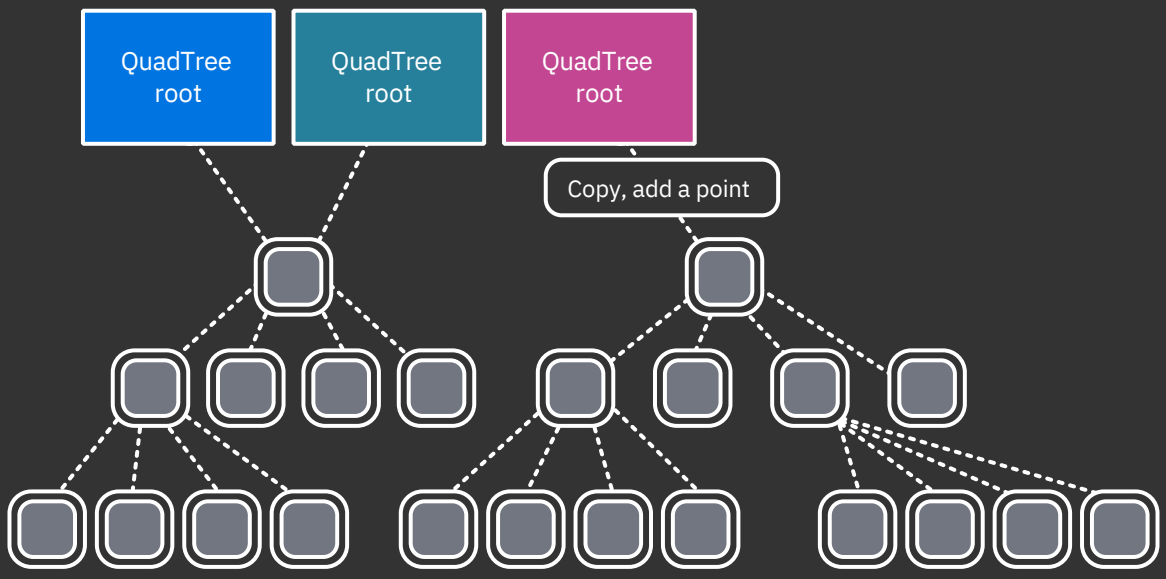
Each node has either zero or four children. The `QuadTree` type itself is a lightweight value type. It has a private property `root` to a reference type `Node` at the top of the tree. It's cheap to copy a `QuadTree` because copies share the `Node` reference. Graphically, sharing looks something like this:





Three QuadTree instances sharing the same node storage

To keep nice value semantics when one of the shared instances decides to modify the tree by adding a point, it must make a deep copy of the tree and add the point. This process is called **copy-on-write** or sometimes **CoW** for short. Graphically, you end up with something that looks like this:



Copy-on-write storage

In this case, you make a copy before adding the new point. The node overflows its capacity and subdivides itself into four new sub-nodes.

Implementing `QuadTree`

Begin the implementation of `QuadTree` by opening the **QuadTree** Xcode starter project. Use the file navigator to familiarize yourself with the files in the project. Skim through these five files without worrying too much about the details.

- **AppMain.swift**: Contains the basic definition of the SwiftUI app.
- **ContentView.swift**: The top-level user interface. It defines the picker for adding or finding points, a place to show the number of points found and a clear button. It also contains the drag gesture for inserting and finding points.

- **QuadTree.swift**: Stubs out the definition of `QuadTree`. This is where you'll do your work.
- **QuadTreeView.swift**: The canvas where points and rectangles are drawn. It finds the size of the view and reports it to the view-model so points can be stored in normalized coordinates ranging from zero to one.
- **QuadTreeViewModel.swift**: Connects the model (your `QuadTree` instance) to the user interface. This file contains the so-called business logic of your app.

You can build and run the app at this point, but you won't yet be able to insert and find points. To make that happen, you need to fill out the `QuadTree` type.

