Introduction to Functional Programming in Swift

Joe Howard on December 22, 2015

If you're new here, you may want to subscribe to my RSS feed or follow me on Twitter. Thanks for visiting!

Swift's grand entrance to the programming world at WWDC in 2014 was much more than just an introduction of a new language. It brings a number of new approaches to software development for the iOS and OS X platforms.

This tutorial focuses on just one of these methodologies, **FP** (Functional Programming), and you'll get an introduction to a broad range of functional ideas and techniques.

Note: If you want to go a littler deeper into functional programming in Swift, check out the "Functional Programming" chapter in Swift by Tutorials, which is also available in tutorial form here.

Getting Started

Create a new playground in Xcode so you can investigate FP in Swift by select **File \ New \ Playground…** and name it **IntroFunctionalProgramming** and leave the default platform as **iOS**. Click **Next** to choose a location to save it, then **Create** to save the file.

Keep in mind that this tutorial will cover FP at a high level, so thinking about the concepts through the filter of a real world situation is helpful. In this case, imagine you're building an app for an amusement park, and that the park's ride data is provided by an API on a remote server.

Start by replacing the playground boilerplate code with the following:

```
enum RideType {
  case Family
  case Kids
```

```
  case Thrill
  case Scary
  case Relaxing
  case Water
}

struct Ride {
  let name: String
  let types: Set<RideType>
  let waitTime: Double
}
```

Here you've created a Swift enum named **`RideType`** that provides a set of types for rides. You also have a **`Ride`** struct that contains the **`name`** of a ride, a **`Set`** for ride types, and a **`double`** to hold rides' current wait time.

Now you're probably thinking about loop-de-loops and the thrill of feeling g-force on a wild ride, but trust me, there's no ride quite like making the shift from other programming styles to FP! :]

What is Functional Programming?

At the highest-level, the distinction between FP and other software writing approaches is most simply described as the difference between **imperative** and **declarative** thinking.

- Imperative thinking is thinking in terms of algorithms or steps when solving a problem, or the *how* of getting from A (the problem) to B (the solution). A perfect example of an imperative approach is a recipe for cupcakes that tells you how much to measure, how to mix and how to bake them.
- Declarative thinking analyzes a problem in terms of what the solution *is*, but not *how* to get to the solution. Take the concept of a recipe for an example. A declarative recipe would essentially show you a picture or offer a description of cupcakes, rather than tell you how to mix ingredients and how to bake them.

FP encourages a declarative approach to solving problems via functions. You'll see more distinction between imperative and declarative programming as you go through this tutorial.

Functional Programming Concepts

In this section, you'll get an introduction to a number of key concepts in FP. Many treatises that discuss FP single out **immutable state** and **lack of side effects** as the most important aspects of FP, so why not start there?

