202: Swift (Demo)

Closures (10 mins)

We're going to do this demo in a playground. Open the **Demo.playground** from the **1-Starter** folder.

> *Demo.playground*

This is an example of a function. It looks up how often a certain value appears in an array. Exactly what it does isn't really important here, but as you can see it's pretty simple.

If I put it inside a class, it is now a method. We can remove the `array` parameter because the class has its own array.

So there really isn't any difference between a function and a method in Swift.

> *myObject.countOccurrences(999)*

It has a name, a list of parameters, the thing behind the arrow is the return type, and a body with source code statements.

There's another way in Swift that you can bundle up code like this and that's with a closure. A closure is really [this part]. It's everything without the function name.

Let's copy-paste this and then assign it to a variable.

> *let c: [paste]*

We do need to make a small change. The stuff between the curly braces becomes the value of the variable, so it needs an equals sign.

This is really like any other variable declaration: here's the name, colon, the type, equals, and the value. Except that the type is really the type of a function [it's the same thing as this up here], and the value of the variable is a block of code.

A closure is really the same as a function without a name. That's all there is to it.

There are a few small syntax differences, though. You can see that the playground still gives an error. That's because the closure, like a function, needs to know what its parameters are.

To get the parameters into the closure, you have to write it like this. The names of the parameters and their types come before the special "in" keyword, and then the code follows after that.

We can actually get rid of these two labels here [in the type]. For the type signature, it doesn't really matter what the labels are. But you do need to know the labels inside the closure, otherwise the code doesn't know what these things [the variables] are.

To call this closure, you'd use the exact same syntax as calling a function. And there you have a closure in action.

So why would you use a closure? Well, a closure is like an object, and you can treat this as any other kind of object. You can store it in a variable, as we did here, you can pass it to another method, you can put the closure into an array, and so on. Anything you can do with an object, you can do with a closure.

So let's see an example of passing closures around.

Add performClosure method to MyClass.

```
func performClosure(value: Int, closure: (Int, [Int]) -> String) -> Stri
ng {
  return closure(value, array)
}
```

This method takes a closure as parameter and then calls it on its own array. Again, this is the type of the closure. You can see that it's the same as down here.

Now when we pass our closure "c" to this method, we should get the same result.

myObject.performClosure(999, closure: c)

Of course, we can also pass another closure. This time we're writing it inline. This one adds the value to each element of the array and then outputs that as a string.

```
myObject.performClosure(100, closure: { (value: Int, array: [Int]) -> St
ring in
  var newArray = [Int]()
  for element in array {
    newArray.append(element + value)
  }
  return newArray.description
```

```
})
```

As you can see, closures are a convenient way to pass a block of code around. You don't have to wrap it in a function or a method; you can just write it inline.

By the way, you can simplify this a bit by using trailing syntax. If a closure is the last parameter in a method call, you can put it behind the method call. That often makes the code easier to read. That's something you'll see a lot in Swift code.

Speaking of simplification, I don't know about you, but this bit [the parameters] is not very readable. Fortunately, Swift has type inference so we can remove the types, including the return type.

You can use `$0` and `$1` and so on to refer to the parameters if you don't want to give them real names. You sometimes see that in really short closures, but in general I recommend you name the parameters.

There's even more to simplify. Up here [in the class], this may get a bit hard to read with all those parentheses, so you can use a so-called typealias to give this type signature a name.

```
typealias TransformArray = (Int, [Int]) -> String
```

> *Change `performClosure` to use this alias. Also for `c`.*

That works a bit like a typedef in Objective-C.

Tic-Tac-Toe (5 mins)

I want to switch to the Tic-Tac-Toe app for a second now.

The Tic-Tac-Toe app also contains an example of a closure. Closures are often used as completion handlers for iOS APIs.

In ViewController.swift, an alert gets put up when there is a winner or a draw.

> *Xcode: UIAlertController code from starter project*

Each button on the alert is represented by a `UIAlertAction` object, and the code that gets performed when the button is tapped is given by a closure. That's this bit right

here. Closures are usually quite easy to spot because it's a bunch of source code between curly brackets.

As you can see, it uses the trailing syntax because the closure sits behind the method call.

Since closures are really a chunk-of-code as-an-object, they are often used to store code for later use. In this case, the `UIAlertController` holds on to the closure until the user taps the button.

