⚠ This is a pre-release version. Many things are missing. ⚠

Thanks for buying the pre-release book. We're very happy that you chose to support us. We're planning to publish the first content early July. We'll be updating this PDF on a regular basis, but for the latest available content, check out https://github.com/objcio/fpinswift-beta

We would love to hear from you. If there are any topics you'd like to see included, or if there's anything that deserves a better explanation, let us know. File an issue on GitHub, or send an email to mail@objc.io

Florian, Wouter and Chris

objc ↑↓ Functional Programming in Swift

By Chris Eidhof, Florian Kugler and Wouter Swierstra

Introduction

Why write this book? There is plenty of documentation on Swift readily available from Apple, and there are many more books on the way. Why does the world need yet another book on yet another programming language?

This book tries to teach you to think *functionally*. We believe that Swift has the proper language features to teach you how to write *functional programs*. But what makes a program functional? Or why bother learning about this in the first place? In his paper, "Why Functional Programming Matters," John Hughes writes:

> Functional programming is so called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result.

So rather than thinking of a program as a sequence of assignments and method calls, functional programmers emphasize that each program can be repeatedly broken into smaller and smaller pieces. By avoiding assignment statements and side effects, Hughes argues that functional programs are more modular than their imperative- or object-oriented counterparts. And modular code is a Very Good Thing.

In our experience, learning to think functionally is not an easy thing. It challenges the way we've been trained to decompose problems. For programmers who are used to writing for-loops, recursion can be confusing; the lack of assignment statements and global state is crippling; and closures, generics, higher-order functions, and monads are just plain weird.

In this book, we want to demystify functional programming and dispel some of the prejudices people may have against it. You don't need to have a PhD in mathematics to use these ideas to improve your code! We won't cover the complete Swift language specification or teach you to set up your first project in Xcode. But we will try to teach you the basic principles of pure functional programming that will make you a better developer in any language.

Thinking Functionally

⚠ this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes. ⚠

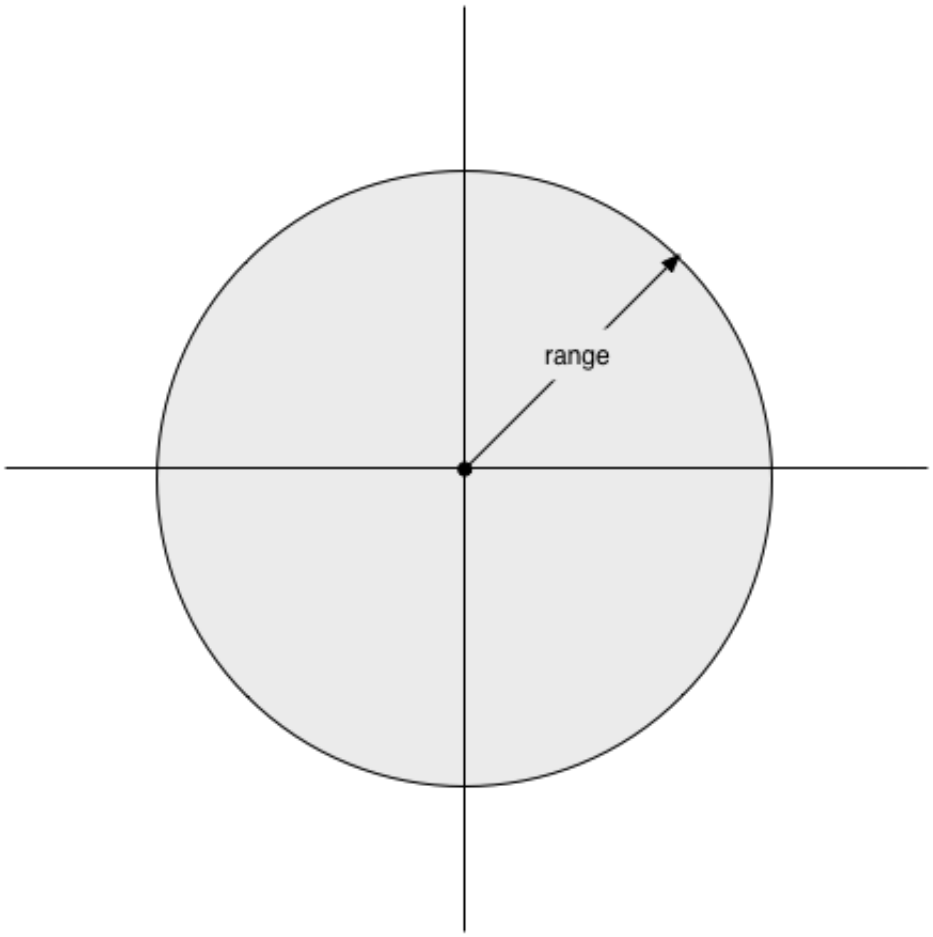
What this chapter is about

Functions in Swift are *first-class values*, i.e., functions may be passed as arguments to other functions. This idea may seem strange if you're used to working with simple types, such as integers, booleans or structs. In this chapter, we will try to motivate why first-class functions are useful and give a first example of functional programming in action.

Example: Battleship

We'll introduce first-class functions using the example of an algorithm you would implement if you would want to create a Battleship-like game.[1] The problem we'll look at boils down to determining whether or not a given point is in range, without being too close to friendly ships or ourselves.

As a first approximation, you might write a very simple function that checks whether or not a point is in range. For the sake of simplicity, we will assume that our ship is located at the origin. We can visualize the region we want to describe as follows:



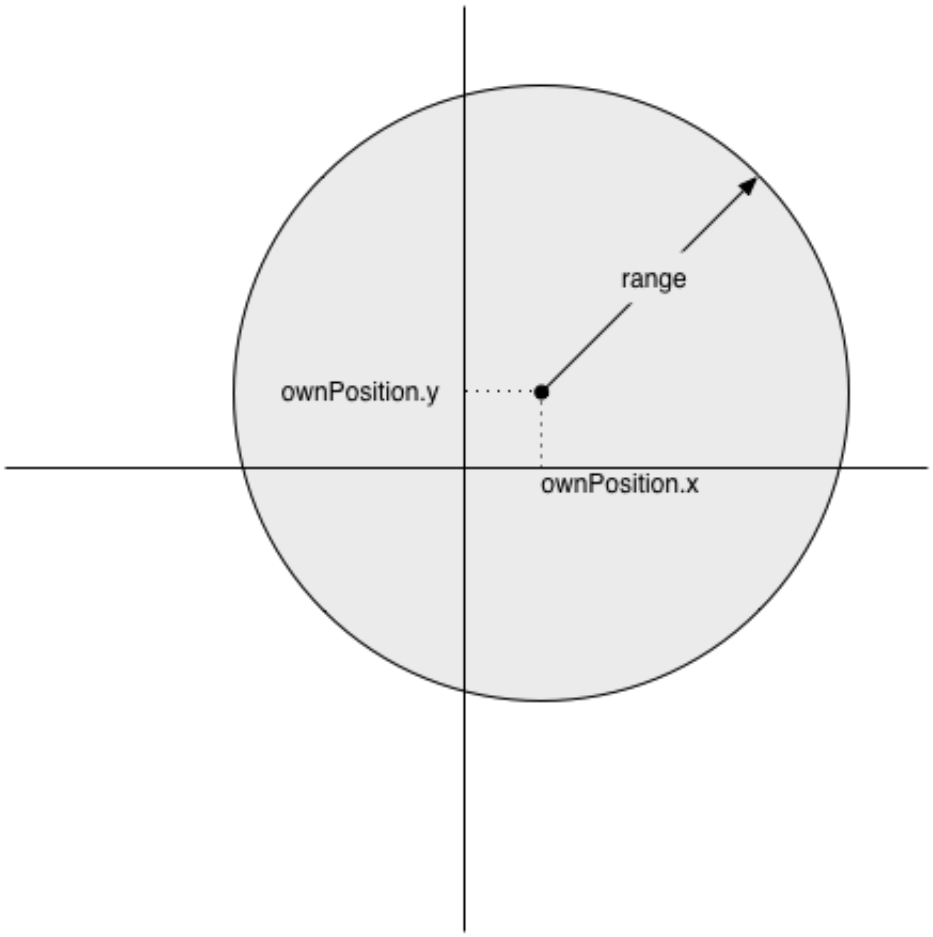
The first function we write, `inRange1` checks when a point is in the grey area in the picture above. Using some basic geometry, we can write this function as follows:

```
typealias Position = CGPoint

func inRange1(target: Position, range: Double) -> Bool {
   return sqrt(target.x * target.x + target.y * target.y) <= range
}
```

Note that we are using Swift's typealias construct let us introduce a new name for an existing type. From now on, whenever we write `Position`, feel free to read `CGPoint`, a pair of an `x` and `y` coordinate.