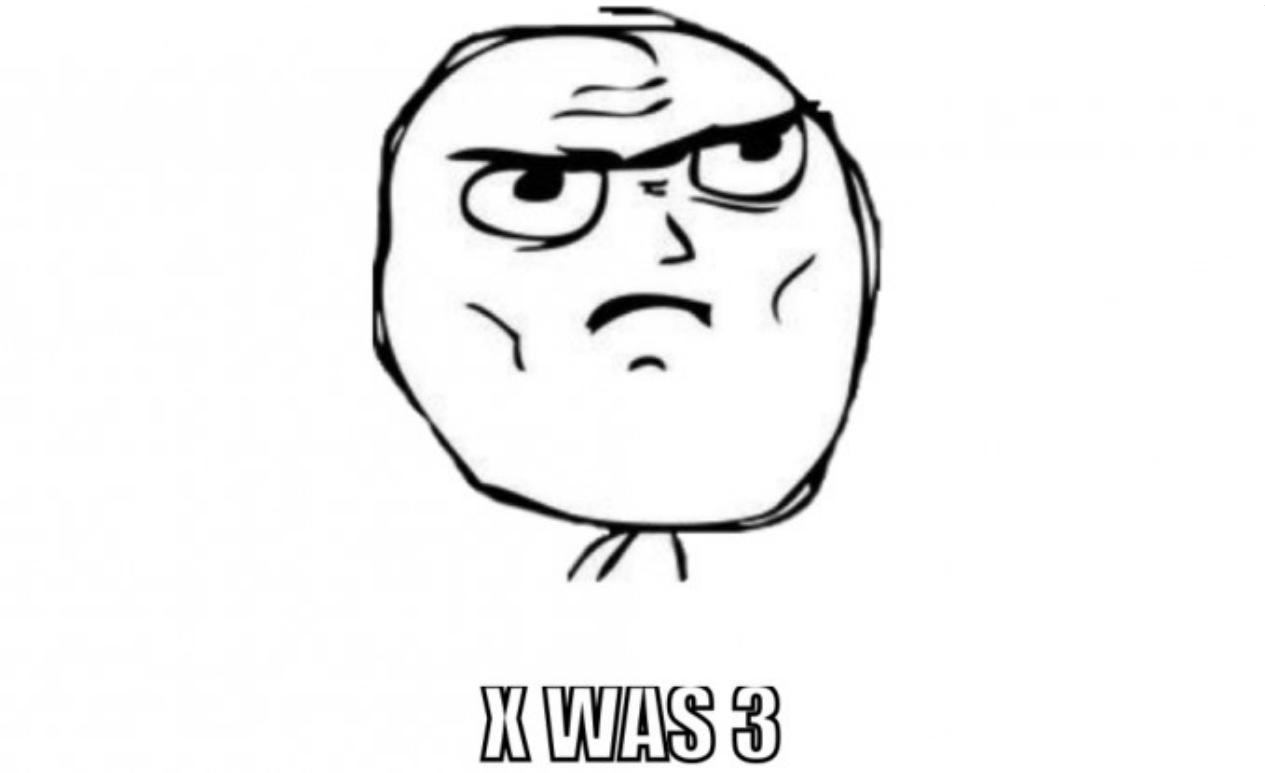
Immutability and Side Effects

No matter what programming language you learned first, it's likely that one of the initial concepts you learned was that a **variable** represents data. If you step back for a moment to really think about the idea, variables can seem quite odd.

Add the following seemingly reasonable and common lines of code to your playground:

```
var x = 3
// other stuff...
x = 4
```

As a developer that's trained in procedural programming and/or OOP, you wouldn't give this a second thought. But how exactly could a quantity be *equal* to 3 and then later be 4?!



The term variable implies a quantity that varies. Thinking of the quantity **x** from a mathematical perspective, you've essentially introduced **time** as a key parameter in how your software behaves. By changing the variable, you create **mutable state**.

By itself or in a relatively simple system, a mutable state is not terribly problematic. However, when connecting many objects together, such as in a large OOP system, a mutable state can produce many headaches.

For instance, when two or more threads access the same variable **concurrently**, they may modify or access it out of order, leading to unexpected behaviour. This includes **race conditions**, **dead locks** and many other problems.

Imagine if you could write code where the state *never* mutated. A whole slew of issues that occur in concurrent systems would simply vanish – *poof!* You can do this by creating an **immutable** property, or data that is not allowed to change over the course of a program.

This was possible in Objective-C by creating a constant, but your default mode for properties was to create mutable properties. In Swift, the default mode is immutable.

The key benefit of using immutable data is that units of code that use it would be free of **side effects**, meaning that the functions in your code don't alter elements outside of themselves, and no spooky effects can happen when function calls occur.

See what happens when you make the primary data you'll work with in this tutorial an immutable Swift constant by adding the following array to your playground below the `Ride` struct:

```
let parkRides = [
  Ride(name: "Raging Rapids", types: [.Family, .Thrill, .Water], waitTime: 45.0),
  Ride(name: "Crazy Funhouse", types: [.Family], waitTime: 10.0),
  Ride(name: "Spinning Tea Cups", types: [.Kids], waitTime: 15.0),
  Ride(name: "Spooky Hollow", types: [.Scary], waitTime: 30.0),
  Ride(name: "Thunder Coaster", types: [.Family, .Thrill], waitTime: 60.0),
  Ride(name: "Grand Carousel", types: [.Family, .Kids], waitTime: 15.0),
  Ride(name: "Bumper Boats", types: [.Family, .Water], waitTime: 25.0),
```

```
  Ride(name: "Mountain Railroad", types: [.Family, .Relaxing], waitTime: 0.0)
]
```

Since you create **parkRides** with **let** instead of **var**, both the array and its contents are immutable. Trying to modify one of the items in the array, via

```
parkRides[0] = Ride(name: "Functional Programming", types: [.Thrill], waitTime: 5.0)
```

