A Swift Introduction

Part Two

Control Flow

Conditionals

- Use `if`, `else-if`, ternary operators (`?:`) and `switch` to construct conditionals

- The **ternary conditional operator** is shorthand for an `else-if statement and` takes the form:

```
question ? answer1 : answer2
```

- Parentheses around the condition variable are optional

- For `if`, `else-if` and switch, Braces around the body are required

Conditionals

- In an `if` statement, the conditional must be a **Boolean** expression

```
let score = 60

if score > 89 {
    print("You got an A!")
}

if score > 60 {
    print("You passed the test!")
} else if score == 60 {
    print("Cutting it close! 😱”)
} else {
    print("Better luck next time!")
}
```

- Switches support any kind of data and a wide variety of comparison operations—they aren’t limited to integers and tests for equality

```
let vegetable = "red pepper"
switch vegetable {
    case "celery":
        print("Add some raisins and make ants on a log.")
    case "cucumber", "watercress":
        print("That would make a good tea sandwich.")
    case let x where x.hasSuffix("pepper"):
        print("Is it a spicy \(x)?")
    default:
        print("Everything tastes good in soup.")
}
```

Loops

- Use `for`-`in`, `while`, and `repeat`-`while` to make loops
- Parentheses around the condition variable are optional
- Braces around the body are required

For-In

- Used to iterate over collections

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    teamScore += score
}

print(teamScore)
```

- Keep an index in a loop by using `..<` to make a range of indexes
    - Use `..<` to make a range that omits its upper value, and use `...` to make a range that includes both values

```
var total = 0
for index in 0..<4 {
    total += index
}

print(total)

var largest = 0
for number in numbers {
    if number > largest {
        largest = number
    }
}

print(largest)
```

For-In

- Iterate over items in a `Dictionary` by providing a pair of names to use for each key-value pair
- Dictionaries are an unordered collection, so their keys and values are iterated over in an arbitrary order

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]

var largest = 0
for (kind, numbers) in interestingNumbers {
    print("Assessing \(kind) numbers")
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}

print(largest)
```

Repeat-While

- Use `while` to repeat a block of code until a condition changes

```
var number = 2
while number < 100 {
    number *= 2
}

print(number)
```

- The condition of a loop can be at the end instead, ensuring that the loop is run at least once

```
var number = 2
repeat {
    number *= 2
} while number < 100

print(number)
```

Structs, Enums and Classes

Enums

- An **enumeration** defines a common type for a group of related values
- Enables working with those values in a type-safe way
- Replaces having lots of string constants (in C/C++) or `enums` solely based on `Ints` which can easily get confusing

Enums

- Can have associated functions and variables
- Variables cannot contain stored values (must be computed)
- Can have `rawValues`
- `rawValues` can be implicit, ex: `Int, String`
- By default, Swift assigns `Int` raw values starting at zero and incrementing by one each time
- This behavior can be changed by explicitly specifying values

```
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king

    var description: String {
        switch self {
        case .ace: return "Ace"
        case .jack: return "Jack"
        case .queen: return "Queen"
        case .king: return "King"
        default: return String(self.rawValue)
        }
    }
}

let cardRank = Rank.king
print(cardRank.rawValue)
print(cardRank.description)
```

Enums

- Can have **associated values**
- Defined when initializing a specific case

```
enum Activity {
    case bored
    case running(destination: String)
    case talking(topic: String)
    case singing(volume: Int)
}

let talking: Activity = .talking(topic: "football")
```

CaseIterable

- Declare an `enum CaseIterable` when it is necessary to access all cases of an `enum` as a collection

```
enum Suit: String, CaseIterable {
    case spades = "Spades"
    case hearts = "Hearts"
    case diamonds = "Diamonds"
    case clubs = "Clubs"

    var name: String { return rawValue }
    var isBlack: Bool { return [.spades, .clubs].contains(self) }
}

Suit.allCases.forEach { suit in
    print("\(suit.name) \(suit.isBlack ? "is Black" : "is Red")")
}
```

Structs

- Can have properties (mutable/ immutable / computed), methods and initializers
- Passed by value
- Employs **copy-on-write**: the object is actually copied when it is changed
- User `mutating` keyword to mark methods that modify the structure

```
struct Student {
    let name: String
    var school: String
    var major: String?

    var id: String { "\(name)-\(UUID().uuidString)" }

    func learn() {}

    mutating func change(major: String) {
        self.major = major
    }
}

var angie = Student(name: "Angie", school: "UMSL", major: nil) // notice that we don't have to define init
angie.change(major: "Computer Science")
```