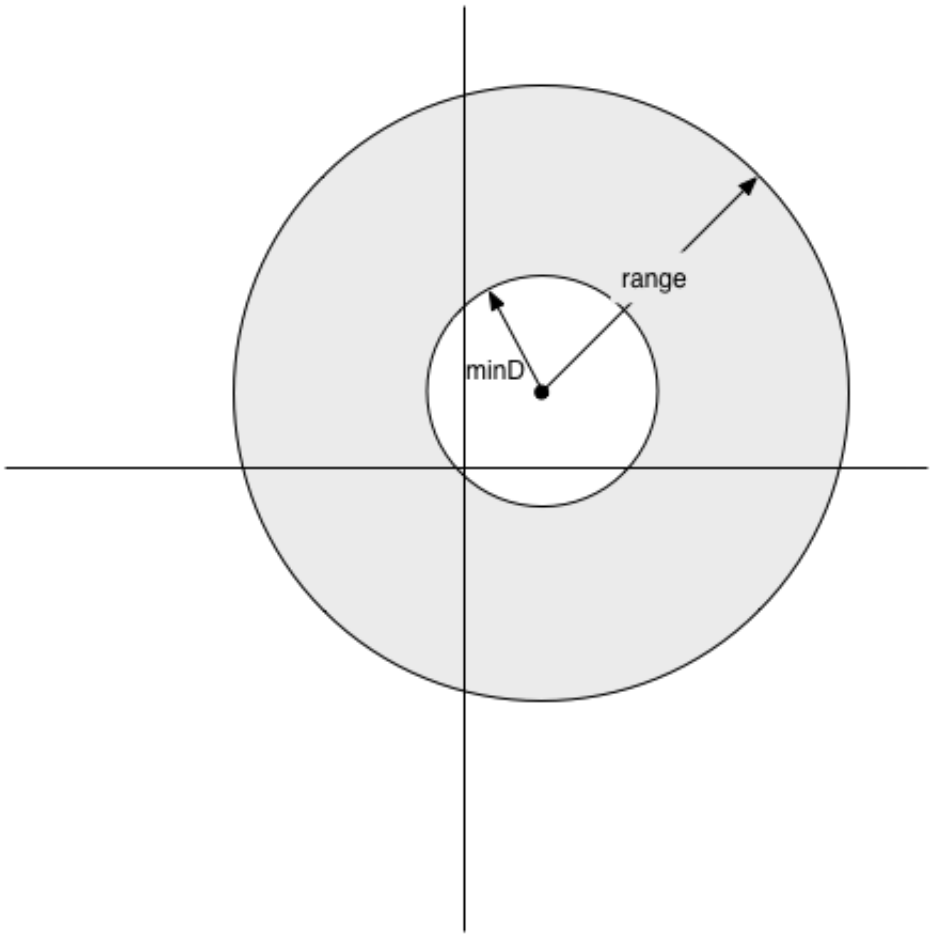
Now this works fine, if you assume that the we are always located at the origin. Suppose the ship may be at a location, `ownposition`, other than the origin. We can update our visualization to look something like this:



We now add an argument representing the location of the ship to our `inRange` function:

```
func inRange2(target: Position, ownPosition: Position, range: Double) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
}
```

But now you realize that you also want to avoid targeting ships if they are too close to yourself. We can update our picture to illustrate the new situation, where we want to target only those enemies that are at least `minD` away from our current position:

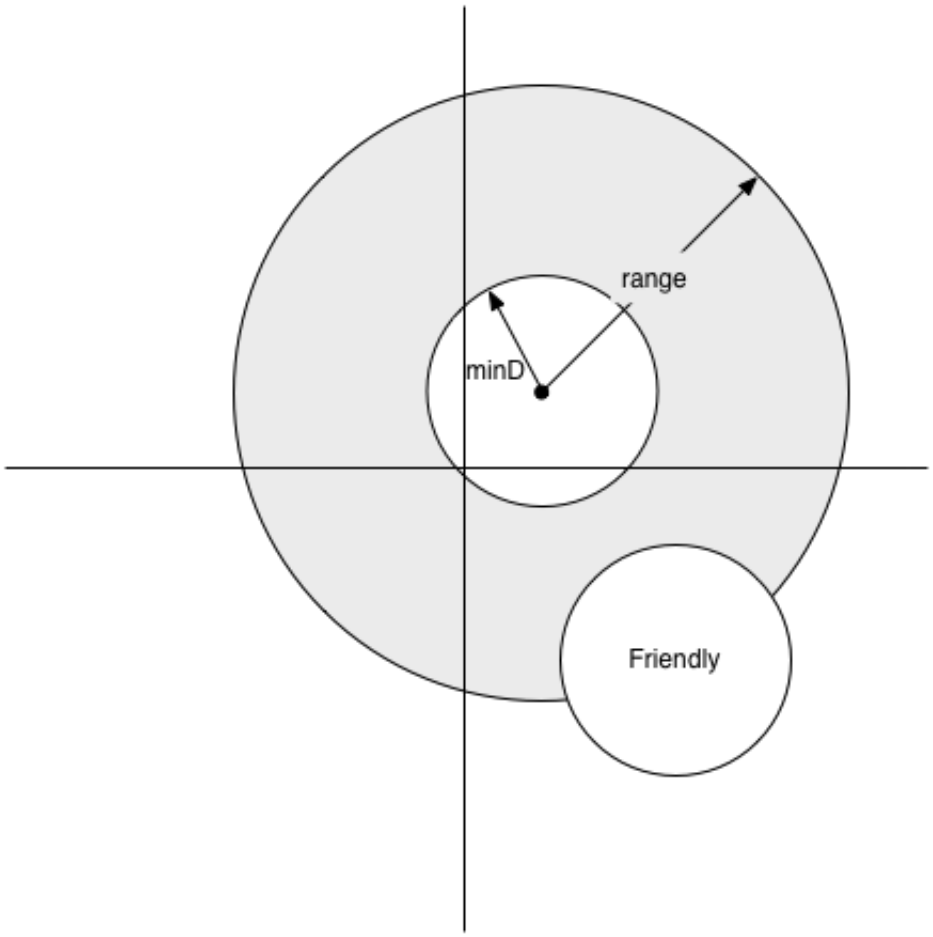


As a result, we need to modify our code again:

```
let minimumDistance = 2.0

func inRange3(target: Position, ownPosition: Position, range: Double) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
          && sqrt(dx * dx + dy * dy) >= minimumDistance
}
```

Finally, you also need to avoid targeting ships that are too close to one of your other ships. We can visualize this by as follows:



Correspondingly, we can add a further argument that represents the location of a friendly ship to our `inRange` function:

```
func inRange4(target: Position, ownPosition: Position, friendly: Position, range:
Double) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   let friendlyDx = friendly.x - target.x
   let friendlyDy = friendly.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
          && sqrt(dx * dx + dy * dy) >= minimumDistance
          && !(sqrt(friendlyDx * friendlyDx + friendlyDy * friendlyDy) >= minimumD
istance)
}
```

As this code evolves, it becomes harder and harder to maintain. This method expresses a complicated calculation in one big lump of code. Let's try to refactor this into smaller, compositional pieces.

First-class functions

There are different approaches to refactoring this code. One obvious pattern would be to introduce a function that computes the distance between two points; or functions that check when two points are 'close' or 'far away' (for some definition of close and far). In this chapter, however, we'll take a slightly different approach.

The original problem boiled down to defining a function that determined when a point was in range or not. The type of such a function would be something like:

```
func pointInRange(point: Position) -> Bool {
    // Implement method here
}
```

The type of this function is going to be so important, that we're going to give it a separate name:

```
typealias Region = (Position) -> Bool
```

From now on, the `Region` type will refer to functions from a `Position` to a `Bool`. This isn't strictly necessary, but it can make some of the type signatures that we'll see below a bit easier to digest.

