Functional Programming In Swift

A Hands On Presentation

What is Functional Programming?

Functional programming is a methodology that emphasizes:

- calculations via mathematical-style functions
- Immutable state leading to a lack of side effects
- Code that is self-explanatory and reads like the problem statement it solves.
- Minimal use of variables and state.

Key Features: Modularity

Repeatedly break each program into smaller and smaller components, that can easily be re-assembled using function application to define a complete program. Such decomposition of a large program into smaller pieces requires avoidance of sharing state between individual components.

Key Features: Careful Treatment Of Mutable State

Object-oriented programming focuses on the design of classes and objects, each with their own encapsulated state. Functional programming, on the other hand, emphasizes the importance of programming with values, free of mutable state or other side effects. By avoiding mutable state, functional programs can be more easily combined than their imperative or object-oriented counterparts.

Key Features:Careful Use of Types

A well-designed functional program makes careful use of types. More than anything else, a careful choice of the types of your data and functions will help structure your code.

Possible Advantages

- Unintended state mutation is minimized or eliminated
- Code is more expressive of its intent and therefore easier to understand
- More comprehensive unit testing is possible because components tested are simpler

Functional Programming Challenges

- learning to think functionally is not easy.
- 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;
- closures, generics, higher-order functions, and monads are just plain weird.

Swift Features That Enable A Functional Approach

- Several “value data types”, where each instance keeps a unique copy of its data, usually defined as a `struct`, `enum`, or tuple. Common “value” types: Array, Dictionary, and String
- Type aliasing to rename an existing type.
- Strongly incentivizes use of constants over variables
- Closures are first class objects. They can be passed to or returned from functions like any other data type.

Type Aliasing

The typealias keyword is used to rename an existing type. This can make code more readable, and clearer in context.

Examples:

```
typealias Distance = Double

typealias DoNothing = () -> Void
```

What’s A Closure?

- self-contained block of functionality that can be passed around in an application.
- can capture and store references to any constants and variables from the context in which they are defined. This is known as *closing over* constants and variables.
- Swift handles all memory management related to capturing
- A function is a special case of a closure

What’s A Closure Look Like?

```
{  // opening bracket

() // parameter grouping

 -> Void //return type

in // beginning of programming statements

} // closing bracket

//example

let doNothing = {() ->Void in }  //assigning closure to a constant
```

Some Built-In Functional Features In Swift

The Sequence type provides features found in many functional languages. Among them:

- Map -- performs an operation on each element in a sequence and returns a new sequence embodying the specified transformation on each element
- Filter -- performs a decision operation that determines which elements from the original sequence will be included in the new sequence returned
- Reduce -- returns a single value based on operations performed on each element in a sequence

Functions Can Be Chained

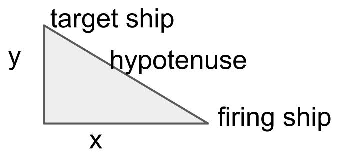
```
let originalArray = [1,2,3,4]

let evenNumber = { (input:Int) -> Bool in return input % 2 == 0 }

let tripleTheNumber = { (input:Int) -> Int in return input * 3 }

let aSingleValue = originalArray.map({ tripleTheNumber($0) })
                                .filter({ evenNumber($0)})
                                .reduce(0,+)
```

Hands On Functional Programming

- Example From the Book “Functional Programming in Swift” by Chris Eidhof
- Program to determine whether a firing battle ship can hit a given target ship at a safe range and without friendly fire
- The original example is quite terse. I tarted it up a bit to help me understand it.
- Wrote “smoke” unit tests in a playground to confirm functionality and “de-composability”

Concepts To Understand How Battleship Works

- One ship’s distance from another is expressed as the hypotenuse of a right triangle.



- The formula for determining this hypotenuse is to square the x and y coordinates of the target ship and calculate the hypotenuse by taking the square root of their sums.
- For example, if the target ship is at position x:3, y:4 it will be at a distance of 5 from the firing ship because 3*3 = 9, 4*4 = 16, 16 + 9 = 25, and the square root of 25 = 5 (i.e. 5 * 5)
- This formula is known as the Pythagorean Theorem