Changing one of the items produces a compiler error. Go ahead and try to change those rides! No way, buster. :]

Modularity

You've reached the part where you'll add your first function, and you'll need to use some **NSString** methods on a Swift **String**, so import the **Foundation** framework by putting this at the very top of your playground:

```
import Foundation
```

Suppose you need an alphabetical list of all the rides' names. You are going to start out doing this imperatively. Add the following function to the bottom of the playground:

```
func sortedNames(rides: [Ride]) -> [String] {
  var sortedRides = rides
  var i, j : Int
  var key: Ride

  // 1
  for (i = 0; i < sortedRides.count; i++) {
    key = sortedRides[i]

    // 2
    for (j = i; j > -1; j--) {
      if key.name.localizedCompare(sortedRides[j].name) == .OrderedAscending {
        sortedRides.removeAtIndex(j + 1)
        sortedRides.insert(key, atIndex: j)
      }
    }
  }

  // 3
  var sortedNames = [String]()
  for ride in sortedRides {
    sortedNames.append(ride.name)
  }

  print(sortedRides)

  return sortedNames
}
```

Here you are:

1. Looping over all the rides passed into the function
2. Performing an insertion sort
3. Gathering the names of the sorted rides

From the perspective of a caller to `sortedNames(:)`, it provides a list of rides, and then outputs the list of sorted names. Nothing outside of `sortedNames(:)` has been affected. To prove this, first print out the output of a call to sorted names:

```
print(sortedNames(parkRides))
```

Now gather the names of the list of rides that were passed as a parameter and print them:

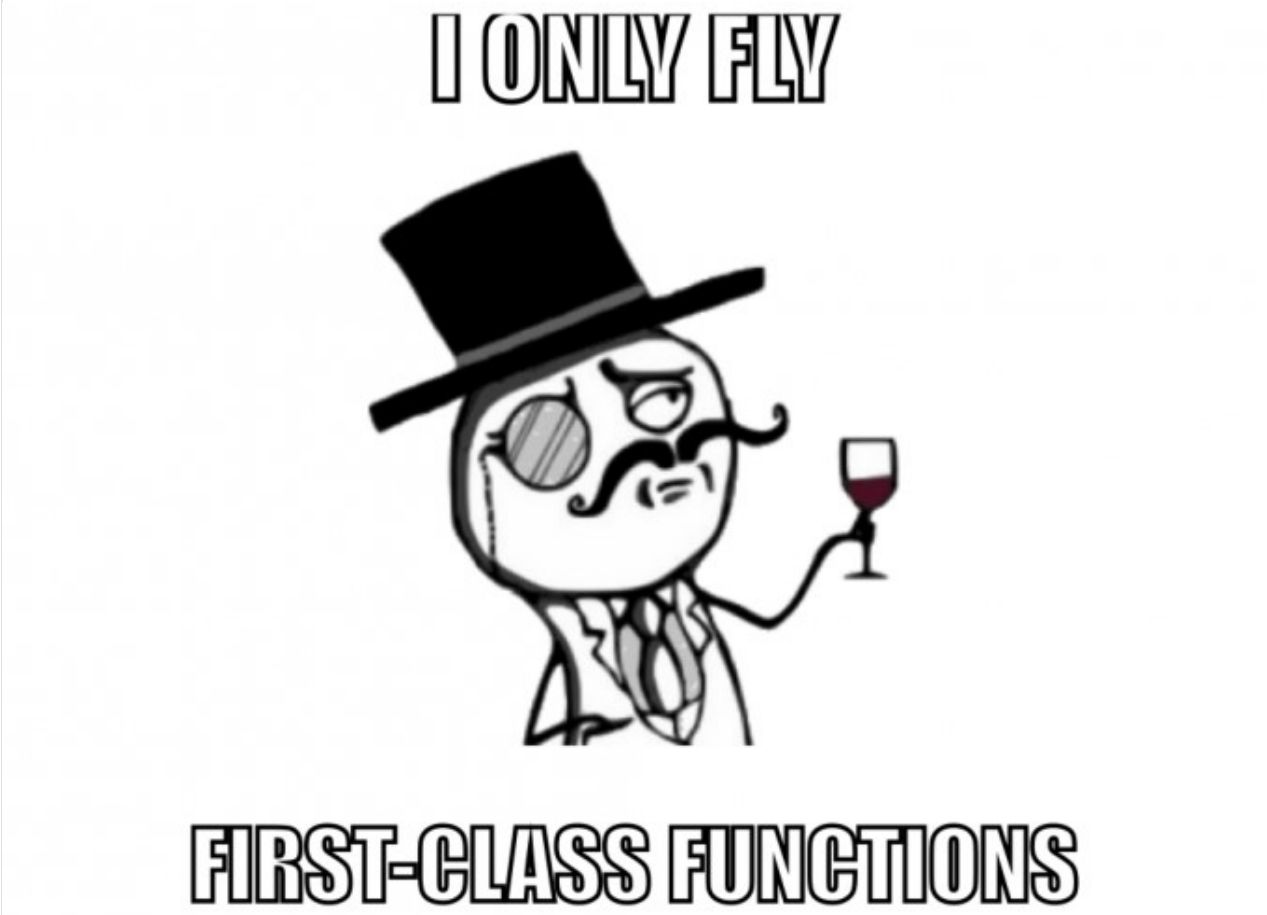
```
var originalNames = [String]()
for ride in parkRides {
  originalNames.append(ride.name)
}

print(originalNames)
```