Instead of defining an object or struct to represent regions, we represent a region by a *function* that determines if a given point is in the region or not. If you're not used to functional programming this may seem strange, but remember: functions in Swift are first-class values! We conciously choose the name `Region` for this type rather than something like `CheckInRegion` or `RegionBlock`. These names suggest that they denote a function type; yet the key philosophy underlying *functional programming* is that functions are values, no different from structs, Ints or Bools – using a separate naming convention for functions would violate this philosophy.

We will now write several functions that create, manipulate and combine regions.

The first region we define is a `circle`, centered around the origin:

```
func circle(radius: Double) -> Region {
    return { point in sqrt(point.x * point.x + point.y * point.y) <= radius }
}
```

Note that, given a radius `r`, the call `circle(r)` *returns a function*. Here we use Swift's notation for closures to construct the function that we wish to return. Given an argument position, `point`, we check that the `point` is in the region delimited by a circle of the given radius centered around the origin.

Of course, not all circles are centered around the origin. We could add more arguments to the `circle` function to account for this. Instead, though, we will write a *region transformer:*

```
func shift(offset: Position, region: Region) -> Region {
    return { point in
        let shiftedPoint = Position(x: point.x - offset.x, y: point.y - offset.y)
        return region(shiftedPoint)
    }
}
```

The call `shift(offset, region)` moves the region to the right and up by `offet.x` and `offset.y` respectively. How is it implemented? Well we need to return a `Region`, that is a function from points to `Bool`. To do this, we start writing another closure, introducing the point we need to check. From this point, we compute a new point with the coordinates `point.x - offset.x` and `point.y - offset.y`. Finally, we check that this new point is in the *original* region by passing it as an arguments to the `region` function.

Interestingly, there are lots of other ways to transform existing regions. For instance, we may want to define a new region by inverting a region. The resulting region consists of all the points outside the original region:

```
func invert(region: Region) -> Region {
    return { point in !region(point) }
}
```

We can also write functions that combine existing regions into larger, complex regions. For instance, these two functions take the points that are in *both* argument regions or *either* argument region, respectively:

```
func intersection(region1 : Region, region2 : Region) -> Region {
    return { point in region1(point) && region2(point) }
}

func union(region1 : Region, region2 : Region) -> Region {
    return { point in region1(point) || region2(point) }
}
```

Of course, we can use these functions to define even richer regions. The `crop` function takes two regions as argument, `region` and `minusRegion`, and constructs a region with all points that are in the first, but not in the second region.

```
func difference(region: Region, minusRegion: Region) -> Region {
    return intersection(region, invert(minusRegion))
}
```

This example shows how Swift lets you compute and pass around functions no differently than integers or booleans.

Now let's turn our attention back to our original example. With this small library in place, we can now refactor the complicated `inRange` function as follows:

```
func inRange(ownPosition: Position, target: Position, friendly: Position, range: D
ouble) -> Bool {
  let targetRegion = shift(ownPosition, difference(circle(range), circle(minimumDi
stance)))
  let friendlyRegion = shift(friendly, circle(minimumDistance))
  return difference(targetRegion, friendlyRegion)(target)
}
```

The way we've defined the `Region` type does have its disadvantages. In particular, we cannot inspect *how* a region was constructed: is it composed of smaller regions? Or is it simply a circle around the origin? The only thing we can do is to check whether or not a given point is within a region or not. If we would want to visualize a region, we would have to sample enough points to generate a (black and white) bitmap.

Type-driven development

In the introduction, we mentioned how functional programs take the application of functions to arguments as the canonical way to assemble bigger programs. In this chapter, we have seen a concrete example of this functional design methodology. We have defined a series of functions for describing regions. Each of these functions is not very powerful by itself. Yet together, they can describe complex regions that you wouldn't want to write from scratch.

The solution is simple and elegant. It is quite different from what you might write, had you just refactored the `inRange4` function into separate methods. The crucial design decision we made was *how* to define regions. Once we chose the `Region` type, all the other definitions followed naturally. The moral of the example is **choose your types carefully**. More than anything else, types guide the development process.

```
        result.append(f(x))
    }
    return result
}
```

Now we can pass different arguments, depending on how we want to compute a new array from the old array. The `doubleArray` and `incrementArray` functions become one-liners that call `computeIntArray`:

```
func doubleArray2 (xs : [Int]) -> [Int] {
  return computeIntArray(xs){x in x * 2}
}
```

Note that we are using Swift's syntax for trailing closures again here.

This code is still not as flexible as it could be. Suppose we want to compute a new array of booleans, describing whether the numbers in the original array were even or not. We might try to write something like:

```
func isEvenArray (xs : [Int]) -> [Bool] {
    computeIntArray(xs){x in x % 2 == 0}
}
```

Unfortunately, this code gives a type error. The problem is that our `computeIntArray` function takes an argument of type `Int -> Int`, function that computes a new integer. In the definition of `isEvenArray` we are passing an argument of type `Int -> Bool` which causes the type error.

How should we solve this? One thing we *could* do is define a new version of `computeIntArray` that expects takes a function argument of type `Int -> Bool`. This might look something like this:

```
func computeBoolArray (xs : [Int], f : Int -> Bool) -> [Bool] {
    let result : [Bool] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
```

```
    {
        result.append(x + 1)
    }
    return result
}
```

Now suppose we also need a function that computes a new array, where every element in the argument array has been doubled. This is also easy to do using a `for` loop:

```
func doubleArray1 (xs : [Int]) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(x * 2)
    }
    return result
}
```

Both these functions share a lot of code. Can we abstract over the differences and write a single, more general function that captures this pattern? Such a function would look something like this:

```
func computeIntArray (xs : [Int]) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(... \\ something using x)
    }
    return result
}
```

To complete this definition, we need to add a new argument describing

To complete this definition, we need to add a new argument describing how to compute a new integer from `xs[i]` – that is, we need to pass a function as an argument!

```
func computeIntArray (xs : [Int], f : Int -> Int) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

Now we can pass different arguments, depending on how we want to compute a new array from the old array. The `doubleArray` and `incrementArray` functions become one-liners that call `computeIntArray`:

```
func doubleArray2 (xs : [Int]) -> [Int] {
  return computeIntArray(xs){x in x * 2}
}
```

Note that we are using Swift's syntax for trailing closures again here.

This code is still not as flexible as it could be. Suppose we want to compute a new array of booleans, describing whether the numbers in the original array were even or not. We might try to write something like:

```
func isEvenArray (xs : [Int]) -> [Bool] {
    computeIntArray(xs){x in x % 2 == 0}
}
```

Unfortunately, this code gives a type error. The problem is that our `computeIntArray` function takes an argument of type `Int -> Int`, function that computes a new integer. In the definition of `isEvenArray` we are

that computes a new integer. In the definition of `isEvenArray` we are passing an argument of type `Int -> Bool` which causes the type error.

How should we solve this? One thing we *could* do is define a new version of `computeIntArray` that expects takes a function argument of type `Int -> Bool`. This might look something like this:

```
func computeBoolArray (xs : [Int], f : Int -> Bool) -> [Bool] {
    let result : [Bool] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

This doesn't scale very well though. What if we need to compute a `String` next? Do we need to define yet another higher-order function, expecting an argument of type `Int -> String`?

Luckily there is a solution to this problem: use *generics*. The definitions of `computeBoolArray` and `computeIntArray` are identical; the only difference is in the *type signature*. If we were to define another version, `computeStringArray`, the body of the fuction would be the same again. In fact, the same code will work for *any* type. What we'd really want to do is write a single *generic* function once and for all, that will work for every possible type.

```
func genericComputeArray<T> (xs : [Int], f : Int -> T) -> [T] {
    var result : [T] = []
    for x in xs
    {
        result.append(f(x))
    }
```

```
    return result
}
```

most interesting about this piece of code is its type signature. You may want to read `genericComputeArray<T>` as a family of functions; any choice of the *type* variable `T` determines a function that takes an array of integers and a function of type `Int -> T` as arguments, and returns an array of type `[T]`.