Open **QuadTree.swift**, which contains a skeleton definition.

Inside the `QuadTree` definition, add the private nested class `Node`:

```
private final class Node {
  let maxItemCapacity = 4
  var region: CGRect
  var points: [CGPoint] = []
  var quad: Quad?

  init(region: CGRect, points: [CGPoint] = [], quad: Quad? = nil) {
    self.region = region
    self.quad = quad
    self.points = points
    self.points.reserveCapacity(maxItemCapacity)
    precondition(points.count <= maxItemCapacity)
  }

  struct Quad {
    // more to come...
  }
}
```

The nested class `Node` will do the heavy lifting for `QuadTree`. Each node instance keeps a `region` and can hold up to four points (the bucket size) before it spills over and subdivides itself into four more nodes contained in `Quad`'s structure.

> **Note**: The bucket size is set to four so you can easily visualize what is happening. An actual implementation would probably have a much higher bucket size based on an analysis of insertion and find times.

Next, add this to the definition of `Quad` inside `Node`:

```
var northWest: Node
var northEast: Node
```

```
var southWest: Node
var southEast: Node

var all: [Node] { [northWest, northEast, southWest, southEast] }

init(region: CGRect) {
  let halfWidth = region.size.width * 0.5
  let halfHeight = region.size.height * 0.5

   northWest =
     Node(region: CGRect(x: region.origin.x,
                         y: region.origin.y,
                         width: halfWidth, height: halfHeight))
   northEast =
     Node(region: CGRect(x: region.origin.x + halfWidth,
                         y: region.origin.y,
                         width: halfWidth, height: halfHeight))
   southWest =
     Node(region: CGRect(x: region.origin.x, y:
                         region.origin.y + halfHeight,
                         width: halfWidth, height: halfHeight))
   southEast =
     Node(region: CGRect(x: region.origin.x + halfWidth,
                         y: region.origin.y + halfHeight,
                         width: halfWidth, height: halfHeight))
}

// more to come...
```

This code defines the four sub-nodes of a `Quad`. The initializer is verbose, but all it's doing is dividing the parent region into four equal sub-regions.

You need to be able to make a deep copy `Node`, so add this initializer and copy method to `Quad`:

```
init(northWest: Node, northEast: Node,
     southWest: Node, southEast: Node) {
  self.northWest = northWest
  self.northEast = northEast
  self.southWest = southWest
  self.southEast = southEast
}

func copy() -> Quad {
  Quad(northWest: northWest.copy(),
       northEast: northEast.copy(),
       southWest: southWest.copy(),
       southEast: southEast.copy())
}
```

That completes the definition of `Quad`.

Copying

The code above expects a `copy()` method on `Node`, so add that now to the

body of `Node`:

```
func copy() -> Node {
  Node(region: region, points: points, quad: quad?.copy())
}
```

This function will resolve the four compiler errors. Somewhat amazingly, adding this allows `Node` and `Quad` to recursively copy themselves from a root node, all the way down the tree.

Next, add a helper method to subdivide a `Node` by adding this to the `Node` definition:

```
func subdivide() {
  precondition(quad == nil, "Can't subdivide a node already
subdivided")
  quad = Quad(region: region)
}
```

All this does is assign `quad` to an instance. The initializer that you wrote above does the real work of rectangle division. The precondition makes sure you aren't subdividing a node that has already been subdivided. It's always a good idea to check your assumptions if it's computationally cheap.

Next, write `insert` for `Node`. Add this:

```
@discardableResult
func insert(_ point: CGPoint) -> Bool {
  // 1
  guard region.contains(point) else {
    return false
  }
  // 2
  if let quad = quad {
    return quad.northWest.insert(point) ||
      quad.northEast.insert(point) ||
      quad.southWest.insert(point) ||
      quad.southEast.insert(point)
  }
  else {
    // 3
    if points.count == maxItemCapacity {
      subdivide()
      return insert(point)
    }
    else {
      points.append(point)
      return true
    }
  }
}
```

This function returns a `Bool` that is `true` if it inserts the point and `false` if it doesn't.