In the assistant editor, you'll see that sorting rides inside of `sortedNames(:)` didn't affect the list that was passed in. The modular function you've created could be considered *quasi-functional* with all that imperative code. The logic of sorting rides by name has been captured in a single, testable **modular** and reusable function.

Still, the imperative code made for a pretty long and unwieldy function. Wouldn't it be nice if there were techniques to simplify the code within a function like `sortedNames(:)` even further?

First-Class and Higher-Order Functions

Another characteristic of FP languages is that functions are **first-class** citizens, meaning that functions can be assigned to values and passed in and out of other functions. There's a higher level yet: **higher-order** functions. These accept functions as parameters or return other functions.



In this section, you'll work with some of the most common higher-order functions in FP languages, namely **filter**, **map** and **reduce**.

Filter

In Swift, `filter(:)` function is a method on `CollectionType` values, such as Swift arrays, and it accepts another function as its single parameter as it maps the type of values in the array to a Swift `Bool`.

`filter(:)` applies the input function to each element of the calling array and returns another array that has only the array elements for which the parameter function returns `true`.

This back to your list of actions that `sortedNames` did:

1. Looping over all the rides passed into the function
2. Performing an insertion sort
3. Gathering the names of the sorted rides

Think about this declaratively rather than imperatively.

Start by commenting out `sortedNames` because you are going to write this much more efficiently. Create a function for `Ride` values to be used as a function parameter at the bottom of the playground:

```
func waitTimeIsShort(ride: Ride) -> Bool {
  return ride.waitTime < 15.0
}
```

The function `waitTimeIsShort(:)` accepts a ride and returns `true` if the ride wait time is less than 15 minutes, otherwise it returns `false`.

Call `filter` on your park rides and pass in the new function you just created:

```
var shortWaitTimeRides = parkRides.filter(waitTimeIsShort)
print(shortWaitTimeRides)
```

You defined `shortWaitTimeRides` as a `var` only because you'll reuse it in the playground. You'll repeat that approach elsewhere. In the playground output, you only see Crazy Funhouse and Mountain Railroad in the call to filter's output, which is correct.

Since Swift functions are just named **closures**, you can produce the same result by passing a trailing closure to `filter` and using closure-syntax:

```
shortWaitTimeRides = parkRides.filter { $0.waitTime < 15.0 }
print(shortWaitTimeRides)
```

Here, .filter() is taking every ride in the parkRides array ($0), looking at its waitTime property, and gauging if it is less than 15.0. You are being declarative and telling the program what you want it to do instead of how it is done. This can look rather cryptic the first few times you work with it.

Map

The `CollectionType` method `map(:)` also accepts a *single function* as a parameter, and in turn, it produces an array of the same length after being applied to each element of the collection. The return type of the mapped function does *not* have to be the same type as the collection elements.

Apply `map` to the elements of your `parkRides` array to get a list of all the ride names as strings:

```

```

```
let rideNames = parkRides.map { $0.name }
print(rideNames)
```

You can also sort and print the ride names as shown below, when you use the (non-mutating) `sort` method on `MutableCollectionType` to perform the sorting:

```
print(rideNames.sort(<))
```

Your `sortedNames(:)` method from before is now just two lines thanks to `map(:)`, `sort(:)` and use of a closure!

