iOS Development with Swift

Optional Chaining, Error Handling, Protocols & Extension

Week *4*

Outline

- Optional Chaining
- Error Handling
- Type Casting
- Nested Types
- Extensions
- Protocols
- Workshop

Optional Chaining

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be **nil**.

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

Question or Exclamation Mark

You specify optional chaining by placing a question mark (**?**) after the optional value on which you wish to call a property, method or subscript if the optional is non-nil.

This is very similar to placing an exclamation mark (**!**) after an optional value to force the unwrapping of its value.

What is nil?

nil means “no value” sometimes called an empty or unassigned value. It is the absence of a value of a certain type. Optionals of any type can be set to nil, not just object type

```
var havingChildren : Int? // it can be nil or assigned
```

Optional Chaining Cont'd

If you create a new **Person** instance, its residence property is default initialized to **nil**, by virtue of being optional. In the code below, john has a residence property value of **nil**:

```
let john = Person()
```

Optional Chaining Cont'd

If you try to access the **numberOfRooms** property of this person's residence, by placing an exclamation mark after residence to force the unwrapping of its value, you trigger a runtime error, because there is no residence value to unwrap:

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

Optional Chaining Cont'd

Optional chaining provides an alternative way to access the value of **numberOfRooms**. To use optional chaining, use a question mark in place of the exclamation mark:

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

Optional Chaining Cont'd

This tells Swift to "chain" on the optional residence property and to retrieve the value of numberOfRooms if residence exists.

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

Optional Chaining Cont'd

The main difference is that optional chaining fails gracefully when the optional is nil, whereas forced unwrapping triggers a runtime error when the optional is nil.

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
```

Error Handling

Error handling is the process of responding to and recovering from error conditions in your program.

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

Error Handling

Another example of error handling. You will see using of **guard** keywords.

```
enum EncryptionError: Error {
    case Empty
    case Short
}

func checkPassword(password: String) throws {
    guard password.characters.count > 0 else { throw EncryptionError.Empty }
    guard password.characters.count >= 5 else { throw EncryptionError.Short }
}

do{
    try checkPassword(password: "abc")
}catch EncryptionError.Empty{
    print("empty")
}catch EncryptionError.Short{
    print("short")
}
```

Type Casting

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

```
class Animal { }

class Cat : Animal{ }

class Tiger : Cat{ }

func printAnimal(_ animal : Animal){
    print("here is an animal")
}

let tiger = Tiger()
printAnimal(tiger)
```

Type Casting for Any and AnyObject

Swift provides two special types for working with nonspecific types:

- **Any** can represent an instance of any type at all, including function types.
- **AnyObject** can represent an instance of any class type.

Type Casting for Any and AnyObject

Use **Any** and **AnyObject** only when you explicitly need the behavior and capabilities they provide.

```
var things = [Any]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

Nested Types

Enumerations are often created to support a specific class or structure's functionality. Similarly, it can be convenient to define utility **classes** and **structures** purely for use within the context of a more complex type.

```
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
    }
}
```

Extensions

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code.

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

Extensions

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

Extensions

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

Extensions

Here is an example of extension:

```
extension String{
    var htmlString : String{
        return "<html><head></head><body>\(self)</body></html>"
    }
}
```

Protocols

A **protocol** defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements.

Protocols Cont'd

You define protocols in a very similar way to classes, structures, and enumerations:

```
protocol EngineProtocol{
    func throttle()
    func start()
    func stop()
}
```

Workshop - 1

Question: Create a protocol called **GameProtocol**

- Protocol will have following functions

  moveLeft()

  moveRight()

  moveUp()

  moveDown()

- Create a class called **Point** having Float x and y coordinates.
- Write a class called **SimpleGame** that implements **GameProtocol.**
- **SimpleGame** game class will have a point instance. If **moveLeft** function is called x coordinate will be **x = x -1 moveUp** will be **y = y+1** and etc.

Workshop - 2

Question: Extend **Double** class and implement following functions.

- Write a function named **celsiusToFahrenheit()** that converts celsius degree to fahrenheit and return that value.
- Write a function named **fahrenheitToCelsius()** that converts fahrenheit degree to celsius and return that value.
- Write a function named **celsiusToKelvin()** that converts celcius degree to kelvin and return that value.
- Write a function named **kelvinToCelcius()** that converts kelvin degree to celsius and return that value.

Workshop - 3

Question: Create a class named **Student** that has id, first name, last name attributes.

- Create another class named **Job** that has department and position attributes.
- **Student** class will have optional **Job** instance which means student will have part-time job.
- Write a function called **info()** inside of Student class that prints student info and job's department and position if available.
- Create two instances of Student class. One will have a job another don't.
- Call **info()** functions on each instance.