Functional Programming
in
Swift

What is functional programming?

First…what is object-oriented programming?

Object-oriented programming is a paradigm in which objects are the smallest unit of computation.

Functional programming is a paradigm in which the function* is the smallest unit of computation.

* Function in the mathematical sense

```c
void write_to_file(char *string, char *path)
{
    FILE *fh = fopen(path, "w");
    if (!fh) return;
    fwrite(string, strlen(string), sizeof(char), fh);
    fclose(fh);
}
```

Computational Functions

(aka procedures)

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0, n = 0 \\ A(m-1, A(m, n-1)) & m > 0, n > 0 \end{cases}$$

Mathematical Functions

(Like in functional programming languages)

Why does this matter?

- Referential transparency means less state to reason about, and (possibly) fewer bugs
- An alternative way to think about constructing programs
- Better compiler support for writing correct programs (for static languages)

Static vs. Dynamic

- ML (1978)
- Standard ML (1990)
- Haskell (1990)
- Scala (2003)
- F# (2005)

- Lisp (1958)
- Scheme (1975)
- Common Lisp (1984)
- Erlang (1986)
- Clojure (2007)

Swift is a statically-typed language.

Modeling Computations

In OOP, you generally start off by thinking about all your program’s classes and how they interact with each other.

But which OOP model is best?

- Simula 67/C++: Static types, classes, inheritance, virtual methods
- Smalltalk 80/Objective-C: Dynamic types, classes, inheritance, “messages”
- Smalltalk 76/Erlang: Dynamic types, processes, message passing
- But…ah…which one is best? So confusing!

Functional programming: Start at the bottom and work your way up

What key constructs does Swift have to support functional programming?

1. Types (and type inferencing)

Unlike Objective-C, Swift's objects have a concrete type.

But the compiler can figure them out for you, so you don’t have to annotate them à la C++ or Java.

```
id obj = get_some_object();
// What is the type of obj? Not even the compiler knows.
```

```
let obj = "my object"
// What is the type of obj? It's a String!
```

2. Option types

```java
Person john = Person.getPerson("John");
if (john != null) {
    if (john.getLastName() != null) {
        System.out.println("John's last name is " + john.getLastName());
    } else {
        System.out.println("John has no last name.");
    }
} else {
    System.out.println("No person named John.");
}
```

```
let john = getPerson("John")
if let lastName = john?.lastName {
    println("John's last name is \(lastName)")
} else {
    println("John does not exist or has no last name");
}
```

3. Enums

- Swift enums are not like C enums—they are more like algebraic data types (sort of)
- Algebraic data types are a kind of composite data type (but not like classes in object-oriented languages, or even structs in C and C++).
- You can (usually) do pattern matching on abstract data types!

```swift
enum JSValue {
    case JSNumber(Double)
    case JSString(String)
    case JSBool(Bool)
    case JSArray(Array<JSValue>)
    case JSObject(Dictionary<String,JSValue>)
    case JSNull
}

func jsonValueToString(obj: JSValue) -> String {
    switch (obj) {
    let .JSNumber(n):
        return n.description
    let .JSString(s):
        return s
    let .JSBool(b):
        if b {
            return "true"
        } else {
            return "false"
        }
    let .JSArray(vals):
        return vals.map { jsonValueToString }.join("\n")
    let .JSObject(key, val):
        return "\(key): \(jsonValueToString(val))"
    case .JSNull
        return "null"
}
```

So much more

- Closures!
- First-class functions!
- Higher-order functions!
- (Functional) reactive programming!