Reduce

The `CollectionType` method `reduce(:,:)` takes two parameters. The first is a starting value of a **generic** type `T`, and the second is a function that combines a value of type `T` with an element in the collection to produce another value of type `T`. The input function applies to each element of the calling collection, one-by-one, until it reaches the end of the collection and produces a final accumulated value of type `T`.

Suppose you want to know the average wait time of all the rides in your park.

Pass in a starting value of 0.0 into `reduce` and use a trailing-closure syntax to add in the contribution of each ride to the average, which is its wait time divided by the total number of rides.

```
let averageWaitTime = parkRides.reduce(0.0) { (average, ride) in average +
(ride.waitTime/Double(parkRides.count)) }
print(averageWaitTime)
```

In this example, the first two iterations look like the following:

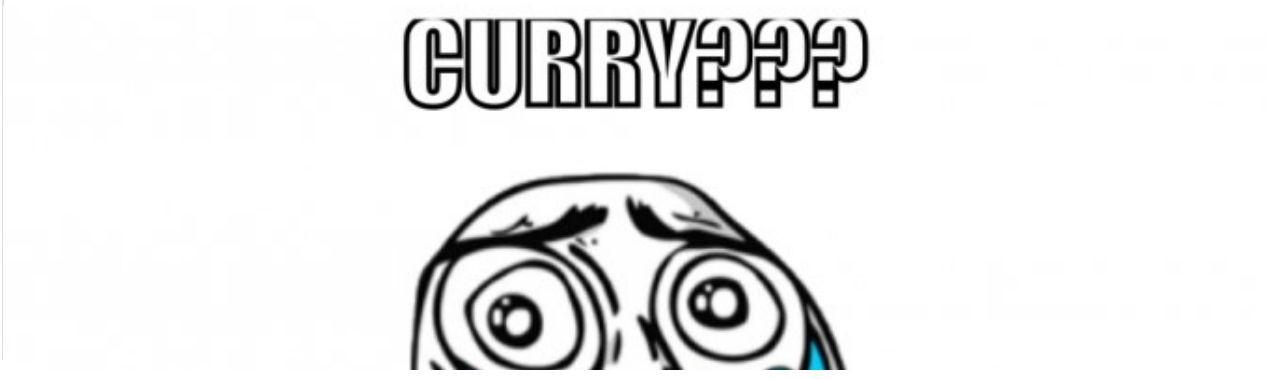
| iteration | average | ride.waitTime / Double(parkRides.count) | resulting average ||—|—|—|—||1|0.0|45.0 / 8.0 = 5.625 | 0.0 + 5.625 = 5.625||2|5.625|10.0 / 8.0 = 1.25 | 5.625 + 1.25 = 6.875|

As you can see, the resulting average carries over as the average for the following iteration. This continues until `reduce` iterates through every `Ride` in `parkRides`. This allows you to get the average with just one line of code!

Currying

> **Note**: With Swift now having gone open source, it looks like function currying syntax will be removed from the language in Swift 3.0. You can find more info here, and especially check out the “Detailed design” section.

One of the more advanced techniques in FP is **currying**, named after mathematician Haskell Curry. Currying essentially means thinking of multi-parameter functions as a sequence of function applications for each parameter, one-by-one.



Just to be clear, in this tutorial, Curry is not a delicious, spicy meal that elicits tears of joy, but you can demonstrate currying by creating a two-parameter function that separates the two parameters via parentheses:

```
func rideTypeFilter(type: RideType)(fromRides rides: [Ride]) -> [Ride] {
  return rides.filter { $0.types.contains(type) }
}
```

Here, `rideTypeFilter(:)(:)` accepts a `RideType` as its first parameter and an array of rides as its second.

One of the best uses of currying is to make other functions. By calling your curried function with just one parameter, you make another function that will only require one parameter when called.

Try it for yourself by adding the following `RideType` filter factory to your playground:

```
func createRideTypeFilter(type: RideType) -> [Ride] -> [Ride] {
  return rideTypeFilter(type)
}
```

Notice that `createRideTypeFilter` returns a function that maps an array of rides to an array of rides. It does so by calling `rideTypeFilter` and passing in only the first parameter. Use `createRideTypeFilter` to create a filter for kid rides, and then apply that filter to your list of park rides, like so:

```
let kidRideFilter = createRideTypeFilter(.Kids)

print(kidRideFilter(parkRides))
```

In the playground output, you see that `kidRideFilter` is a function that filters out all non-kid rides.