We can generalize this function even further. There is no reason for this function to operate exclusively on input arrays of type `[Int]`. Abstracting over this yields the type signature of

```
func map<T,U> (xs : [T], f : T -> U) -> [U] {
    var result : [U] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

Here we have written a function `map` that is generic in two dimensions: for any array of `T`s and function `f : T -> U`, it will produce a new array of `U`s. This `map` function is even more generic than the `genericComputeArray` function we have seen so far. In fact, we can define `genericComputeArray` in terms of `map`:

```
func genericComputeArray2<T> (xs : [Int], f : Int -> T) -> [T] {
       return map(xs,f)
}
```

Once again, the definition of the function is not that interesting; given

Once again, the definition of the function is not that interesting: given two arguments, `xs` and `f`, apply `map` to `(xs,f)` and return the result. The types are the most interesting thing about this definition. The `genericComputeArray` is an instance of the `map` function, it only has a more specific type.

There is already a `map` method defined in the Swift standard library in the Array class. Instead of writing `map(xs,f)` for some array `xs` and function `f`, we can call the `map` function from the Array class by writing `xs.map(f)`. Here is an example definition of the `doubleArray` function using Swift's built-in `map` function:

```
func doubleArray3 (xs : [Int]) -> [Int] {
  return xs.map{x in 2 * x}
}
```

The point of this chapter is *not* to argue that you should define `map` yourself; we do want to argue that there is no magic involved in the definition of `map` – you *could* have defined it yourself!

Filter

The `map` function is not the only function in Swift's standard `Array` library that uses generics. In this section we will introduce a few others.

Suppose we have an array containing Strings, representing the contents of a directory:

```
let exampleFiles = ["README.md", "HelloWorld.swift", "HelloSwift.swi
ft", "FlappyBird.swift"]
```

Now suppose we want an array of all the .swift files. This is easy to compute with a simple loop:

```
func getSwiftFiles(files: [String]) -> [String] {
    var result : [String] = []
    for file in files {
        if file.hasSuffix(".swift") {
            result.append(file)
        }
    }
    return result
}
```

We can now use this function to ask for the Swift files in our `exampleFiles` array:

```
getSwiftFiles(exampleFiles)

> [HelloWorld.swift, HelloSwift.swift, FlappyBird.swift]
```

Of course, we can generalize the `getSwiftFiles` function. For instance, we could pass an additional `String` argument to check against instead of hardcoding the `.swift` extension. We could then use the same function to check for `.swift` or `.md` files. But what if we want to find all the files without a file extension? Or the files starting with the string `"Hello"`?

To perform such queries, we define a general purpose `filter` function. Just as we saw previously with `map`, the `filter` function takes a *function* as an argument. This function has type `T -> Bool` – for every element of the array this function will determine whether or not it should be included in the result.

```
func filter<T> (xs : [T], check : T -> Bool) -> [T] {
    var result : [T] = []
    for x in xs {
```

```
        if check(x) {
            result.append(x)
        }
    }
    return result
}
```

It is easy to define `getSwiftFiles` in terms of `filter`. Just like `map`, the `filter` function is defined in the Array class in Swift's standard library.

Now you might wonder: is there an even more general purpose function that can be used to define *both* `map` and `filter`? In the last part of this chapter, we will answer that question.

Reduce

Once again, let's consider a few simple functions, before defining a generic function that captures the general pattern.

It is straightforward to define a function that sums all the integers in an array:

```
func sum(xs : [Int]) -> Int {
    var result : Int = 0
    for x in xs {
        result += x
    }
    return result
}
```

```
let xs = [1,2,3,4]
sum(xs)
```

```
> 10
```

A similar for-loop computes the product of all the integers in an array:

```
func product(xs : [Int]) -> Int {
    var result : Int = 1
    for x in xs {
        result = x * result
    }
    return result
}
```

Similarly, we may want to concatenate all the strings in an array:

```
func concatenate(xs : [String]) -> String {
    var result = ""
    for x in xs {
        result += x
    }
    return result
}
```

Or concatenate all the strings in an array, inserting a separate header line and newline characters after every element:

```
func prettyPrintArray (xs : [String]) -> String {
    var result = "Entries in the array xs:\n"
    for x in xs {
        result = "  " + result + x + "\n"
    }
    return result
}
```

What do all these functions have in common? They all initialize a variable, `result`, with some value. They proceed by iterating over all the elements of the input array `xs`, updating the result somehow. To define a generic function that can capture this pattern there are two pieces of information that we need to abstract over: the initial value assigned to the `result` variable; the *function* used to update the `result` in every iteration.

With this in mind, we arrive at the following definition for the `reduce` function that captures this pattern:

```
func reduce<A,R>(arr : [A], initialValue: R, combine: (R,A) -> R) ->
 R {
    var result = initialValue
    for i in arr {
        result = combine(result,i)
    }
    return result
}
```

The type of reduce is a bit hard to read at first. It is generic in two ways: for any *input array* of type `[A]` it will compute a result of type `R`. To do this, it needs an initial value of type `R` (to assign to the `result` variable) and a function `combine : (R,A) -> R` that is used to update the result variable in the body of the for loop. In some functional languages, such as OCaml and Haskell, `reduce` functions are called `fold` or `fold_right`.

We can define every function we have seen in this chapter so far using `reduce`. Here are a few examples:

```
func sumUsingReduce (xs : [Int]) -> Int {
  return reduce(xs, 0) {result, x in x + result}
  }
```

```
  }

func productUsingReduce (xs : [Int]) -> Int {
  return reduce(xs, 1) {result, x in x * result}
  }

func concatUsingReduce (xs : [String]) -> String {
  return reduce (xs, "") {result, x in x + result}
  }
```

In fact, we can even redefine `map` and `filter` using reduce:

```
func mapUsingReduce<T,U> (xs : [T], f : T -> U) -> [U] {
  return reduce (xs,[]) {result, x in result + [f(x)]}
}

func filterUsingReduce<T> (xs : [T], check : T -> Bool) -> [T]{
  return reduce(xs, []) {result, x in check(x) ? result + [x] : resu
lt}
}
```

This shows how the `reduce` function captures a very common programming pattern: iterating over an array to compute a result.

Putting it all together

To conclude this section, we will give a small example of `map`, `filter` and `reduce` in action.

Suppose we have the following `struct` definition, consisting of a city's name and population (measured in thousands-of-inhabitants):

```
struct City {
```

```
struct City {
    let name : String
    let population : Int
}
```

We can define several example cities:

```
let paris = City(name: "Paris", population: 2243)
let madrid = City(name: "Madrid", population: 3216)
let amsterdam = City(name: "Amsterdam", population: 811)
let berlin = City(name: "Berlin", population: 3397)

let cities = [paris, madrid, amsterdam, berlin]
```

Now suppose we would like to print a list of cities with at least one million inhabitants, together with their total population. We can define a helper function that scales up the inhabitants:

```
func scaleBy1000(city : City) -> City {
    return City(name: city.name, population: city.population * 1000)
}
```

Now we can use all the ingredients we have seen in this chapter to write the following statement:

```
cities.filter({city in city.population > 1000})
      .map(scale)
      .reduce("City : Population", {result, c in result + "\n" + "\(
c.name) : \(c.population)" })
```

We start by filtering out those cities that have less than one million inhabitants; we then map our `scale` function over the remaining cities; and finally, we compute a `String` with a list of city names and

populations using the `reduce` function. Here we use the `map`, `filter` and `reduce` definitions from the Array *class* in Swift's standard library. As a result, we can chain together the results of our maps and filters nicely. The `cities.filter(..)` expression computes an Array, on which we call `map`; we call `reduce` on the result of this call to obtain our final result.

QuickCheck

⚠ this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes. ⚠

In recent years, testing has become much more prevalent in Objective-C. Many popular libraries are now tested automatically with continuous integration tools. The standard framework for writing unit tests is XCTest. In addition, a lot of third-party frameworks are available (such as Specta, Kiwi and FBSnapshotTestCase), and a number of frameworks are currently being developed in Swift.

All of these frameworks follow a similar pattern: they typically consist of some fragment of code, together with an expected result. The code is then executed; its result is then compared to the expected result mentioned in the test. Different libraries test at different levels: some might test individual methods, some test classes and some perform integration testing (running the entire app). In this chapter, we will build a small library for property-based testing of Swift code.

When writing unit tests, the input data is static and defined by the programmer. For example, when unit-testing an addition method, we might write a test that verifies that `1 + 1` is equal to `2`. If the implementation of addition changes in such a way that this property is broken, the test will fail. More generally, however, we might test that addition is commutative, or in other words, that `a + b` is equal to `b + a`. To test this, we might write a test case that verifies that `42 + 7` is equal to `7 + 42`.

In this chapter, we'll build Swift port (a part of) QuickCheck [2] a Haskell

In this chapter, we'll build Swift port (a part of) QuickCheck, a Haskell library for random testing. Instead of writing individual unit tests that each test a function is correct for some particular input, QuickCheck allows you to describe abstract *properties* of your functions and *generate* tests to verify them.

This is best illustrated with an example. Suppose we want to verify that plus is commutative. To do so, we start by writing a function that checks where `x + y` is equal to `y + x` for two integers `x` and `y`:

```
func plusIsCommutative(x : Int, y : Int) -> Bool {
    return x + y == y + x
}
```

Checking this statement with QuickCheck is as simple as calling the `check` function:

```
check("Plus should be commutative", plusIsCommutative)

> "Plus should be commutative" passed 100 tests.
> ()
```

The `check` function works by calling the `plusIsCommutative` function with two random integers, over and over again. If the statement isn't true, it will print out the input that caused the test to fail. The key insight here is that we can describe abstract *properties* of our code (like commutativity) using *functions* that return a `Bool` (like `plusIsCommutative`). The `check` function now uses this property to *generate* unit tests; giving much better code coverage than you could achieve using hand-written unit tests.

Of course, not all tests pass. For example, we can define a statement that describes that the subtraction is commutative:

```
func minusIsCommutative(x : Int, y : Int) -> Bool {
```

```
    return x - y == y - x
}
```

Now, if we run QuickCheck on this function, we will get a failing test case:

```
check("Minus should be commutative", minusIsCommutative)

> "Minus should be commutative" doesn't hold: (0, 1)
> ()
```

Using Swift's syntax for trailing closures, we can also write tests directly, without defining the property (such as `plusIsCommutative` or `minusIsCommutative`) separately:

```
check("Additive identity") {(x : Int) in x + 0 == x }

> "Additive identity" passed 100 tests.
> ()
```

Building QuickCheck

In order to build this library, we will need to do a couple of things. First, we need a way to generate random values for different kinds of types. Then, we need to implement the `check` function, which will feed our test with random values a number of times. Should a test fail, then we'd like to make the input smaller. For example, if our test fails on an array with 100 elements, we'll try to make it smaller and see if the test still fails. Finally, we'll need to do some extra work to make sure our check function works on types that have generics.

Generating Random Values

First, let's define a protocol that knows how to generate arbitrary values. As the return type, we use `Self`, which is the type of the class that implements it.

```
protocol Arbitrary {
    class func arbitrary() -> Self
}
```

First, let's write an instance for `Int`. We use the `arc4random` function from the standard library and convert it into an `Int`:

```
extension Int : Arbitrary {
    static func arbitrary() -> Int {
        return Int(arc4random())
    }
}
```

Now we can generate random integers like this:

```
Int.arbitrary()


> 2194277656
```

To generate random strings, we need to do a little bit more work. First, we generate a random length `x` between 0 and 100. Then, we generate `x` random characters, and reduce them into a string.

```

extension Character : Arbitrary {
  static func arbitrary() -> Character {
    return Character(UnicodeScalar(random(from: 13, to:255)))
  }
```

```
  func smaller() -> Character? { return nil }
}

extension String : Arbitrary {
    static func arbitrary() -> String {
        let randomLength = random(from: 0, to: 100)
        let randomCharacters = repeat(randomLength) { _ in Character
.arbitrary() }
        return reduce(randomCharacters, "", +)
    }

}
```

We can call it in the same way as we generate random `Int`s, except, that we call it on the `String` class:

```
String.arbitrary()

> Âejòĉ[ ĆòÁ4æ.Oôòaeæ Q;mą=Ą3V=Đ1({.Y½NuíSÖEH·¢Y³OPÇê¸8*×ċ|'Ä6ÕJ=ÆćsÐ
è»
```

Implementing the `check` function

Now we are ready to implement a first version of our check function. The `check1` function consists of a simple loop that generates random input for the argument property in every iteration. If a counterexample is found, it is printed and the function returns; if no counterexample is found, the `check1` function reports the number of successful tests that have passed. (Note that we called the function `check1`, because we'll write the final version a bit later).

```
func check1<A: Arbitrary>(message: String, prop: A -> Bool) -> () {
    for _ in 0..<numberOfIterations {
        let value = A.arbitrary()
        if !prop(value) {
            println("\"\(message)\" doesn't hold: \(value)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

Here's how we can use this function to test properties:

```
check1("Additive identity") {(x : Int) in x + 0 == x }

> "Additive identity" passed 100 tests.
> ()
```

Making values smaller

If we run our `check1` function on strings, we might get quite a long failure message:

```
check1("Every string starts with Hello") {(s: String) in s.hasPrefix
("Hello")}

> "Every string starts with Hello" doesn't hold: aU;³Ø#ÆÇðIg«¡'¡ëbßÈ
Z[ą/Ø¤ÿYª,d4$FĂMÉâRăk¶4Àz
> ()
```

Ideally, we'd like our failing input to be a short as possible. In general, the smaller the counterexample, the easier it is to spot which piece of

the smaller the counterexample, the easier it is to spot which piece of code is causing the failure. In principle, the user could try to trim the input that triggered the failure and try rerunning the test; rather than place the burden on the user, however, we will automate this process.

To do so, we will make an extra protocol called `Smaller`, which does only one thing: it tries to shrink the counterexample.

```
protocol Smaller {
    func smaller() -> Self?
}
```

Note that the return type of the `small` function is marked as optional. There are cases when it is not clear how to shrink test data any further. For example, there is no obvious way to shrink an empty array. We will return `nil` in that case.

In our instance for integers we just try to divide the integer by two until we reach zero:

```
extension Int : Smaller {
    func smaller() -> Int? {
        return self == 0 ? nil : self / 2
    }
}
```

We can now test our instance:

```
100.smaller()


> 50
```

For strings, we just drop the first character (unless the string is empty).

```
extension String : Smaller {
    func smaller() -> String? {
        return self.isEmpty ? nil : self[startIndex.successor()..<en
dIndex]
    }
}
```

To use the `Smaller` protocol in the `check` function, will need the ability to shrink any test data generated by our `check` function. To do so, we will redefine our `Arbitrary` protocol to extend the `Smaller` protocol:

```
protocol Arbitrary : Smaller {
    class func arbitrary() -> Self
}
```

Repeatedly shrinking

We can now redefine our `check` function to shrink any test data that triggers a failure. To do this, we use the `iterateWhile` function that takes a condition, an initial value and repeatedly applies a function as long as the condition holds.

```
func iterateWhile<A>(condition: A -> Bool, initialValue: A, next: A
-> A?) -> A {
    if let x = next(initialValue) {
        if condition(x) {
           return iterateWhile(condition,x,next)
        }
    }
    return initialValue
}
```

Using `iterateWhile` we can now repeatedly shrink counterexamples that we uncover during testing:

```
func check2<A: Arbitrary>(message: String, prop: A -> Bool) -> () {
    for _ in 0..<numberOfIterations {
        let value = A.arbitrary()
        if !prop(value) {
            let smallerValue = iterateWhile({ value in !prop(value)
}, value) {
              $0.smaller()
            }
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

Arbitrary Arrays

Currently, our `check2` function only supports `Int` and `String` values. While we are free to define new extensions for other types, such as `Bool`, things get more complicated when we want to generate arbitrary arrays. As a motivating example, let's write a functional version of QuickSort:

```
func qsort(var array: [Int]) -> [Int] {
    if array.isEmpty { return [] }
    let pivot = array.removeAtIndex(0)
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    return qsort(lesser) + [pivot] + qsort(greater)
}
```

We can also try to write a property to check our version of QuickSort against the built-in sort function:

```
check2("qsort should behave like sort", { (x: [Int]) in return qsort
(x) == x.sorted(<) })
```

However, the compiler warns us that `[Int]` doesn't conform to the `Arbitrary` protocol. In order to implement `Arbitrary`, we first have to implement `Smaller`. As a first step, we provide a simple definitian that drops the first element in the array:

```
extension Array : Smaller {
    func smaller() -> [T]? {
        return self.isEmpty ? nil : Array(self[startIndex.successor(
)..<endIndex])
    }
}
```

We can also write a function that generates an array of arbitrary length for any type that conforms to the `Arbitrary` protocol:

```
func arbitraryArray<X: Arbitrary>() -> [X] {
  let randomLength = Int(arc4random() % 50)
  return repeat(randomLength) {_ in return X.arbitrary() }
}
```

Now what we'd like to do is define an extension that uses the `abritraryArray` function to give the desired `Arbitrary` instance for arrays. However, to define an instance for `Array`, we also need to make sure that the element type of the array is also an instance of `Arbitrary`. For example, in order to generate an array of random numbers, we first need to make sure that we can generate random numbers. Ideally, we would

to make sure that we can generate random numbers. Ideally, we would write something like this, saying that the elements of an array should also conform to the arbitrary protocol:

```
extension Array<T: Arbitrary> : Arbitrary {
    static func arbitrary() -> [T] {
        ...
    }
}
```

Unfortunately, it is currently not possible to express this restriction as a type constraint, making it impossible to write an extension that makes `Array` conform to the `Arbitrary` protocol. Instead we will modify the `check2` function.

The problem with the `check2<A>` function was that it required the type `A` to be `Arbitrary`. We will drop this requirement, and instead require the necessary functions, `shrink` and `arbitrary`, to be passed as arguments.

We start by defining an auxiliary struct that contains the two functions we need:

```
struct ArbitraryI<T> {
    let arbitrary : () -> T
    let smaller: T -> T?
}
```

We can now write a helper function that takes such an `ArbitraryI` struct as an argument. The definition of `checkHelper` closely follows the `check2` function we saw previously. The only difference between the two is where the `arbitrary` and `smaller` functions are defined. In `check2` these were constraints on the generic type, `<A : Arbitrary>`; in `checkHelper` they are passed explicitly in the `ArbitraryI` struct:

```
func checkHelper<A>(arbitraryInstance: ArbitraryI<A>, prop: A -> Boo
l, message: String) -> () {
    for _ in 0..<numberOfIterations {
        let value = arbitraryInstance.arbitrary()
        if !prop(value) {
            let smallerValue = iterateWhile({ !prop($0) }, value, ar
bitraryInstance.smaller)
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

This is a standard technique: instead of working with functions defined in a protocol, we pass the required information as an argument explicitly. By doing so, we have a bit more flexibility. We no longer rely on Swift to *infer* the required information, but have complete control over this ourself.

We can redefine our `check2` function to use the `checkHelper` function. If we know that we have the desired `Arbitrary` definitions, we can wrap them in the `ArbitraryI` struct and call `checkHelper`:

```
func check<X : Arbitrary>(message: String, prop : X -> Bool) -> () {
    let instance = ArbitraryI(arbitrary: { X.arbitrary() }, smaller:
 { $0.smaller() })
    checkHelper(instance, prop, message)
}
```

If we have a type for which we cannot define the desired `Arbitary` instance, like arrays, we can overload the `check` function and construct the desired `ArbitraryI` struct ourself:

```
func check<X : Arbitrary>(message: String, prop : [X] -> Bool) -> ()
 {
    let instance = ArbitraryI(arbitrary: arbitraryArray, smaller: {
(x: [X]) in x.smaller() })
    checkHelper(instance, prop, message)
}
```

Now, we can finally run `check` to verify our QuickSort implementation. Lots of random arrays will get generated and passed to our test.

```
check("qsort should behave like sort", { (x: [Int]) in return qsort(
x) == x.sorted(<) })

> "qsort should behave like sort" passed 100 tests.
> ()
```

Next steps

This library is far from complete, but already quite useful. There are a couple of obvious things that could be improved:

- The Arbitrary instances are quite simple. For different datatypes, we might want to have more complicated arbitrary instances. For example, when generating arbitrary enum values, we might want to generate certain cases with different frequencies. We might also want to generate constrained values (for example, what if we want to test a function that expects sorted arrays?). When writing multiple `Arbitrary` instances, we might want to define some helper functions that aid us in writing these instances.
- We might want to classify the generated test data. For example, if we generate a lot of arrays of length 1, we could classify this as a 'trivial' test case. The Haskell library has support for classification, these

ideas could be ported directly.

There are many other small and large things that could be improved to make this into a full library.

Diagrams

⚠ this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes. ⚠

In this chapter, we'll look at a functional way to describe diagrams, and how to draw them with Core Graphics. By wrapping Core Graphics with a functional layer, we get an API that's simpler and more composable.

Drawing squares and circles

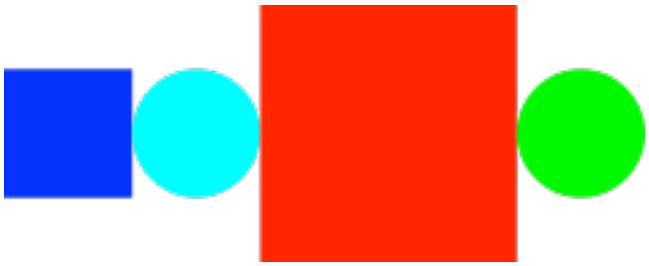
Imagine drawing the following diagram:

In Core Graphics, we could achieve this drawing with the following command:

```
[[NSColor blueColor] setFill]
CGContextFillRect(context, CGRectMake(0.0,37.5,75.0,75.0))
[[NSColor redColor] setFill]
CGContextFillRect(context, CGRectMake(75.0,0.0,150.0,150.0))
```

```
CGContextFillRect(context, CGRectMake(75.0,0.0,150.0,150.0))
[[NSColor greenColor] setFill]
CGContextFillEllipseInRect(context, CGRectMake(225.0,37.5,75.0,75.0))
```

This is very nice and short, but it is a bit hard to maintain. For example, what if we wanted to add an extra circle:



We would need to add the code for drawing a rectangle, but also update the drawing code to move some of the other objects to the right. In Core Graphics, we always describe *how* to draw things. In this chapter, we'll build a library for diagrams that allows us to express *what* we want draw. For example, the first diagram can be expressed like this:

```
let blueSquare = square(side: 1).fill(NSColor.blueColor())
let redSquare = square(side: 2).fill(NSColor.redColor())
let greenCircle = circle(radius: 1).fill(NSColor.greenColor())
let example1 = blueSquare ||| redSquare ||| greenCircle
```

And adding the second circle is as simple as changing the last line of code:

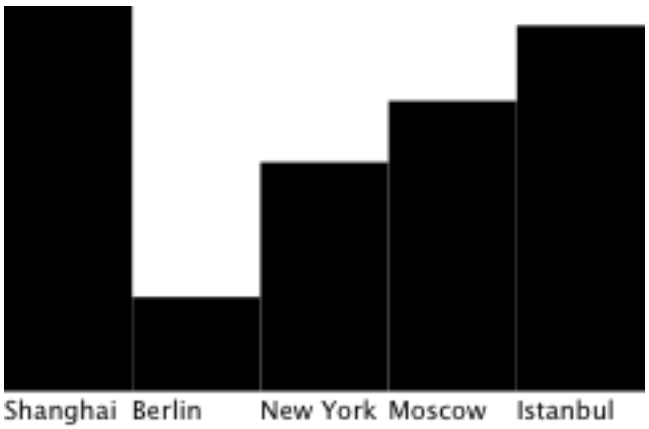
```
let example2 = blueSquare ||| circle(radius: 1).fill(NSColor.cyanCol
or()) ||| redSquare ||| greenCircle
```

The code above first describes a blue square, with a relative size of 1. The red square is twice as big (it has a relative size of 2), and we

compose the diagram by putting the squares and the circle next to each other with the `|||` operator. Changing this diagram is very simple, and there's no need to worry about calculating frames or moving things around. The examples describe *what* should be drawn, not *how* it should be drawn.

One of the techniques we'll use in this chapter is building up an intermediate structure of the diagram. Instead of executing the drawing commands immediately, we build up a data structure that describes the diagram. This is a very powerful technique, as it allows us to inspect the data structure, modify it and convert it into different formats.

As a more complex example of a diagram generated by the same library, here's a bar graph:



We can write a `barGraph` function that takes a list of names (the keys) and values (the relative heights of the bars). For each value, we draw a rectangle, and horizontally concatenate them with the `hcat` function. For each name, we draw it as text, also horizontally concatenate them using `hcat`, and finally put the bars and the text below each other using the `---` operator:

```
func barGraph(input: [(String,Double)]) -> Diagram {
```

```
    let values : [Double] = input.map { $0.1 }
    let bars =  hcat(normalize(values).map { (x: Double) -> Diagram
in
        return rect(width: 1, height: 3*x).fill(NSColor.blackColor()
).alignBottom()
    })
    let labels = hcat(input.map { x in
        return text(width: 1, height: 0.3, text: x.0).fill(NSColor.c
yanColor()).alignTop()
    })
    return bars --- labels
}
let cities = ["Shanghai": 14.01, "Istanbul": 13.3, "Moscow": 10.56,
"New York": 8.33, "Berlin": 3.43]
let example3 = barGraph(cities.keysAndValues)
```

Primitives

In our library, we'll draw three kinds of things: ellipses, rectangles and text. Using enums, we can define a datatype for that:

```
enum Primitive {
    case Ellipsis
    case Rectangle
    case Text(String)
}
```

It would be very possible to extend this with other primitives, such as images or more complex shapes.