Here is a rundown of what the function does:

1. First, checks if the point is in the region at all. If it's not, exit early returning `false`.
2. Checks whether the node has been subdivided. If it has, the function attempts to insert the point into each one of the nodes in the quad. The logical or `||` short-circuits and stops inserting as soon as it does.
3. If the node is at maximum capacity, it subdivides the node and attempts the insert again. Otherwise, just adds the point and returns `true`.

The last method you'll define for `Node` is for finding points in a region. Add this method to the `Node` type:

```
func find(in searchRegion: CGRect) -> [CGPoint] {
  guard region.intersects(searchRegion) else {
    return []
  }
  var found = points.filter { searchRegion.contains($0) }
  if let quad = quad {
    found += quad.all.flatMap { $0.find(in: searchRegion) }
  }
  return found
}
```

COPY

This code first checks whether the search region overlaps with the region the node is responsible for. If it doesn't, it returns no points. This return is the base case of the recursion. Next, it filters the points in the region and adds them to the `found` list. Finally, if the node has been subdivided, it goes through the quads and recursively calls `find(in:)` to add points to the `found` list before returning it.

Implementing `QuadTree` methods

Now that you have completed the `Node` type, you can implement `QuadTree`'s methods. First, add a private property to `QuadTree` right above `count`:

```
private var root: Node
```

COPY

The initializer of `QuadTree` specifies a region of space it handles. Replace the stubbed out implementation with this:

```
init(region: CGRect) {
  root = Node(region: region)
```

COPY

```
}
```

Next, replace `find(in:)` and `points()` in `QuadTree` with these:

```
func find(in searchRegion: CGRect) -> [CGPoint] {
  root.find(in: searchRegion)
}

func points() -> [CGPoint] {
  find(in: root.region)
}
```

`find(in:)` simply delegates to the root node. `points()` gathers all the points by finding them from the root node down.

Next, replace the placeholder for `regions()` that returns the region each `Node` is responsible for:

```
private func collectRegions(from node: Node) -> [CGRect] {
  var results = [node.region]
  if let quad = node.quad {
    results += quad.all.flatMap { collectRegions(from: $0) }
  }
  return results
}

func regions() -> [CGRect] {
  collectRegions(from: root)
}
```

`regions()` calls the private helper method `collectRegions(from:)`, which recursively gathers all the regions of all the nodes.

Finally, replace an insert method with this implementation:

```
@discardableResult
mutating func insert(_ point: CGPoint) -> Bool {
  if !isKnownUniquelyReferenced(&root) {
    root = root.copy()
  }
  guard root.insert(point) else {
    return false
  }
  count += 1
  return true
}
```

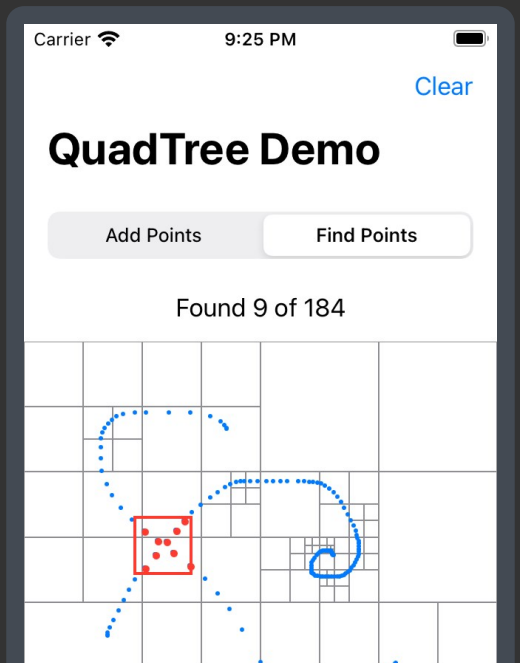
This function is marked with `@discardableResult` because clients might not wish to check if the insertion succeeded. The only way it can fail is if you try to insert a point that is outside the region specified by the `QuadTree` initializer.