Pure Functions

One of the primary concepts in FP that leads to the ability to reason consistently about program structure, as well as confidently test program results, is the idea of a **pure function**.

A function can be considered **pure** if it meets two criteria:

- **Always** produces the same output when given the same input, e.g., the output *only* depends on the input
- Creates **zero** side effects outside of the function

A pure function's existence is closely tied to the usage of immutable states.

Add the following pure function to your playground:

```
func ridesWithWaitTimeUnder(waitTime: Double, fromRides rides: [Ride]) -> [Ride] {
  return rides.filter { $0.waitTime < waitTime }
}
```

`ridesWithWaitTimeUnder(:fromRides:)` is a pure function because its output is always the same when given the same wait time threshold and list of rides as input.

With a pure function, it's simple to write a good **unit test** against the function. To simulate a unit test, add the following **assert** into your playground:

```
var shortWaitRides = ridesWithWaitTimeUnder(15.0, fromRides: parkRides)
assert(shortWaitRides.count == 2, "Count of short wait rides should be 2")
print(shortWaitRides)
```

Here you're testing if you always get a count of two rides when given the fixed input **`parkRides`** and a wait time threshold of 15.0.

Referential Transparency

Pure functions are closely related to the concept of **referential transparency**. An element of a program is referentially transparent if you can replace it with its definition and always produce the same result. It makes for predictable code and allows the compiler to perform optimizations.

Pure functions satisfy this condition. If you're familiar with Objective-C, then you'll find that pre-processor **`#define`** constants and macros are familiar examples of referentially transparent code items.

Check if the pure function **`ridesWithWaitTimeUnder`** is referentially transparent by replacing its use with its function body:

```
shortWaitRides = parkRides.filter { $0.waitTime < 15.0 }
assert(shortWaitRides.count == 2, "Count of short wait rides should be 2")
print(shortWaitRides)
```

Recursion

The final concept to discuss is function **recursion**, which occurs whenever a function calls *itself* as part of its function body. In functional languages, recursion effectively replaces **looping** constructs that are used in imperative languages.

When the function's input leads to the function being called again, it's considered the **recursive case**. In order to avoid an infinite stack of function calls, recursive functions need a **base case** to end them.

Add a recursive sorting function for your rides by first making the **`Ride`** struct conform to the **`Comparable`** protocol by placing a **`Ride`** extension in your playground:

```
extension Ride: Comparable { }

func <(lhs: Ride, rhs: Ride) -> Bool {
  return lhs.waitTime < rhs.waitTime
}

func ==(lhs: Ride, rhs: Ride) -> Bool {
  return lhs.name == rhs.name
}
```

You've added the required `Comparable` operator methods outside the extension, per the requirements of Swift operator-overloading. You're calling one ride *less than* another ride if the wait time is less, and calling them *equal* if the rides have the same name.

Now, add a recursive `quicksort` method to your playground:

```
func quicksort<T: Comparable>(var elements: [T]) -> [T] {
  if elements.count > 1 {
    let pivot = elements.removeAtIndex(0)
    return quicksort(elements.filter { $0 <= pivot }) + [pivot] + quicksort
(elements.filter { $0 > pivot })
  }
  return elements
}
```

`quicksort(:)` is a generic function that takes an array of `Comparable` values and sorts them by picking a pivot element, then it calls itself for elements before and after the pivot .

Add this to check the sorting of a list of rides:

```
print(quicksort(parkRides))
print(parkRides)
```

The second line confirms that `quicksort(:)` didn't modify the immutable array that was passed in.

Imperative vs. Declarative Smackdown

By considering the following problem, you can combine much of what you've learned here on FP and get a clear demonstration of the differences between imperative and functional programming:

> **Problem Statement**: A family that has very young kids wants to go on as many rides as possible between frequent bathroom breaks, so they need to find which kid-friendly rides have the shortest lines. Help them out by finding all family rides with wait times less than 20 minutes and sort them by wait time (ascending).