Classes

- Support many of the same behaviors as `structs`
    - `mutating` keyword not necessary, methods on a `class` can always modify the `class`
- Passed by reference
- Require at least one initializer
    - All stored properties must be initialized before leaving the scope of `init()`

```
class Dog {
    let name: String
    var energy: Int = 10
    var happiness: Int = 10

    var isContent: Bool { return energy > 5 && happiness > 7 }

    init(name: String) {
        self.name = name
    }

    func eat() {
        energy += 3
        happiness += 5
        daydream()
    }

    private func daydream() {
        happiness += 2
    }
}
```

Class Inheritance

- Classes can inherit from, at most, **one** other `class`

```
class GoldenRetriever: Dog {
    var loyalty: Int = 10

    override var isContent: Bool { return super.isContent && loyalty > 9 }

    override func eat() {
        // don't have to call the super class implementation; just depends on logic that you want
        super.eat()
        loyalty += 1
    }
}

let rover = GoldenRetriever(name: "Rover")
```

Protocols

- A list of defined behavior
- Objects adopting a `protocol` must implement all methods and properties defined in the `protocol` (similar to Java interfaces or a class in C++ with all pure virtual methods)
- Classes, enumerations, and structs can all adopt multiple protocols

```
protocol CanEat {
    var isHungry: Bool { get set }

    func eat()
}

protocol CanSleep {
    func sleep()
}

class Cat: CanEat, CanSleep {
    var isHungry: Bool = true

    func eat() {
        // eat something
    }

    func sleep() {
        // go to sleep
    }
}
```

Value and Reference Types

- Types in Swift fall into one of two categories:
  - **Value types**: Each instance keeps a unique copy of its data, ex:
    - `struct`
    - `enum`
    - Tuple
- **Reference types**: Instances share a single copy of the data, ex:
  - `class`

Value Types

- Copying — the effect of assignment, initialization, and argument passing — creates an **independent instance** with its own unique copy of its data

```
struct ValueTypeExample {
    var data: Int = -1
}

var structA = ValueTypeExample()
var structB = structA                           // structA is copied to structB
structA.data = 42                               // Changes structA, not structB
print("\(structA.data), \(structB.data)")       // prints "42, -1"
```

Reference Types

- Copying a reference implicitly creates a shared instance
- After a copy:
  - Two variables then refer to a single instance of the data
  - Modifying the data in the second variable also affects the original

```
class ReferenceTypeExample {
    var data: Int = -1
}

var classA = ReferenceTypeExample()
var classB = classA                         // classA is copied to classB
classA.data = 42                            // changes the instance referred to by classA (and classB)
print("\(classA.data), \(classB.data)")     // prints "42, 42"
```

Extensions

- Add functionality to an existing type, such as:
  - New methods
  - Computed properties
- Can be used to add protocol conformance to a type

```
extension Array {
    func at(_ index: Int) -> Element? {
        return index >= 0 && index < count ? self[index] : nil
    }
}
```

Points to Remember

- Swift supports the standard **ternary operator** (?:)
- `switch` statements support pattern matching capabilities
- `enums` can be simple or can have raw values
- `enums` can have computed variables and methods
- `structs` can have immutable and mutable **properties**, as well as **computed variables** and **methods**
- `structs` have an automatically-generated memberwise initializer, custom initializers may still be defined
- `structs` are great for pieces of data that need to get passed around a lot because they're **value types**: meaning they are **passed-by-value** to functions
- `enums` are value types as well
- Many of Swift's 'primitive' types (`Int, String, Bool, Double`, ...) are actually implemented as Structs
- Under the covers, Swift uses **copy-on-write** for Structs, so the actual value is only copied when it is changed (which helps performance a LOT)

Points to Remember

- Classes are **passed-by-reference** and must be used carefully to avoid side-effects
- Classes require you to define at least one initializer (unlike Structs)
- All stored properties must be initialized in init() and before calling any methods (same for Structs)
- At a basic level, most of the capabilities of Classes are similar to Structs (mutable and immutable properties, methods, computed properties, access control)
- Classes can only have one super class from which they inherit
- In iOS/Swift, we typically use Classes for Models and View Controllers (and other objects that need to persist 'state'), but for data which is passed around, prefer value types first
- Protocols define required behavior of whoever implements them (Classes or Structs)
- A Class or Struct can adopt multiple protocols
- Prefer **composition and using protocols** over **inheritance**