Diagrams

Diagrams are defined using an enum as well. First, a diagram could be a primitive, which has a size and is either an ellipsis, a rectangle or text. Note that we call it `Prim`, because at the time of writing, the compiler gets confused by a case that has the same name as another enum.

```
case Prim(CGSize, Primitive)
```

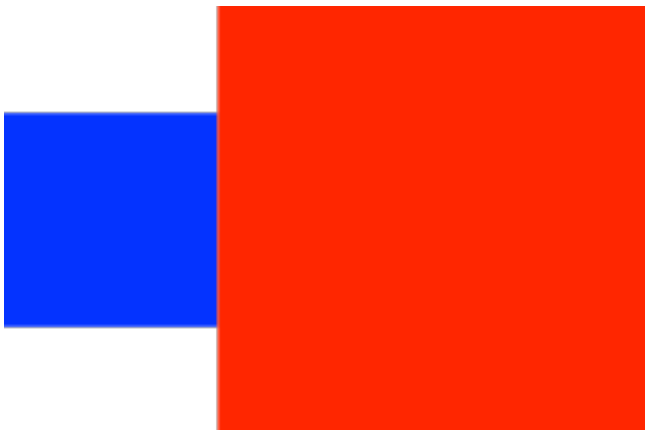
Then, we have cases for diagrams that are beside each other (horizontally) or below each other (vertically). Note how a `Beside` diagram is defined recursively: it consists of two diagrams next to each other.

```
case Beside(Diagram,Diagram)
case Below(Diagram,Diagram)
```

To style diagrams, we'll add a case for attributed diagrams. This allows us to set the fill color (for example, for ellipses and rectangles). We'll define the `Attribute` type later.

```
case Attributed(Attribute,Diagram)
```

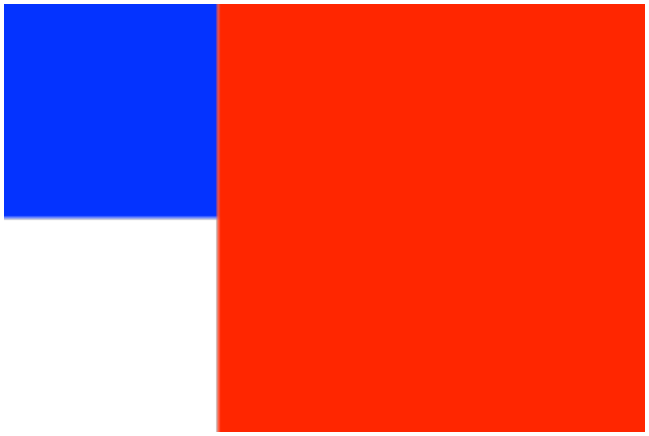
The last case is for alignment. Suppose we have a small and a large rectangle that are next to each other. By default, the small rectangle gets centered vertically:



But by adding a case for alignment we can control the alignment of smaller parts of the diagram.

```
case Align(Vector2D, Diagram)
```

For example, here's a diagram that's top-aligned:



It is drawn using the following code:

```
Diagram.Align(Vector2D(x: 0.5,y: 1), blueSquare) ||| redSquare
```

Unfortunately, in the current version of Swift, recursive datatypes are not allowed. So instead of having a `Diagram` case that contains other `Diagram`s, we created an extra protocol `DiagramLike`, and change our `Diagram` definition accordingly:

```
enum Diagram {
    case Prim(CGSize, Primitive)
    case Beside(DiagramLike, DiagramLike)
    case Below(DiagramLike, DiagramLike)
```

```
    case Attributed(Attribute, DiagramLike)
    case Align(Vector2D, DiagramLike)
}
```

The `DiagramLike` protocol has just one function, and only one instance:

```
protocol DiagramLike { func diagram() -> Diagram }

extension Diagram: DiagramLike {
  func diagram() -> Diagram { return self }
}
```

The `Attribute` enum is a datatype for describing different attributes of diagrams. Currently, it only supports `FillColor`, but it could easily be extended to support attributes for stroking, gradients, text attributes, etcetera:

```
enum Attribute {
    case FillColor(NSColor)
}
```

Calculating the size

Calculating the size for the `Diagram` datatype is easy. The only case that's not straightforward is for `Beside` and `Below`. In case of beside, the width is equal to the sum of the widths, and the height is equal to the maximum height of the left and right diagram. For below, it's a similar pattern. For all the other cases we just call size recursively.

```
extension Diagram {
  var size : CGSize {
    switch self {
```

```
      case .Prim(let size, _):
        return size
      case .Attributed(_, let x):
        return x.diagram().size
      case .Beside(let l, let r):
        let sizeL = l.diagram().size
        let sizeR = r.diagram().size
        return CGSizeMake(sizeL.width+sizeR.width,max(sizeL.height,s
izeR.height))
      case .Below(let l, let r):
        let sizeL = l.diagram().size
        let sizeR = r.diagram().size
        return CGSizeMake(max(sizeL.width,sizeR.width),sizeL.height+
sizeR.height)
      case .Align(_, let r):
        return r.diagram().size
    }
  }

}
```

Drawing the diagram

Before we start drawing, we will first define one more function. The `fit` function takes an alignment vector (which we used in the `Align` case of a diagram), an input size (i.e. the size of a diagram) and a rectangle that we want to fit the input size into. The input size is defined relatively to the other elements in our diagram. We scale it up, and maintain its aspect ratio.

```
func fit(alignment: Vector2D, inputSize: CGSize, rect: CGRect) -> CG
Rect {
```

```
    let div = rect.size / inputSize
    let scale = min(div.width, div.height)
    let size = scale * inputSize
    let space = alignment.size * (size - rect.size)
    let result = CGRect(origin: rect.origin - space.point, size: siz
e)
    return result
}
```

For example, if we fit and center square of 1x1 into a rectangle of 200x100, we get the following result:

```
fit(Vector2D(x: 0.5, y: 0.5), CGSizeMake(1,1), CGRectMake(0,0,200,10
0))

> (50.0,0.0,100.0,100.0)
```

To align the rectangle to the left, we would do the following:

```
fit(Vector2D(x: 0, y: 0.5), CGSizeMake(1,1), CGRectMake(0,0,200,100)
)

> (0.0,0.0,100.0,100.0)
```

Now that we can represent diagrams and calculate their sizes, we're ready to draw them. We use pattern matching to make it easy to know what to draw. The `draw` method takes a couple of parameters: the context to draw in, the bounds to draw in and the actual diagram. Given the bounds, the diagram will try to fit itself into the bounds using the `fit` function defined before. For example, when we draw an ellipse, we center it and make it fill the available bounds:

```
func draw(context: CGContextRef, bounds: CGRect, diagram: Diagram) {
```

```
  switch diagram {
    case .Prim(let size, .Ellipsis):
      let frame = fit(Vector2D(x: 0.5, y: 0.5), size, bounds)
      CGContextFillEllipseInRect(context, frame)
```

For rectangles, this is almost the same, except that we call a different Core Graphics function. You might note that the `frame` calculation is the same as for ellipses. It would be possible to pull this out and have a nested switch statement, but we think is is more readable when presenting in book-form.

```
    case .Prim(let size, .Rectangle):
        let frame = fit(Vector2D(x: 0.5, y: 0.5), size, bounds)
        CGContextFillRect(context, frame)
```

In the current version of our library, all text is set in the system font with a fixed size. It's very possible to make this an attribute, or change the `Text` primitive to make this configurable. In its current form though drawing text works like this:

```
    case .Prim(let size, .Text(let text)):
        let frame = fit(Vector2D(x: 0.5, y: 0.5), size, bounds)
        let attributes = [NSFontAttributeName: NSFont.systemFontOfSi
ze(12)]
        let attributedText = NSAttributedString(string: text, attrib
utes: attributes)
        attributedText.drawInRect(frame)
```