Imperative Approach

Momentarily ignore all you've learned about FP so far and think about how you would solve this problem with an algorithm. You would probably:

1. Create an empty mutable array of rides
2. Loop over your existing rides and find ones that have a type `.Family`
3. If a family ride has a wait time under 20 minutes, you'd add it to your mutable array
4. Finally, you'd sort those rides by wait time

Step-by-step, this is how you'd answer the problem statement. Here it is as an algorithm implementation, please add it to your playground:

```
var ridesOfInterest = [Ride]()
for ride in parkRides {
  var typeMatch = false
  for type in ride.types {
```

```
    if type == .Family {
      typeMatch = true
      break
    }
  }
  if typeMatch && ride.waitTime < 20.0 {
    ridesOfInterest.append(ride)
  }
}

var sortedRidesOfInterest = quicksort(ridesOfInterest)

print(sortedRidesOfInterest)
```

You should see that Mountain Railroad, Crazy Funhouse and Grand Carousel are the best bets, and they're listed in order of increasing wait time. You've taken advantage of the `quicksort(:)` function written above to do the sorting.

As written, the imperative code is acceptable, but a quick glance does not give a clear, immediate idea of what it's doing. You have to pause to look at the algorithm in detail to grasp it. No big deal you say? What if you're coming back to do some maintenance, debugging or you're handing it off to a new developer? As they say on the Internet, "You're doing it wrong!"

Functional Approach 1

FP can do better. Add the following one-liner to your playground:

```
sortedRidesOfInterest = parkRides.filter({ $0.types.contains(.Family) && $0.waitTime
< 20.0 }).sort(<)
```

You've used the familiar methods `filter` and `contains` on `Set` to find matching rides, as well as `sort` from before.

Verify that this single line produces the same output as the imperative code by adding:

```
print(sortedRidesOfInterest)
```

There you have it! In one line, you've told Swift *what* to calculate; you want to filter your `parkRides` to `.Family` rides with wait times less than 20 minutes and then sort them. That precisely – and elegantly – solves *the problem statement*.

FP not only makes your code more concise, but also makes what it does self-evident. I dare you to find a side effect!

Functional Approach 2

The above one-line solution could be considered slightly opaque, but a modified variation can take advantage of the curried filter factory method you made earlier to create a family ride filter:

```
let familyRideFilter = createRideTypeFilter(.Family)
sortedRidesOfInterest = ridesWithWaitTimeUnder(20.0, fromRides: familyRideFilter
(parkRides)).sort(<)

print(sortedRidesOfInterest)
```

Here you produce the same result in a slightly more human-friendly form. Once you have the family ride filter, you can state the solution to the problem in one line of code again.

The When and Why of Functional Programming

Swift is not a purely functional language, but it does combine multiple programming methodologies to give you flexibility for application development.

A great place to start working with FP techniques is in your Model layer, your ViewModel layer, and anywhere that your application's business logic appears.

For user interfaces, how to use FP techniques is a little less clear. **FRP** (Functional Reactive Programming) is an example of an approach to FP for UI development. For example, Reactive Cocoa is an FRP library for iOS and OS X programming.

In terms of *why* to use FP, hopefully this tutorial has given you some good reasons and ideas for use.

By taking a functional, declarative approach, your code can be more concise and clear. As if you need another reason to work with FP, keep in mind that your code easier to test when isolated into modular functions that are free from side effects.

Finally, as multi-processing cores become the norm for both CPUs and GPUs, minimizing side effects and issues from concurrency will become increasingly important, and FP will one of the most important ways to achieve smooth performance!

Where to Go From Here?

You can download the complete playground with all the code in this tutorial here.

While you've reviewed many of the important concepts of FP in this tutorial, you've only scratched the surface of what FP is and what it can do. Once you have some experience with these basic FP ideas, take the plunge into the heart of FP:

- Monads, Endofunctors, Category Theory
- Programming languages where FP is the focus like Haskell and Clojure

You'll get great insights into various aspects of Swift by studying both of the above. You might be able to do things you never dreamed possible with your new-found knowledge.

Also, I highly recommended checking out:

Functional Programming: The Failure of State, a talk given by Robert C. Martin in 2014

Learning Functional Programming without Growing a Neckbeard, a talk by Kelsey Innis. It's focused on Scala but has lots of good info.

Joe Howard

Joe's path to software development began in the fields of computational physics and systems engineering. He has been a mobile software developer on iOS and Android since 2009. He now lives in Boston and is a Lead Engineer at Raizlabs, a leading app development company dedicated to great software.