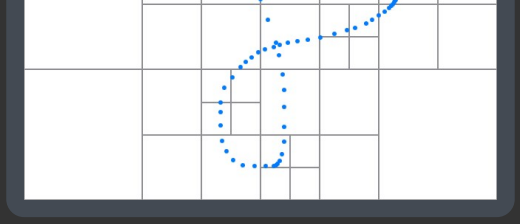
This code is where the **copy-on-write** magic happens. Swift has a special function, `isKnownUniquelyReferenced()`, which returns true if there is only one instance. If there is more than one instance, you need to make a copy of the underlying tree. Then, you can add the point.

The property `count` exists to make it cheap **O(1)** to know `QuadTree`'s point count without recursively traversing all the nodes. It only gets incremented if the insertion succeeds. The success of insertion depends on the original size of the region `QuadTree` was initialized with.

> **Note**: To maintain value semantics on mutation, you **must make a deep copy** of the underlying storage for every mutating method. It's an *optimization* not to copy with the instance if it's not shared. This optimization can be prevented if extra copies are lying around. The SwiftUI framework has special property wrappers `@State` and `@Published` used to manage UI updates. Unfortunately, these wrappers make an extra copy that interferes with the `isKnownUniquelyReferenced` optimization. If you look closely at **QuadTreeViewModel.swift**, you'll see `quadTree` is not a `@Published` property but instead calls `objectWillChange.send()` directly to handle UI update. Doing this prevents extra copying from happening, which will slow the user interface after a few hundred points are added.

Build and run the app. Drag your finger around to add some points. If you tap to add points one by one, you'll see that a region subdivides itself on the fifth tap because the node overflows maximum capacity. But finding points in a region is where `QuadTree` shines. Rather than traversing the entire point list, it can focus on a particular region quickly. You can try it out by switching to find mode and dragging the little find box around. Finding points in a region quickly is why `QuadTree` is used in collision detection and compression applications.





QuadTree demo app

Key points

Swift is a strongly typed language that allows the compiler to check your program's correctness before it runs. The better you get at working with types, the easier it will be to write correct programs.

Here are some key points to keep in mind from this chapter:

- Structures, enumerations and classes are the fundamental named types that Swift uses to make every other concrete type, including `Bool`, `Int`, `Optional`, `String`, `Array`, etc.
- Create custom types to solve problems in your domain elegantly.
- Structures and enumerations are value types. Classes are reference types.
- Any type can be designed to have value semantics.
- The most straightforward way to get value semantics is to use a value type (structure or enumeration) that only contains other types with value semantics.
- All the standard types, including `String`, `Array`, `Set` and `Dictionary`, already have value semantics, making them easy to compose into larger types with value semantics.
- Making a class immutable is one way to give reference types value semantics.
- Value types are generally copied around on the stack, whereas reference types are allocated on the heap and are reference counted.
- Reference types have a built-in notion of lifetime and identity.
- Instance methods secretly pass in self.
- The mutating instance methods of value types pass `inout self`.
- Enumerations model a finite set of states.
- Avoid initializing half-baked, invalid objects. Instead, create failing initializers.
- A good set of types can act as compiler-checkable documentation.
- The foundation `Measurement` types make working with different units less error-prone by defining them as concrete types.

- Swift lets you improve the usage ergonomics for types you don't even own.
- The protocol `RawRepresentable` lets you create simple, expressive types.
- Copy-on-write is a way to give reference types mutating value semantics.

Mark Complete

3. Protocols →

← 1. Introduction

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.

Have feedback to share about the online reading experience? If you have feedback about the UI, UX, highlighting, or other features of our online readers, you can send them to the design team with the form below:

Feedback about the UI, UX, or other features of the online reader? Leave them here!

Send Feedback