GENERAL ASSEMBLY

# MOBILE DEVELOPMENT

## INTRO TO FUNCTIONS

William Martin
Head of Product, Floored

Angel X. Moreno
EIR, Developer

# LEARNING OBJECTIVES

- Identify functions and implement best practices
- Be able to call and define functions that take parameters
- Be able to use the returned value from a function
- Understand what returning from a function does

# REVIEW LESSON 3

# INTRO TO PROGRAMS

# CONTROL FLOW

# CONTROL.PLAYGROUND

# CONTROL FLOW

- Programs are executed one line at a time, but it's not useful to execute all lines of code all of the time.
- Conditional statements leverage Boolean expressions to begin to define the logic of our apps. We can execute some code under certain conditions, and other code under other conditions.
- We can start to reason like this:
  - e.g. "If the temperature is less than or equal to 32 degrees, show a freezing icon, otherwise, show water drop icon."
- Also, we can start to leverage a computer's automation abilities by using loops.
  - e.g. "Keep executing this code as long as the temperature is less than 32."

# CONTROL FLOW - CONDITIONALS

Conditional statements, or “if-else” statements, look like this:

```
if temp <= 32 {
    // This “block” is executed if the condition is true.
    // Show a freezing icon.
} else {
    // And this “block” if false.
    // Show a water drop icon.
}
```

# CONTROL FLOW - CONDITIONALS

Conditional statements can contain multiple blocks or clauses, using “else if”:

```
if temp <= 32 {
    // Show a freezing icon.
} else if temp >= 212 {
    // Show a boiling water icon.
} else {
    // Show a water drop icon.
}
```

# CONTROL FLOW - WHILE LOOPS

The simplest kind of loop, while loops execute a block of code repeatedly as long a given condition is true.

```
var sum = 0
while sum < 50 {
    sum += 10
}
println(sum)
```

# CONTROL FLOW - FOR LOOPS

Strangely named, “for-loops” use conditionals to continue executing code given a conditional and a variable that is used for counting.

```
for (var temp=0; temp<=32; temp++) {
    // Do something here.
}
```

# CONTROL FLOW - FOR LOOPS

```
for (var temp=0; temp<=32; temp++) {
    // Do something here.
}
```

1. The loop declares and initializes a variable (temp),
2. checks the conditional, and if it's true,
3. executes the block of code within the braces, then
4. calls the incrementing expression (temp++)
5. checks the conditional again, etc.

# CONTROL FLOW - CONTROL TRANSFER - BREAK

```
let toCheck = 289
for (var i=2; i<toCheck; i++) {
    println(i)
    if toCheck % i == 0 {
        println(“composite!”)
        break
    }
}
```

The “break” statement aborts from the for loop.

Advanced students: make this more efficient. Write as a while loop.

# CONTROL FLOW - CONTROL TRANSFER - CONTINUE

```
let toCheck = 289
for (var i=2; i<toCheck; i++) {
    if i % 2 == 0 { continue }
    if toCheck % i == 0 {
        println(“composite!”)
        break
    }
}
```

The “continue” statement skips everything after it in the block, but continues executing the loop.

# OPEN OPTIONALS.PLAYGROUND

# OPTIONALS AND NIL

- `nil`
  - A value that represents no value.
- Optional - a type that represents `nil` or a value of another specified type
- Syntax:
  - `var [symbol] : [type]?`
- Example
  - `var name : String?   // initialized as nil`
  - `var name : String? = “Toshi”`

# OPTIONALS AND NIL

- Why use Optionals?
  - Sometimes we need a variable before we get a chance to give it a real value.
  - e.g. Imagine a web request that takes some time. We need a place to put the response to that query, but we won't know what the response is until the request is done.

# INTRO TO FUNCTIONS

# WHAT IS A FUNCTION?

- A function is a series of repeatable steps that, at some point, ends.
- Optional input and output.
- Multiple inputs and outputs are possible, as needed.
- Functions can contain variables visible only inside the function.
  - In general, a variable is only visible in the braces in which it was defined.

# WHAT IS A FUNCTION?

- Let's say we want to run the same few lines of code in multiple situations. We might wrap those lines of code into a function, let's call ours *sayHello:*

```
func sayHello() {
    println("Hello!")
}
sayHello()
```

# SYNTAX - SIMPLE FUNCTIONS

- Defining a function:

```
func sayHello() {
    /* our code here */
}
```

- Calling: `sayHello()`
- This runs all the code within the above function, then continues to the next line of code.

