GENERAL ASSEMBLY

MOBILE DEVELOPMENT

FUNCTIONS

William Martin
Head of Product, Floored

LEARNING OBJECTIVES

- Describe the general uses of functions.
- Call and define functions that take parameters.
- Define and use functions that return values.

ACTIVITY

EXERCISE

KEY OBJECTIVE(S)

Describe and learn about functions from your peers.

TIMING

14 min	1. Discuss the following questions as a group.
1 min	2. Debrief and post to Slack.

DELIVERABLE

Post a single question to Slack that indicates the most important (or confusing) thing you'd like to learn about functions.

PROMPTING QUESTIONS

- How would you describe a function to someone in non-programming terms?
- How are functions and variables/constants similar? How are they different?
- What does it mean for a function to return a value?
- What's a parameter?
- What would a function look like that converts Fahrenheit to Celsius?

INTRO TO FUNCTIONS

WHAT IS A FUNCTION?

- A function is like a Rube Goldberg machine.
- It’s like a machine that from the outside performs a really simple task, but inside, is potentially really complicated.
- http://coolmaterial.com/roundup/rube-goldberg-machines/
- The "Honda Cog" (HD Version) : https://youtu.be/bl2U1p3fVRk

WHAT IS A FUNCTION?

- A function is a way of grouping code under a name.
- Writing a function is called “defining a function.”
- Executing that code is called “calling” a function.
- Like a variable, but can represent instructions in addition to fundamental values.
- Optional input and optional output.
- Term comes from mathematics:
  - *e.g.* $f(x) = ax^2 + bx + c$
  - $y = f(x)$, which means: $input\ x \rightarrow f \rightarrow output\ y$

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.

First, we must "define" the function.

DEFINING SIMPLE FUNCTIONS

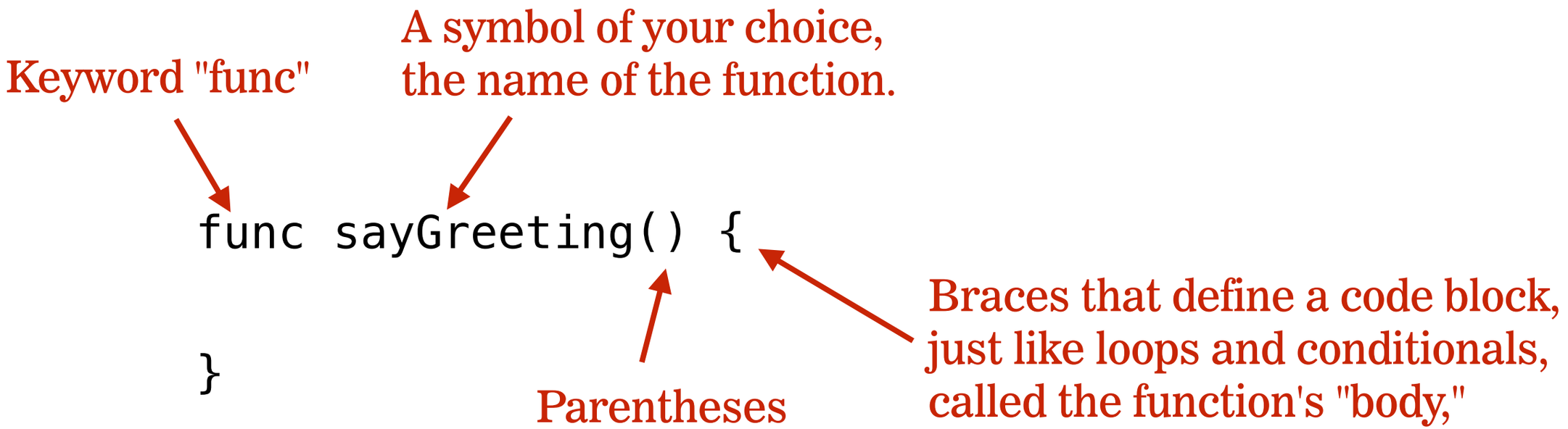
SYNTAX - DEFINING SIMPLE FUNCTIONS

"Defining a function" uses the following basic syntax. Everything we do will be variations on this basic form:

```
func sayGreeting() {
    /* Our code goes here in the function's "body." */
}
```

SYNTAX - DEFINING SIMPLE FUNCTIONS

Anatomy of a function definition:



SYNTAX - CALLING SIMPLE FUNCTIONS

“Calling a function” runs all the code within it, then continues in the spot of code where the function was called (i.e. the "call site").

Calling the function looks like this:

```
sayGreeting()
```

XCODE DEMO

FUNCTIONS WITH PARAMETERS

SYNTAX - FUNCTION WITH ONE PARAMETER

Functions can accept inputs. Here, "name" is a special variable called a “parameter,” which represents a single value as input to the function.

```
func sayGreeting(name:String) {
    println("Hello, \(name)!")
}
```

"name" is has a type like any other variable, denoted by a colon and a type name, just like declaring the type of a variable explicitly. It is only available within the braces that define the function’s “body.”

