MOBILE DEVELOPMENT

OBJECT ORIENTED PROGRAMMING

LEARNING OBJECTIVES

- Define object-oriented programming.
- Identify and apply object oriented principles:
  - inheritance,
  - polymorphism,
  - encapsulation.
- *Differentiate between classes and structs.*
- *Create protocols and apply them to classes, structs, and types.*

CLASSES

WHAT IS THIS ‘CLASS’ KEYWORD?

- A basic building block for *custom, user-defined types*.
- A bundle of *state* and *behavior* that comprise a template for a type:
  - **Variables** (state, in this case known as ‘properties’)
  - **Functions** (behavior, in this case known as ‘methods’)
- One can create “instances” of classes.
  - “Dog” could be a class; “Toshi,” a specific dog, is an instance.
- We’ll use the word “object” somewhat interchangeably with “instance.”

XCODE DEMO: CREATING CLASSES

SYNTAX: DEFINING A CLASS

```
class Dog {
    /* some variables and functions */
}
```

SYNTAX: DEFINING A CLASS

```
class Dog {
    init() {
        println(“Here I am!”)
    }
}
```

SYNTAX: CREATING AN INSTANCE

This creates a single instance of the class “Dog” and assigns it to a variable.

```
var toshi = Dog()
```

SYNTAX: DEFINING A CLASS WITH A METHOD

```
class Dog {
    init() {
        println(“Here I am!”)
    }

    func bark() {
        println(“Woof!”)
    }
}
```

SYNTAX: CALLING A METHOD

Using “dot notation,” we can reference the method (function) “bark” within the instance and call it like any other function.

```
var toshi = Dog()
toshi.bark()
```

SYNTAX: DEFINING A CLASS WITH A PROPERTY

```
class Dog {
    var name : String

    init() {
        self.name = “Toshi”
        println(“Here I am!”)
    }
}
```

Note the keyword “self”, which enables an instance to refer to itself.

CLASSES: MORE ON INITIALIZATION

- Classes have properties (variables) which *must equal a value* at the time the object is initiated.
  - **Important!** This means that **every** instance variable must either be Optional or be assigned a default value during initialization.
- Classes can specify custom initializers that take parameters.

SYNTAX: ACCESSING A PROPERTY

Using “dot notation,” we can also reference a property that contains data specific to the instance.

```
var toshi = Dog()
println(toshi.name)
```

SYNTAX: DEFINING A CLASS WITH A CUSTOM INITIALIZER

Initializers can accept arguments. We use those arguments in various ways; one way is to initialize the non-Optional properties of the instance:

```
class Dog {
    var name : String

    init(name:String) {
        self.name = name
        println(“Here I am!”)
    }
}
```

SYNTAX: ACCESSING PROPERTIES

We can declare two independent instances of the class “Dog”:

```
var myDog = Dog(name:”Toshi”)
myDog.name

var mySistersDog = Dog(name:”Layla”)
mySistersDog.name
```

These instances have different values for their “name” property.

SYNTAX: A CLASS WITH OPTIONAL PROPERTIES

Optional properties are declared like Optional variables, with a ?

```
class Dog {
    var name : String
    var age : Integer?

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

They don't have to be initialized, but they contain the value nil if they aren't.

SYNTAX: A CLASS WITH OPTIONAL PROPERTIES

```
class Dog {
    var name : String
    var age : Integer?

    init(name:String) {
        self.name = name
    }
}
var myDog = Dog(name:”Toshi”)
myDog.age = 2
```

SYNTAX: FUNCTIONS THAT ACCEPT CLASSES AS TYPES

Here we can write a function that takes a Dog as the type of it's argument.

```
var myDog = Dog(name:"Toshi")
func prettyPrintDog(oneDog:Dog) {
    println("\(oneDog.name) is a Dog!")
}
prettyPrintDog(myDog)
```

Note that we can use .name to access that property of oneDog.

SYNTAX: EXTENDING A CLASS + INHERITANCE

We can make classes from other classes and give them new behaviors and state. This enables us to extend the functionality of our classes and empower us to customize UI elements.

```
class Shiba : Dog {
    var temperament : String = “stubborn”
}
```

We say that Shiba “inherits” from Dog. It has all the properties and functions from Dog, plus an additional property, temperament.

SYNTAX: EXTENDING A CLASS + INITIALIZERS

When we extend a class, sometimes the initialization process needs to be different, especially if we have new properties that need to be initialized:

```
class Shiba : Dog {
    var temperament : String

    init(name:String) {
        self.temperament = “stubborn”
        super.init(name:name)
    }
}
```

SYNTAX: EXTENDING A CLASS + INITIALIZERS

The “super” keyword refers to the superclass Dog. So super.init(name:…) is the initializer defined inside the class definition of Dog.

