Swift - Enums

Mobile Computing - iOS

Objectives

- Students will be able to:
  - explain the purpose of enums
  - create enums

Enums

- In C, enums are used to define named integer constants easily, to make programs more legible.

- Swift generalizes this: enums are used to **group related values into their own type**. For instance, you could have an enum called TrafficLight, with values .Red, .Yellow and .Green. A variable of type TrafficLight could take on only one of those three values.

- Enums ensure type safety. You *could* use an Int with values 0, 1 and 2, for Red, Yellow, and Green, but then you could inadvertently assign it a value 3. That cannot happen with enums

- Enums make programs more readable. Assigning .Red to a TrafficLight variable makes it *immediately* clear what its value is, whereas assigning a 0 means ... well, you'd have to look that up.

- Apple uses enums all the time in their APIs. Anytime you find yourself with a small set of values, use an enum instead of an Int. Your programs will be more legible and maintainable.

Enums By Example

```
enum Day{case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}

enum PointsOfCompass {
    case North
    case East
    case South
    case West
}
var startOfWeek:Day = .Monday // or Day.Monday
let startingDirection:PointsOfCompass = .North // or PointsOfCompass.North

var favoriteDays:[Day] = [.Monday, .Friday, .Saturday]

switch(startingDirection){
    case .North: print("It'll get chilly up here")
    case .East:print("Hey, look, it's the Atlantic Ocean")
    case .South:print("AC ... we need AC")
    default:print("We're lost here")
}
```

1. Enums start with the keyword **enum**, the name, and, in { }, the enum cases. The cases can be listed on one line, separated by commas, or each case can be on its own line (with no terminating comma)
2. Unlike in C, by default enums do not have an integer value associated with them

Type Safety in Action

```
enum TrafficLight {
    case Red
    case Yellow
    case Green
}

// anything other than .Red, .Yellow or .Green won't compile -- we can't make a
mistake, hence the term "type safety"

var trafficLight:TrafficLight = .Red


var BadTrafficLight:Int // we promise to assign 0 for Red, 1 for Yellow 2 for Green

BadTrafficLight = -1 // But the compiler will not complain about this.
```

Raw Values

- We can assign values of a specific type to an enum. Their type must be declared, and accessed using .rawValue

must declare enum's of that type

can assign raw values to enums

```
enum TrafficTicket:Double {
        case Speeding = 50.00, Parking = 25.00, DUI=100.00
}

var violation = TrafficTicket.Parking

// Bonus factoid: Here's how to format a number ...

var ticketNote = String(format:"Fine: € %5.2f",violation.rawValue/1.63)
// ticketNote has the value "Fine: € 15.34"
```

Raw Values

- If you assign an Int as a raw value, Swift will assign subsequent values by incrementing by 1, until another value is explicitly assigned

```
enum Clothes : Int {
    case hat = 1
    case sunglasses          // rawValue = 2
    case ordinaryGlasses     // rawValue = 3
    case necklace = 10
    case tie                 // rawValue = 11
    case shirt,pants,shoes // 12, 13, and 14, respectively
}
```

Making Enums From Raw Values

- You can extract a raw value out of an enum: you can also use a raw value to initialize an enum (

```
enum Clothes : Int {
    case hat = 1
    case sunglasses                 // rawValue = 2
    case ordinaryGlasses            // rawValue = 3
    case necklace = 10
    case tie                         // rawValue = 11
    case shirt,pants,shoes           // 12, 13, and 14, respectively
}

let clothes:Clothes? = Clothes(rawValue:10) // optional -- why is that?
print(clothes!) // output: necklace
```