Optionals in Swift

Mobile Computing - iOS

Objectives

- Students will be able to:
    - explain the purpose of optionals
    - distinguish between optionals and implicitly unwrapped optionals
    - explain the purpose of optional chaining

What are Optionals?

- An optional is a variable or stored property used in situations where a value may be unknown.

- An optional either *has* a value (of a specific type), or it is *nil*

- An optional is declared by writing a **?** after the type

```
var age:Int?
var weight:Double?
```

- Optionals need not be assigned a value before init() is complete, because ... it is *optional*: if you don't supply one, **it is assigned the value nil.** [Hence, this does not violate the rule that all variables must have a value before initialization is complete.]

- Note that Int? ≠ Int. For example, you can't do arithmetic on an Int? directly. Instead, you must write an exclamation point (**!**) after the variable name to access the enclosed value. This is **forced unwrapping.**

Arithmetic on an Int?

```
var x:Int? // x currently contains nil

x = 1       // now x contains 1
x = 2 * x   // compile error: x is not unwrapped.
x = 2 * x! // success: ! unwraps x to access the Int inside




var y:Int?  // y currently contains nil

y = 2 * y! // fatal error: unexpectedly found nil while
unwrapping an Optional value
```

Where Might You See An Optional?

- Optionals pop up in all *sorts* of places. One common one is when converting a String to a numeric value (Int, Double, etc.). Since the String might not be in the correct format, initialization could fail. To indicate that, the initializer returns an optional.

```swift
let possibleNum:String = "123.45" // good luck!
let convertedNum:Int? = Int(possibleNum)

// Since conversions from Strings to Ints can fail (if
the String is not in the correct format, e.g., 123.45),
the Int initializer returns an Int? It could be nil, or
it could contain an Int value
```

if statements and forced unwrapping

- You can use an if statement to see if an optional has a value, and only force unwrap it if it does.

```
let potentialNum:String = "123.45" // good luck!
let convertedNum:Int? = Int(potentialNum)

if convertedNum != nil {
    print("Our number is \(convertedNum!)")
} else {
    print("Oops, \(potentialNum) is not in Int format")
}

// output: Our number is 123
// output: Oops, 123.45 is not in Int format
```

Techy Aside: How to tell when an initializer returns an optional

- How can you tell when an initializer returns an optional?

- When reading the docs, look for init?(). This indicates that an initializer can fail (and return nil). Many classes use this, for example ...

- Int:

```
convenience init?(_ text: String, radix: Int = default)
// fails if the String is not formatted like an Int
```

- UIImage:

```
init?(named: String) // fails if it can't find the image
```

Optional Binding

- Optional binding lets you find out if an optional contains a value (i.e., it is non-nil), and if so, bind it to a (non-optional) constant.
- This is a convenience - you *could* check if the optional is nil using the if stmt. shown previously - but the pros use if let

```
  if let constantName = someOptional {
      statements
  }

// Note: If someOptional is SomeType?, constantName will
  be of type SomeType
```

Optional Binding @ Work

Rather than this

```
let potentialNum:String = "123.45" // good luck!
let convertedNum:Int? = Int(potentialNum)

if convertedNum != nil {
    print("Our number is \(convertedNum!)")
} else {
    print("Oops, \(potentialNum) is not in Int format")
}
```

do this

```
var potentialNum = "123,45"

if let convertedNum = Int(potentialNum) { // Int() returns an Int?
   print("x is \(convertedNum!)")          //convertedNum is an Int
} else {
    print("couldn't convert \(potentialNum), sorry")
}
```

The Nil Coalescing Operator, aka ??

- Consider two values, a and b, where a is optional and b is the same type as the value contained in a.
- a ?? b has the value of a!, if a is non-nil, or b, if a is nil

```
var luckyNumber:Int?
var actualNumber:Int = luckyNumber ?? 7 // 7

luckyNumber = 5
actualNumber = luckyNumber ?? 7     // 5
```

Implicitly Unwrapped Optionals (!)

- In some situations, it may be assumed that an optional will *always* have a value after it is set
- It is time consuming to use if or if let to check & unwrap, because we *know* it will have a valid value.
- To avoid this Swift defines **Implicitly Unwrapped Optionals**, created by following a type with a **!**
- With IUOs, don't use an ! to force the unwrapping: it happens implicitly, without you having to raise a finger

```
var w:Int?
w = 5
w = w! + 5
```

$$\downarrow$$

```
var y:Int!
y = 5
y = y + 5
```

Common Use of Implicitly Wrapped Optionals

- Often used during class initialization of ViewControllers' outlets.

- The ViewController is instantiated *before* it is connected to the objects in a storyboard

- Can define properties as implicitly unwrapped optionals, and not have to insist on assigning them values in init()

- Some programmers are not keen on using !, however. Read this, for instance.

Downcasting

- A variable of a particular type may refer to a subclass behind the scenes (for instance, a variable may be of type UIControl, but it may really be a *particular* UIControl, say UIButton)

- You can downcast using the type cast operators **as?** or **as!** to convert the variable to that type

- Downcasting may or may not succeed.

- as?, the conditional type cast operator, returns an optional, and should be used when you are not certain if the down casting will succeed.

- as!, the forced type cast operator, returns the indicated type (non-optional), and should be used when you are certain that the downcasting will succeed. Should it fail, a run time error will result.

An Uplifting Downcasting Example

```
// library is an array of Movie and Song objects, but
for item in library {
        if let movie = item as? Movie {
            print("Movie: \(movie.name), dir. \(movie.director)")
        } else if let song = item as? Song {
            print("Song: \(song.name), by \(song.artist)")
        }
   }

   // Movie: Casablanca, dir. Michael Curtiz
   // Song: Blue Suede Shoes, by Elvis Presley
   // Movie: Citizen Kane, dir. Orson Welles
   // Song: The One And Only, by Chesney Hawkes
   // Song: Never Gonna Give You Up, by Rick Astley
```

Optional Chaining

- A process for calling properties, methods and subscripts on an optional (an object reference) that might be nil
- If the optional is nil, the property, method or subscript call simply returns nil (rather than crashing)
- If the optional is not-nil, the property, method or subscript returns an optional value (even if the property, method or subscript returns a non-optional value)
- Multiple queries can be chained together - the entire chain fails gracefully

Optional Chaining

- Specify optional chaining by placing a ? after the optional value on which you wish to call a property, method or subscript if it is non-nil.

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

class Residence {
       var numberOfRooms = 1
}

let john = Person()
// the next line would cause a crash
let roomCount = john.residence!.numberOfRooms

// this would be OK: roomCount2 would be nil
let roomCount2 = john.residence?.numberOfRooms
// roomCount2 is nil
```

Exercises

1. Declare a variable x as an optional Int, and do not assign it a value.

2. Other exercises pending ...