The closure has one parameter, a reference back to the `UIAlertAction` object, and it returns nothing. That's what `Void` means. You can also write `Void` as `()`. These mean the same thing. Sometimes there are just too many parentheses, and then `Void` is a bit easier to read.

We can simplify the parameter here by removing the type name as we did earlier, but as the `action` parameter is never used in this closure at all, you can type a `_`. This is the wildcard symbol in Swift and you use it to tell the compiler that you want to ignore something.

(You can't leave out the "in" bit altogether, because then the thing between the curly brackets no longer has the right type.)

I mentioned earlier that closures are the same as regular functions and methods, but there is an important difference. Closures can capture variables. If you use a variable from an outer scope inside the closure, it will copy that variable so that it is still available when the closure is performed, which may be much later. And the original variable may not exist anymore.

This particular closure calls the `reset()` method. But because that is an instance method of `ViewController`, the closure will capture `self`, which is a reference to that `ViewController`. Swift wants you to make this explicit, so if you forget `self`, the compiler will tell you to add it back in.

So inside a closure, if you're using an instance variable or method, you always need to write `self`.

The important thing to remember is that if the thing that's being captured is a reference type, such as an object from a class, then this can lead to unwanted ownership cycles and memory leaks, so be careful with it.

To avoid capturing such objects using a strong reference, you can specify a so-called capture list. [weak self] Now the value of `self` is captured as a weak reference. But of course that makes it an optional. You don't really need `weak self` here, because the

ownership cycle is only temporary, but it's a thing to keep an eye on.

This was a quick overview of closures. You'll see them again in the lab and the challenges, to get some more experience with them. They're very handy and you're going to be using them a lot when you're writing Swift.

Generics (5 mins)

The second topic of this talk is generics – the fancy term for this is *parametric polymorphism*. If you've done C++, these are just like C++ templates. Many other languages offer generics as well, but if you've done mostly C or Objective-C then generics will probably be new to you.

So what are they?

Well, in the example in the playground we have so far, the `countOccurrences()` function works on arrays of integers only. What if we also need to do this on an array of strings? Then you'd have to rewrite this function specifically for strings. It's possible, but you end up with a lot of duplicate code. Generics let you abstract the specific type away.

We can make this function generic like so.

```
func countOccurrences<T>(value: T, array: [T]) -> String {
```

These angle brackets following the function name tell Swift that this is a generic function. The T is a placeholder for the actual type name. I also replaced all the Ints with T.

T is not a real type name, but a placeholder for the actual type that will be filled in later. T can literally stand in for any possible type.

You could call this anything you want, for example, PlaceholderType. But T is the convention. (And if you had more than one type you wanted to make generic, you'd call the second one U, the third one V, and so on.)

Note that there is an error on this line. Swift does not know what to do with the equals operator, because not all types will have this operator. We know that Int and String do, but your own types may not. So T can't actually be "all possible types" in this case.

We need to restrict the possible types that T can stand in for.

```
<T: Equatable>
```

This is a type restriction. It tells Swift that T is now limited to any class or struct or other type that conforms to the `Equatable` protocol. This is one of the built-in protocols from the Swift standard library. It makes it possible to have the == operator.

To prove this really works with Strings, let's add an array for that as well.

```
let stringArray = ["A", "B", "C", "B", "D", "E"]
countOccurrences("B", stringArray)
```

So now you've made this function generic for any type that conforms to Equatable. Without having to write any new code you can reuse this function across many different data types. Swift is smart enough to figure out which version to use based on type inference.

Of course, you've already seen this at work because Swift's array and dictionary are exactly that. If you want to use an array you need to specify the type of objects that go into the array. Likewise for a dictionary.

Here we wrote `[Int]` but that's really shorthand for `Array<Int>`. Notice the angle brackets again? If we look at the definition of Array in the Swift library, you'll see that it also uses this notation with T. In other words, Array is a generic type. (By the way, you can look up these Swift types by holding down Cmd and then clicking on them.)

And in our generic function you could also have written `Array<T>`.

OK, that's a brief intro into generics. You'll get back to this in the lab where you'll learn how to make your own generic type. But first I want to mention enums.