The only attribute we support is fill color. It's very easy to add support for extra attributes, but we left that out for brevity. To draw a diagram with a `FillColor` attribute, we save the current graphics state, set the fill color, draw the diagram and finally restore the graphics state:

```
    case .Attributed(.FillColor(let color), let d):
        CGContextSaveGState(context)
        color.set()
        draw(context, bounds, d.diagram())
        CGContextRestoreGState(context)
```

To draw two diagrams next to each other, we first need to find their respective frames. We created a function `splitHorizontal` that splits a `CGRect` according to a ratio (in this case, the relative size of the left diagram). Then we draw both diagrams with their frames.

```
    case .Beside(let left, let right):
        let l = left.diagram()
        let r = right.diagram()
        let (lFrame, rFrame) = splitHorizontal(bounds, l.size/diagra
m.size)
        draw(context, lFrame, l)
        draw(context, rFrame, r)
```

The case for `Below` is exactly the same, except that we split the `CGRect` vertically instead of horizontally. This code was written to run on the Mac, and therefore the order is `bottom` and `top` (unlike UIKit, the Cocoa coordinate system has the origin at the bottom left).

```
    case .Below(let top, let bottom):
        let t = top.diagram()
        let b = bottom.diagram()
        let (lFrame, rFrame) = splitVertical(bounds, b.size/diagram.
size)
        draw(context, lFrame, b)
        draw(context, rFrame, t)
```

Our last case is aligning diagrams. Here, we can reuse the fit function that we defined earlier to calculate new bounds that fit the diagram exactly.

```
    case .Align(let vec, let d):
        let diagram = d.diagram()
        let frame = fit(vec, diagram.size, bounds)
        draw(context, frame, diagram)
  }
}
```

We've now defined the core of our library. All the other things can be built on top of these primitives.

Wrapping the drawing code with an `NSView`

We can create a subclass of `NSView` that performs the drawing, which is very useful when working with playgrounds or when you want to draw these diagrams in Mac applications.:

```
class Draw : NSView {
    let diagram: Diagram

    init(frame frameRect: NSRect, diagram: Diagram) {
        self.diagram = diagram
        super.init(frame:frameRect)
    }

    override func drawRect(dirtyRect: NSRect) {
        draw(NSGraphicsContext.currentContext().cgContext, self.boun
ds, diagram)
    }
}
```

```
}
```

Now that we have an `NSView`, it's also very simple to make a PDF out of our diagrams. We calculate the size and just use `NSView`s method `dataWithPDFInsideRect` to get the PDF data.

```
func pdf(diagram: Diagram, width: CGFloat) -> NSData {
    let v : Draw = {
        let unitSize = diagram.size
        let height = width * (unitSize.height/unitSize.width)
        return Draw(frame: NSMakeRect(0, 0, width, height), diagram:
 diagram)
    }()
    return v.dataWithPDFInsideRect(v.bounds)
}
```

Extra combinators

To make the construction of diagrams easier, it's nice to add some extra functions (also called combinators). This is a common pattern in functional libraries: have a small set of core datatypes and functions, and then build convenience functions on top of that. For example, for rectangles, circles, text and squares we can define convenience functions:

```
func rect(#width: Double, #height: Double) -> Diagram {
    return Diagram.Prim(CGSizeMake(width, height), .Rectangle)
}

func circle(#radius: Double) -> Diagram {
    return Diagram.Prim(CGSizeMake(radius, radius), .Ellipsis)
}
```

```
func text(#width: Double, #height: Double, text theText: String) ->
Diagram {
    return Diagram.Prim(CGSizeMake(width, height), .Text(theText))
}

func square(#side: Double) -> Diagram {
  return rect(width: side, height: side)
}
```

Also, it turns out that it's very convenient to have operators for combining diagrams horizontally and vertically, making the code more readable. They are just wrappers around `Beside` and `Below`.

```
operator infix ||| { associativity left }
@infix func ||| (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Beside(l, r)
}

operator infix --- { associativity left }
@infix func --- (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Below(l, r)
}
```

We can also extend the `Diagram` type and add methods for filling and alignment. Instead, we might have defined these methods as top-level functions. This is a matter of style, one is not more powerful than the other.

```
extension Diagram {
    func fill(color: NSColor) -> Diagram {
      return Diagram.Attributed(Attribute.FillColor(color), self)
    }
```

```
    func alignTop() -> Diagram {
        return Diagram.Align(Vector2D(x: 0.5,y: 1), self)
    }

    func alignBottom() -> Diagram {
        return Diagram.Align(Vector2D(x:0.5, y: 0), self)
    }
}
```

Finally, we can define an empty diagram and a way to horizontally concatenate a list of diagrams. We can just use the array's `reduce` function to do this.

```
let empty : Diagram = rect(width: 0, height: 0)

func hcat(diagrams: [Diagram]) -> Diagram {
    return diagrams.reduce(empty, combine: |||)
}
```

By adding these small helper functions, we have a powerful library for drawing diagrams. Many things are still missing but can be added easily. For example, it's straightforward to add more attributes and styling options. A bit more complicated would be adding transformations (e.g. rotation), but this too is doable.

Wrapping Core Image

⚠ this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes. ⚠

Core Image is a powerful image processing framework, but its API can be a bit clunky to use at times. The Core Image API is loosely typed – image filters are configured using key-value-coding. It is all too easy to

Image filters are configured using key-value coding. It is all too easy to make mistakes in the type or name of arguments, which can result in run-time errors.

In this chapter we will develop a functional API, wrapped around the (object-oriented) Core Image framework. As a result, we can create and chain filters easily, while improving the static guarantees about our filters at the same time.

The Filter Type

When you instantiate a `CIFilter` object, you (almost) always provide an input image via the `kCIInputImageKey` key and then retrieve the filtered result via the `kCIOutputImageKey` key. Then you can use this result as input for the next filter.

So we can define a filter simply as a function that takes an image as parameter and returns an image:

```
typealias Filter = CIImage -> CIImage
```

This is the base type that we are going to build upon.

Building Filters

Now that we have the `Filter` type defined, we can start building functions defining specific filters. These are convenience functions that take the parameters needed for a specific filter and construct a value of type `Filter`. These functions will all have the following general shape:

```
func myFilter(/* parameters */) -> Filter
```

Note that the return value, `Filter`, is a function as well. This will help us later on to compose multiple filters to achieve the image effects we want.

want.

To make our lives a bit easier, we'll extend the `CIFilter` class with a convenience initializer and a computed property to retrieve the output image:

```
extension CIFilter {

    class func filter(name: String, parameters: Dictionary<String, A
nyObject>) -> CIFilter {
        let filter = self(name: name)
        filter.setDefaults()
        for (key, value : AnyObject) in parameters {
            filter.setValue(value, forKey: key)
        }
        return filter;
    }

    var outputImage: CIImage { return self.valueForKey(kCIOutputImag
eKey) as CIImage }

}
```

The convenience initializer takes the name of the filter and a dictionary as parameters. The key-value pairs in the dictionary will be set as parameters on the new filter object. As a convenience initializer we follow the Swift pattern of calling the designated initializer first.

The computed property is an easy way to retrieve the output image from the filter object. But it also makes sure that the output image has a defined type (`CIImage`), so that we don't have to write `as CIImage` all over the place in our code.

Blur

Blur

The gaussian blur filter only has the blur radius as parameter. Therefore we can write a blur `Filter` very easily:

```
func blur(radius: Double) -> Filter {
    return { image in
        let filter = CIFilter.filter("CIGaussianBlur", parameters: [
kCIInputRadiusKey: radius, kCIInputImageKey: image])
        return filter.outputImage
    }
}
```

That's all there is to it. The `blur` function returns a function that takes an image as parameter (`image in`...) and returns an image (`return filter.outputImage`). Therefore the return value of the `blur` function conforms to the `Filter` type we have defined as `CIImage -> CIImage`.

This example is just a thin wrapper around a filter that already exists in Core Image. We can use the same pattern to create our own filter functions that do more than that.

Color Overlay

Let's define a filter that overlays an image with a solid color of our choice. Core Image doesn't have such a filter by default, but we can of course compose it from existing filters.

The two building blocks we're going to use for this is the color generator filter (`CIConstantColorGenerator`) and the source over compositing filter (`CISourceOverCompositing`). Let's first define a filter to generate a constant color plane: