A Swift Introduction

Part Four

Guard Statement

- Used to transfer program control out of a scope if one or more conditions aren’t met
- Has the following form:

```
guard condition else {
    statements
}
```

- Can be used to unwrap optional values

Guard Statement

- The value of any condition in a `guard` statement must be of type `Bool` or a type bridged to `Bool`
- The `else` clause of a `guard` statement is required, and must:
  - Either call a function with the `Never` return type
  - Or transfer program control outside the guard statement's enclosing scope using one of the following statements
    - `return`
    - `break`
    - `continue`
    - `throw`

Guard Statement

- Ex:

```
func sqrt(number: Double) -> Double? {
    guard number >= 0 else { return nil }

    return sqrt(number)
}
```

- Ex:

```
let numbers = Array(1...10)

for number in numbers {
    guard number.isMultiple(of: 2) else { continue }

    print("\(number) is a mulitple of 2")
}
```

- Ex:

```
func submit(feedback: String?) {
    guard let feedback = feedback else { return }

    let serviceClient = ServiceClient()
    serviceClient.makeServiceCall(data: feedback.data(using: .utf8), completion: {_ in})
}
```

Any and AnyObject

- **Any** is a generic placeholder for **any type whatsoever** (instances of a struct, enum, class, protocol)

- **AnyObject** is similar but is limited to instances of classes and class protocols

Access Control

- Restricts access to parts of your code from code in other source files and modules

- There are **five** access control levels:

  - **open**: only applies to classes (it means the class can be subclassed and methods can be overridden)

  - **public**: visible to any module importing your module

  - **internal**: the ***default*** access scope when one is not specified; visible to any file or entity in your module

  - **fileprivate**: use restricted to the source-defining file

  - **private**: use restricted to the enclosing declaration **private**: use restricted to the enclosing declaration

Instance Methods

- Methods and variables on structs and classes are by default **instance methods**

- Ex:

```
class Game {
    var players: [String] = []

    func addPlayer(player: String) {
        players.append(player)
    }
}

let gameInstance = Game()
gameInstance.addPlayer(player: "Ethan")
```

Type Methods

- Methods and variables on structs and classes can be at the **Type level** as well
- Ex:

```
class SettlersGame {
    static let name = "Settlers of Catan"   // Instance Method
    var players: [String] = []

    class func rules() -> String {  // Instance Method
        return "Get 10 victory points"
    }

    func addPlayer(player: String) {
        players.append(player)
    }
}

let gameInstance = SettlersGame()
gameInstance.addPlayer(player: "Ethan")

let rules = SettlersGame.rules()
let name = SettlersGame.name
```

Type Methods

- Indicate type methods by writing the `static` keyword before the method's `func` keyword

- Classes can use the `class` keyword instead, to allow subclasses to override the superclass's implementation of that method

Type Casting

A way to:

1. Check the type of an instance
2. Treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy
3. Treat a type as a protocol or as the underlying type adopting the protocol

as Operator

```
class MediaItem {}

class Movie: MediaItem {}
class Song: MediaItem {}

for item in library {
    if let movie = item as? Movie {
        // The item is of type Movie
    } else if let song = item as? Song {
        // The item is of type Song
    }
}
```

`is` Operator

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}
```

Error Handling

- Some operations aren't guaranteed to always complete execution or produce a useful output.

- Optionals are used to represent the absence of a value

- BUT when an operation fails, *sometimes* you want to know why to understand what caused the failure

- Reading and processing data from a file on disk is a good example. It could fail because the file does not exist, the user does not have the right permissions, etc

Error Handling

- Use `throw` to throw an error
- Use `throws` to mark a function that can throw an error.
- Throwing an error in a function causes the function to immediately return and the code that called the function handles the error

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}

func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

Do-Catch

- Inside the `do` block: Mark code that can throw an error by writing `try` in front of it

- Inside the `catch` block: The error is automatically given the name `error` unless explicitly given a different name

- Can have multiple `catch` blocks

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
```

Error Handling

```swift
enum FileError: Error {
    case cannotAccessFile, fileDoesNotExist
}

func canThrowErrors(number: Int) throws -> String {
    if number < 0 {
        throw FileError.cannotAccessFile
    }
    return "A string!"
}

// This won't compile because the function is marked with `throws` and we haven’t handle the error.
let result1 = canThrowErrors(number: 0)

// We need to use `do`, `try`, and `catch` to handle the error
do {
    let result2 = try canThrowErrors(number: -1)
    print("This doesn't get executed")
} catch (let error as FileError) {
    print("Error: \(error)")
}

// Or we can also convert errors to optionals, when we don’t care about the details
let result3 = try? canThrowErrors(number: -1)
print(result3)
```