GENERAL ASSEMBLY

MOBILE DEVELOPMENT
OBJECT ORIENTED PROGRAMMING

Tedi Konda
Executive Director, Technology, Unison

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

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
  - “Tedi”, the instructor, is an instance of the class “Human”
- Object == Instance
- There are also class methods

WHAT IS A CLASS? AN OBJECT?

- Instance methods can access instance variables
  - Class methods can not access instance variables
- Examples of classes:
  - UIViewController
  - UIView
  - UILabel
  - UITextField

CLASS DEMO

EXERCISE

- In pairs, create a class called “Animal”
  - It should have two strings as instance variables, species and name
  - It should have one method, stringRepresentation, which should return
    - “The animal is a {species}, its name is {name}
- Create a view controller with two buttons and a text field
  - One button creates a cat, the other creates a dog. You pick the name
- When the button is clicked, the label should display the results of stringRepresentation for a new instance of Animal

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 called a subclass
  - A class can only have one superclass
    - Why?

INHERITANCE DEMO: ANIMALS

INITIALIZATION

- Classes have 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

INITIALIZATION DEMO - ANIMAL & UIVIEWS

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

PEER PROGRAMMING - A GAME

- Work in a playground
- One person should make three classes, ‘Player’, ‘GoodPlayer’ and ‘BadPlayer’
  - Player has an ‘attack’ method, which returns a tuple (message: String, damage: Int). Message is the message that the player says during the attack, and damage is the amount of damage it does
  - Both good players and bad players have some (>=2) possible attacks. Good and bad players have different possible attacks, they are performed randomly when attack is called
  - Players also have a health integer (default to 100), and an isAlive method (a player is alive if their health if above 0)
- The other person creates a ‘Match’ class, which takes two players during initialization
  - It also has a ‘playGame()’ method, which pits each player against each other, alternating taking turns until one of the players is no longer alive. At the end of the match, print out the winner
- Put one GoodPlayer against a BadPlayer, look at the printed results!
- 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

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

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

VALUE & REFERENCE - DEMO

PROTOCOLS

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

DEMO - PROTOCOLS

PAIR PROGRAMMING

- Create an app that has a view controller with a table view
- Make that view controller both the delegate and data source for the table view
- Create an array of ten Players (good or bad) when the view comes into view. Print out those players in the table view.