Notes From the Class

**Protocols and Extensions**

Remember **interfaces** from the pre-bootcamp material? Now, we will take a look at **protocols** in Swift.

According to the [Swift Documentation](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html), "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."

The power of using protocols is that they define the connection between different parts of your code, **without providing implementations.**

*Any type that satisfies the requirements of a protocol is said to conform to that protocol.*

We describe what properties and methods something must have and then we then tell Swift which types use that protocol (conforming to the protocol).

**Multiple protocols can be listed, and are separated by commas:**

struct SomeStructure: FirstProtocol, AnotherProtocol {

// structure definition goes here

}

**Protocol Example**

Take a look at the following example using Swift. We've already gone over protocols and interfaces with Java. But this refresher might help!

*Remember: You cannot create instances of protocols. You can only conform to them.*

protocol Phone {

var name: String { get set }

var color: String { get set }

func sendMessage()

}

a **protocol** does not care about the implementation of its methods. It only cares that the method is implemented in the type conforming to the protocol.

class Iphone: Phone {

var name = "Best OS"

var color = "black"

func sendMessage() {

print("Using iMessage!")

}

}

class Android: Phone {

var name = "Subpar"

var color = "black"

func sendMessage() {

print("Android Message System!")

}

}

**Protocols Give Us Power**

**Did you know?**

Protocol-oriented programming is the idea of designing your app architecture as a series of protocols and using protocol extensions!

Later on, we will be learning about **Dependency Injection**, which uses protocols to create implementations of certain types.

**Extensions**

Extensions make our lives easier, and they add a great deal of power to our programming ability!

According to the [Swift Documentation](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html), we can use extensions to add new functionality to an existing Swift class, struct, enumeration or protocol type!

This includes the ability to extend types for which you don’t have access to the original source code.

Declare extensions with the extension keyword:

extension SomeType {

// new functionality will go here

}

**Let's take a simple example:**

Swift will not allow you to add stored properties in extensions, so we have to use **computed properties** instead. For example, we could add a new isEven computed property to integers that returns true if it holds an even number:

*Remember: A computed property computes its property upon request*

extension Int {

var isEven: Bool {

return self % 2 == 0

}

}

let newNumber = 7

print(newNumber.isEven)

# Type Casting

According to the [Swift Documentation](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html),

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.

*Type casting in Swift is implemented with the is and as operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.*

Hmmm, what's a type again?

As Swift developers, we use **types**, (classes and structs), to represent different kinds of data.

Some examples:

* Int for integers
* Double for decimals
* String for text and words
* UIButton for UI Element

Remember that Swift is strongly typed, and that means that every variable needs a type, and once you decide which type the data is, it can't be changed.

### What's the catch?

But, with type casting, we can treat an object just like another type!

The object doesn't change, but the way we describe it with a type changes.

### A Type Casting Example

Take a look at the following classes:

class Car { }

class Acura: Car { }

class Lexus: Car {

func revEngine() {

print("Vroom!")

}

}

Now let's create an array of affordable luxury cars.

let luxuryCars = [Acura(), Lexus(), Acura(), Lexus()]

Swift can see both Acura and Lexus inherit from the Car class, so it uses **type inference** to make luxuryCars an array of Car.

If we want to loop over the luxuryCars array and ask all the cars to rev their engine, we need to perform a **typecast**:

* Swift will check to see whether each car is a Car object, and if it is we can then call revEngine().

This uses a new keyword called as?, which returns an optional: it will be nil if the typecast failed, or a converted type otherwise.

And we could write a for loop like this:

for car in luxuryCars {

if let acura = car as? Acura {

acura.revEngine()

}

}

# Laziness

What is lazy? It puts off work until you need it ... or this could be called **just in time** or (as soon as you need it)

#### Examples of when we need laziness

* A variable needs to know about other variable values

An example is initializing an object and during that time, you want to have a variable, but the variable doesn't know about the other variables yet.

* Delay expensive tasks

If something takes a long time to process, we want to take care of this **lazily** because we only want to do that heavy process when we need it.

### Lazy Example

Let's take the example of when a variable needs to know about other variable values:

I want an introduction for our player. But, the introduction **doesn't know about the name, team, and position yet** since this going on during initialization.

We are marking introduction as **lazy** and ... we are essentially running a function here (a closure)

import UIKit

struct Player {

var name: String

var team: String

var position: String

lazy var introduction = {

return "Now entering the game: \(name), \(position) for the \(team)"

}()

}

I want the introduction to be a property on player so I can call player.introduction and spit out the introduction string!

The reason we have to make this lazy is because during initialization, we don't know about name, position, and team.

Initialization has to complete before I have that information.

**Let's create a player:**

var jordan = Player(name: "Michael Jordan", team: "Bulls", position: "Shooting Guard")

print(jordan.introduction)

**Run the code!**

So could we use a computed property instead of a lazy variable here?

Yes, but that wouldn't be the best for this situation because it would be re-computed every time we access the property, for example when calling jordan.introduction

Lazy actually stores the value on the object once it has it, check out the example below:

struct Calculator {

static func calculateGamesPlayed() -> Int {

var games: [Int] = []

for i in 1...5000 {

games.append(i)

}

return games.last!

}

}

struct Player {

var name: String

var team: String

var position: String

// create a property on player

var gamesPlayed = Calculator.calculateGamesPlayed()

lazy var introduction = {

return "Now entering the game: \(name), \(position) for the \(team)"

}()

}

var jordan = Player(name: "Michael Jordan", team: "Bulls", position: "Shooting Guard")

print(jordan.introduction)

Now, every time we initialize a player, it's going to run the super long function. It will iterate 5000 times. It gets called when we initialize the jordan player.

Instead of calculating during the initialization, we calculate that lazily (only do 5 second calculation when we need it)

lazy var gamesPlayed = {

return Calculator.calculateGamesPlayed()

}()

* Now, when we initialize the player, it's not going to count to 5000, and our player will be initialized.

That's the power of lazy!

print(jordan.gamesPlayed)

Now, it actually runs!

### Computed Properties vs Lazy

A **computed property** is similar, it's not going to compute gamesPlayed until we actually access it.

But the difference with lazy is that we actually store that value. (5000 for games played)

**Computed Property**

Let's take an example of using a computed property, such as the following:

// computed property

var gamesPlayed : Int {

return Calculator.calculateGamesPlayed()

}

Now if I do print(jordan.gamesPlayed), every time I try to access it, it runs that calculation again.

If we do it 3 times, our calculation runs 15,000 times.

Lazy calculates it once, and **stores** it on our player object . A computed property runs the calculation every time you access it.

Let's say we created 20 players... Lazy comes in handy when we are creating multiple objects.

#### Lazy Pitfall

Why not make everything lazy?

Lazy isn't thread safe. If you are using a lot of multithreaded code, we have to be careful.