SYNTAX - FUNCTION WITH ONE PARAMETER

The variable is initialized during the function call. Place the value you'd like "name" to have between the parentheses when you call the function. We say we are “passing” the value “Toshi” to the function.

```
func sayGreeting(name:String) {
    println("Hello, \(name)!")
}
sayGreeting("Toshi")
```

SYNTAX - FUNCTION WITH ONE PARAMETER

You can also “pass” a value that comes from another variable, or any other expression that has the same type as “name”.

```
func sayGreeting(name:String) {
    println("Hello, \(name)!")
}
let myPupsName = "Toshi"
sayGreeting(myPupsName)
```

SYNTAX - FUNCTION WITH TWO PARAMETERS

You can give multiple parameters to a function by listing them together, separated by commas:

```
func say(greeting:String, name:String) {
    println("\(greeting), \(name)!")
}
say("Hello", "Toshi")
```

XCODE DEMO

FUNCTIONS WITH RETURN VALUES

SYNTAX - FUNCTIONS WITH RETURN VALUES

We say a function "returns a value" when calling it produces a value. That value can then be consumed at the call site.

```
let emailSubject = composeGreeting("Hello", "Toshi")
```

Here, the function call to the right of the = sign is producing a value. That value gets assigned to the constant `emailSubject`.

SYNTAX - FUNCTIONS WITH RETURN VALUES

This is the equivalent of what we want to happen when we call the function `composeGreeting`:

```
let emailSubject = "Hello, Toshi!"
```

When we call the function, it should produce the value of the message. How do we make that happen?

SYNTAX - FUNCTIONS WITH RETURN VALUES

To define a function that returns a value, we have to declare the *type* of the value to be returned. The syntax includes adding an arrow and type name after the parentheses:

```
func composeGreeting(phrase:String, name:String) -> String {


}
```

SYNTAX - FUNCTIONS WITH RETURN VALUES

Then, we *must* use the keyword "return" to provide a value (which must be of type String) that will be produced when the function is called.

```
func composeGreeting(phrase:String, name:String) -> String {
    return phrase + "! " + name
}
```

The value after "return" must be of type String.

SYNTAX - FUNCTIONS WITH RETURN VALUES

Now, when we call the function, it produces a value that we can use like any other value:

```
func composeGreeting(phrase:String, name:String) -> String {
    return "\(phrase), \(name)!"
}
let emailSubject = composeGreeting("Hi", "Toshi")
println(emailSubject)
```

SYNTAX - FUNCTIONS WITH RETURN VALUES

The type of the return value must match the type of the variable to which it is assigned. Here, we make it explicit just to demonstrate they must be identical.

```
func convertFeetToMeters(feet:Double) -> Double {
    let feetPerMeter = 3.28   // A Double
    return feet / feetPerMeter
}
var footballFieldLength : Double = feetToMeters(300.0)
println(footballFieldLength)
```

XCODE DEMO

FUNCTION DETAILS

FUNCTION SCOPE

Variables and constants defined within function bodies are only available inside the function body:

```
func convertFeetToMeters(feet:Double) -> Double {
    let feetPerMeter = 3.28
    let meters = feet / feetPerMeter
    return meters
}
// feetPerMeter and meters won't be available here!
```

FUNCTION COMPOSITION

Functions can call other functions:

```
func convertInToCm(inches:Double) -> Double {
    return inches * 2.54
}
func convertMiToKm(miles:Double) -> Double {
    let inches = miles * 5280 * 12
    let cm = convertInToCm(inches)
    return cm / 100_000
}
```

FUNCTIONS BEST PRACTICES

FUNCTIONS RECAP

- Functions are blocks of code that are runnable from anywhere the function is visible.
- When a function is called, code execution steps into the function until it returns.
- Functions can accept parameters and can return values.
- When defining a function, **return** indicates a place to stop execution of the function and go back to where the function was called (i.e. the “caller” or the “call site”).

FUNCTION TIPS

- Be descriptive: Name your functions with descriptive names and descriptive parameters. (e.g not *f* or *myFunction*, but *isColdOutside* or *convertKmToMi*)
- Be brief: Keep your functions short, and they should each do *one thing*.
- 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 write a function.

WHEN TO USE FUNCTIONS

- Functions are VERY common building blocks when writing code.
  - But figuring out how to break them up is often *hard*.
- Generally, you can use a function:
  - any time you find the urge to copy and paste
  - any time you have multiple parts of your application sharing the same functionality, potentially only differing by the *values* they use.
- Keep it simple! Aim for elegant readability.

FUNCTIONS

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.