```
init(name:String) {
    self.temperament = “stubborn”
    super.init(name:name)
}
```

SYNTAX: POLYMORPHISM

Now, we can use Shiba wherever we can use Dog:

```
var myShiba = Shiba(name:”Toshi”)
prettyPrintDog(myShiba)
```

In this case, we can pass a Shiba into the prettyPrintDog function. All code that has been written with Dog in mind can also take Shiba, since Shiba has all the functions and properties of Dog, and Dog is a superclass of Shiba.

This phenomenon is called “polymorphism.”

CLASSES IN APPS

HOW DOES THIS TIE INTO IB?

- We use IB to set up various **classes** of Controllers, and the **Segues** that they use to connect with each other.
- Uses these storyboards to create an **instance** of the first controller **class** when your app starts. That **instance** is what's displayed on screen.
- When you use a segue to go to a new view controller **class**, a new **instance** of it is created and navigated to.
- Multiple **instances** of the same class can exist.
- Each of those instances have their own **variables** and **functions.**

REVIEWING CLASSES

WHAT IS A CLASS? AN OBJECT?

- A class is a logical grouping of state and methods that “encapsulate” an entity.
  - e.g. a view, a device, an app, a view controller, an array
- A class variables and methods.
- There can be many instances of classes, each of which has “instance” methods and state.
  - “Toshi” is an instance of the class “Dog.”
- “Object” is equivalent to “instance.”

WHAT IS A CLASS? AN OBJECT?

- Examples of classes from UIKit:
  - UIViewController
  - UIView
  - UILabel
  - UITextField

SYNTAX REVIEW: USING CLASSES

```
// Create a new instance of type SomeClass
var myVariable = SomeClass()
// Get a property from an instance
var someProperty = myVariable.property
// Call a method on an instance
myVariable.someMethod()
// Call a method that returns a value
var someReturnValue = myVariable.someOtherMethod()
```

MORE ABOUT OBJECT-ORIENTED CONCEPTS

OO CONCEPTS : ENCAPSULATION

- Classes are a bundle of related state and behavior that are separate from other classes.
- The state of one instance is encapsulated from the state of another instance
- State and behavior can have limited visibility
  - Though we aren't really going over this in class

OO CONCEPTS : ENCAPSULATION

- **Public** access enables entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use public access when specifying the public interface to a framework.
- **Internal** access (default) enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app's or a framework's internal structure.
- **Private** access restricts the use of an entity to its own defining source file. Use private access to hide the implementation details of a specific piece of functionality.

OO CONCEPTS : INHERITANCE

- Classes can “inherit” from one other class (a ‘superclass’).
- A class inherits its methods and state from superclass.
- The class that inherits from another class is its subclass.
- A class can only have one superclass.

OO CONCEPTS : POLYMORPHISM

- A method that takes a class (e.g. Animal) can also accept any of its subclasses
- Example:
  - Animal is a class, and Dog is a subclass of Animal
  - walkAnimal(animal: Animal) {} is a function that walks an animal
  - Because walkAnimal() can accept an Animal or a Dog, because dog **is an** Animal

EXERCISE: SIMULATE A GAME

SIMULATE A GAME

- Make three classes, ‘Player’, ‘GoodPlayer’ and ‘BadPlayer’.
  - Player has an ‘attack’ method, which returns an Int, the amount of damage the attack does.
  - GoodPlayers and BadPlayers, when they attack, attack with different magnitudes (between GoodPlayer and BadPlayer, or [BONUS] different from attack to attack).
  - Players also have a health property (an Int that defaults to 100) and an ‘isAlive’ method (returns a Boolean). A Player is alive if their health is above 0.
- Create a ‘Match’ class, which takes two players during initialization.
  - It has a ‘playGame()’ method, which pits each player against each other, alternating attacks until one of the players is no longer alive. At the end of the match, return the winner.
  - Hint: This means you’ll need a way to have the attack of one player affect the health of the other.
- Pit one GoodPlayer against a BadPlayer.
- Bonus: Give players names, print those out before they match.

STRUCTS AND PROTOCOLS

STRUCTS

WHAT IS A STRUCT?

- A struct is, like a class, a logical grouping of state and methods that encapsulate an entity
  - e.g. a rectangle, an integer, an array
- A struct has variables and methods
- There can be many instances of structs, each of which has methods and state

SYNTAX REVIEW

```
// Creates a new struct, MyStruct
struct MyStruct {
    /* struct definition */
}
```

Accessing variables and calling methods on structs is the same as classes

WHAT'S THE DIFFERENCE BETWEEN A STRUCT AND A CLASS?

- Instances of a struct are **values**, which are copied as they are passed around.
- Instances of a class are **references**, which are not copied as they're passed around.

STRUCTS - DEMO - VALUE VS REFERENCE

PROTOCOLS

- A group of methods that a class has, encapsulated into its own entity.
  - Methods can be required or optional.
- Classes can ‘meet’ as many interfaces as they’d like.

SYNTAX REVIEW

```
// Creates a new protocol
protocol MyProtocol {
    /* protocol definition */
}
// Creates a new class that meets a protocol and inherits
from a superclass
class MyClass: MySuperclass, MyProtocol {
    /* class definition */
}
```

DEMO - PROTOCOLS