# SYNTAX - FUNCTION WITH ONE PARAMETER

```
func sayHello(name:String) {
    println(“Hello! \(name)”)
}
sayHello(“Toshi”)
```

# SYNTAX - FUNCTION WITH TWO PARAMETERS

```
func say(what:String, name:String) {
    println(“\(what)! \(name)”)
}
sayHello(“Hello”, name:“Toshi”)
```

# SYNTAX - FUNCTIONS WITH PARAMETERS

- Let's say we want to run the same few lines of code in multiple situations, and a few variables in those lines of code vary across situations. We might wrap those lines of code into a function that takes some input (i.e. parameters)
- Defining: func *name*(*parameterName*: *Type*) { /* code */} // One parameter
- Defining: func *name*(*parameterName*: *Type*, *parameterTwoName*: *Type*) { /* code */} // Two parameters
- Calling: *name*(*parameter*) // One parameter
- Calling: *name*(*parameter*, *parameterTwoName*: *parameterTwo*) // Two parameters

# SYNTAX - RETURN

- Let's say we're interested in the result of a certain bit of code, we might want that code to return a value which the calling code can capture and use
- Defining: func *name*() -> *ReturnType* { /* code */} // return ReturnType
- Defining: func *name*() -> (*ReturnType*, *ReturnType*) { /* code */} // returns a tuple or ReturnTypes
- Calling: var *value* = *name*() // Value is of type *ReturnType*

# SYNTAX - CALLING FUNCTIONS

- *name*() // No parameters, no return
- *name*(*parameter*) // One parameter, no return
- *name*(*parameter*, *parameterTwoName*: *parameterTwo*) // Two parameters, no return
- var *result* = *name*(*parameter*) // One parameter, one returned value
- let *result* = *name*() // No parameters, two returned values
  - println("\(result.paramOneName) \(result.paramTwoName)")

# SYNTAX – DEFINING FUNCTIONS

- func *name*() { /* code */ } // No parameters, no return
- func *name*(*parameterName*: *Type*) { /* code */} // One parameter, no return
- func *name*(*parameterName*: *Type*, *parameterTwoName*: *Type*) { /* code */} // Two parameters, no return
- func *name*(*parameterName*: *Type*) -> *ReturnType* { /* code */} // One parameter, one returned value
- func *name*() -> (*returnOne*: *valueOne*, *returnTwo*: *valueTwo*) {/* code */} // No parameters, two returned values

# XCODE DEMO: FUNCTIONS

# GETTING STARTED

EXERCISE

## KEY OBJECTIVE(S)

Create and use functions with parameters and return values.

## TIMING

| | |
|---|---|
| 30 min | 1. Code with partner |
| 5 min | 2. Debrief |

## DELIVERABLE

To the best of your ability, complete the provided playground file. If you hit a question you don't feel comfortable with, ask an instructor.

# FUNCTIONS RECAP

- Functions are blocks of code that are runnable from anywhere where the function is visible
- When a function is called from within our code, code execution steps into the function until it returns
- Functions can take parameters and return values
- When defining a function, **return** stops all execution of the function and kicks back out to the caller

# FUNCTIONS RECAP

- Be descriptive: Name your functions with descriptive names and descriptive parameters
- Be brief: Keep your functions short (i.e. approximately less than a screen's worth of content). You should be able to describe what they do in once sentence
- Compose: Your functions can call each other
- DRY: Don't repeat yourself. Any time you find the urge to copy and paste, there may be an opportunity to break into a function

# WHEN TO USE FUNCTIONS

- Functions are VERY common building blocks when writing code
  - But figuring out how to break them up is HARD, even for intermediate developers
- Any time you find the urge to copy and paste
- Any time you have multiple parts of your application sharing the same functionality, or very similar functionality with different parameters
- KISS: Avoid the urge to over-compose. Over-composed code can be just as difficult to read as under-composed code

# GETTING STARTED

EXERCISE

## KEY OBJECTIVE(S)

Create and use functions with parameters and return values.

## TIMING

| | |
|---|---|
| 30 min | 1. Code with partner |
| 5 min | 2. Debrief |

## DELIVERABLE

To the best of your ability, complete the provided playground file. If you hit a question you don't feel comfortable with, ask an instructor.