Enums with Associated Values (8 mins)

If you'll remember from the previous talk, the `Player` type is defined as a enum with two cases:

```
enum Player {
  case X
  case O
}
```

An enumeration is a list of mutually exclusive values, so a player can be either X or it

can be O but it cannot be both, nor can it be anything else.

This pretty much is the same as enums in Objective-C, except that in Swift you can also add methods to enums.

You can even give each of these case labels a value, for example:

```
enum Player: Int {
  case X = 0
  case O = 1
}
```

When you do that, you also have to specify the type of the enum, in this case `Int`. These values don't have to be integers; you can use pretty much anything you want, for example strings:

```
enum Player: String {
  case X = "Player X"
  case O = "Player O"
}
```

This is already a lot more powerful than Objective-C enums, which are limited to just integers.

These values we've just given the labels are also called the "raw values". So you can do this:

```
let p = Player.X
println(p)
```

This just says "(Enum Value"), which is not very useful. So instead you do this:

```
println(p.rawValue)
```

And this will print out the value you've given that case.

Often you'll use `switch` to read the values of an enum, like so:

```
switch p {
case .X:
  println("Player is X")
case .O:
  println("Player is O")
}
```

So far so good, nothing really special here. However, enums in Swift have this really

cool feature called *associated values*.

Now, don't confuse that with these values here. The values that you see here are fixed at compile time; they will never change. That's useful for when the enum labels are really human-readable names for existing constants, such as with the enums from the SDK, such as `UIInterfaceOrientation`.

Associated values, on the other hand, aren't known until runtime.

For example, let's assume the game is now online multiplayer and players can pick their own usernames. We can associate the username with the Player object as follows. First, let's get rid of these strings, and then:

```
case X(username: String)
```

I'm just doing it for the .X player now but it works the same way for the other case.

Whenever the value of this enum is X, it also has a username string to go along with it. So you can't do this anymore, you have to always give `Player.X` a username.

```
let p = Player.X(username: "Steve")
```

Because this value "Steve" is only associated with this particular object, you can create another one with a different name:

```
let t = Player.X(username: "Tim")
```

They both use the X case, but each has different data associated with it.

To read this data, you can't use `rawValue` anymore. Instead, you can slightly modify the switch:

```
case .X(let value):
  println(value)
```

And if we change it to `switch t`, it will print "Tim".

The label `value` doesn't actually have to be the same as what it is called in the enum. (This is an example of switch's pattern matching features.)

You can also associate more than one piece of data, for example for the Tic-Tac-Toe olympics we want to know where the player comes from, so we add the country:

```
case X(username: String, country: String)
```

(Also update the code that creates the Player objects.)

Notice how now the println actually prints out a tuple. To split this up into two fields, we can do:

```
case .X(let username, let country):
```

The `switch` statement is very powerful, it has all kinds of pattern matching you can do on it. In this case, these two labels correspond with the associated values. You can even do stuff like `where country == "USA"`.

You can also write it like this:

```
case let .X(username, country):
```

Or even:

```
case let .X(_, country):
```

if you want to ignore a particular field. Again, you can use the wildcard symbol, the underscore, to let the compiler know that you want to ignore something.

Enums with associated values are one of the most powerful concepts in Swift. It's so powerful that one of the core features, optionals, is nothing more than such an enum.

Show definition (Cmd-click on import Swift, search for "enum optional").

An optional is an enum with two possible cases: None, for when the value is nil, and Some for when it isn't. As you can see, the actual value is associated with the Some case. This uses generics because you can make anything into an optional.

So when you write:

```
var s1: String? = "Not nil"
```

what really happens is that the compiler turns this into:

```
var s2: Optional<String> = .Some("Not nil")
```

It's the exact same thing (`s1 == s2`). The stuff with the question marks and so on is

just syntactic sugar to make optionals a little easier to write.

So that's enums with associated values. They are really cool and I could easily spend the entire session talking about them if we had time for it.

Remember what we did here, because it comes back in the challenge.

Great, I hope this gave you some insight into closures and what they are good for. You've also seen how to use generics in action.

You are ready to move on to the lab. You can find the instructions in the **3-Lab** folder. You've got about 15 minutes to work your way through the assignment, and then we'll move on to the challenge.

If you get stuck or you have any questions, please let me know.