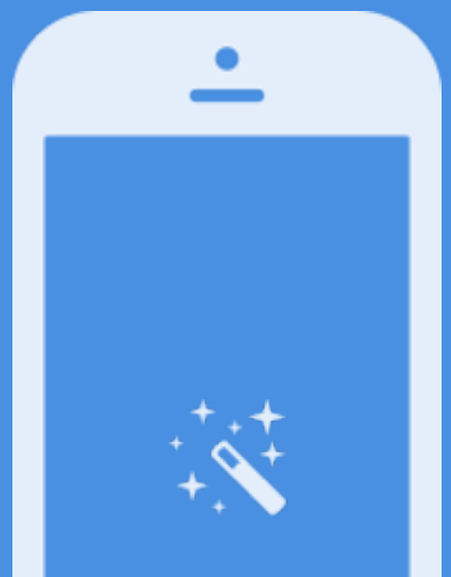
iOS Crash Course

Session Three

Janum Trivedi
iOSCrashCour.se



Session 2 Recap

- History and overview of Objective-C
- Basic structure of an Objective-C code file
- C compatibility (if, for/while, int/float, bool, etc.)
- NSLog and string interpolation (%i, %@, etc.)
- String literals with NSString
- Difference in declaring primitive variables vs. objects
- How we can call methods on objects to do or return something

Session 3 Overview

- Today, we'll be exploring real-world Objective-C classes
- Particularly, how we can use them to create interactive views in our iPhone apps
- From static to dynamic

Session 3 Outline

- Specifically, we'll be talking about:
  - Writing Objective-C functions
  - Creating custom classes
  - Adding properties to classes
  - Model-View-Controller (MVC) Architecture
  - IBOutlets/IBActions, interactivity

Questions

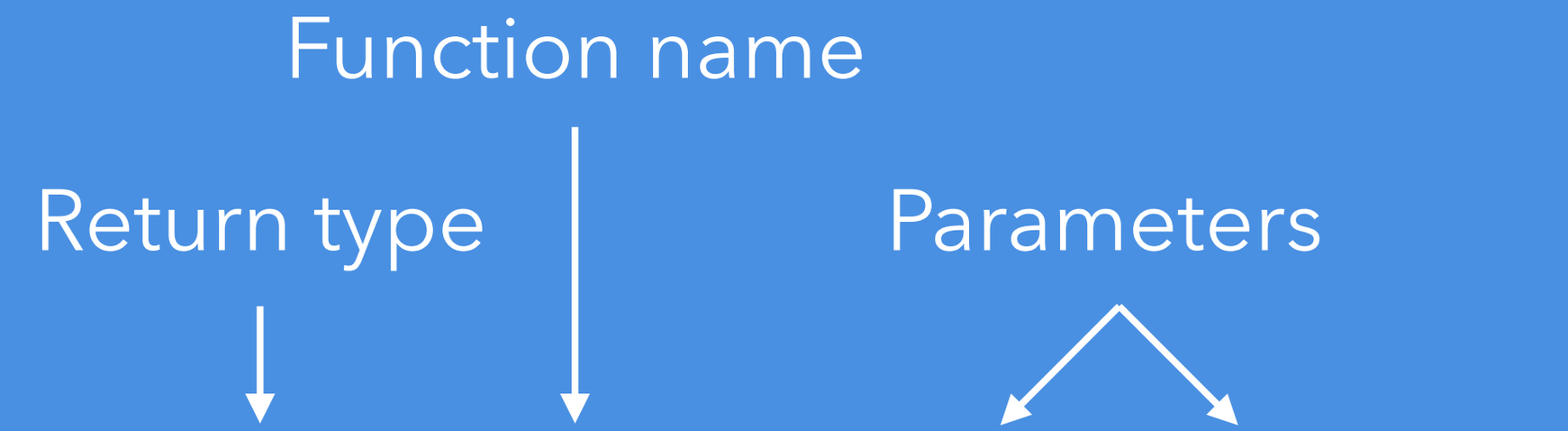
Objective-C Functions

- Last session we learned about how objects have methods (which are functions owned by an object):
  - `[NSString uppercasedString]`
  - `[NSMutableArray addObject:object]`
- But we didn't learn how to define our own Objective-C functions

Objective-C Functions

- Refresher: A function is a self-containing block of code that can be re-used
- Good codebases break up code into modular functions, such that it is easier to maintain, debug, and read

C++ Function Structure




```
int multiply(int x, int y) {
    return x * y;
}
```

Objective-C Function Structure

Function type (instance vs. class)

Name

Return type

Parameters

```
- (int)multiply:(int)x :(int)y {
    return x * y;
}
```

Generic Structure

```
- (returnType)functionName;
- (returnType)functionName:(argType)argName;
```

Calling Functions

- C++: `multiply(x, y)` or `multiply(4, 5)`
- In Objective-C, functions must be called on an object:
  - `[someObject functionName];`
  - `[self multiply:4 :5];`

Custom Classes

- Refresher: A **class** is a template for creating objects. We can create a class to model anything– a person, shape, photo, etc.

- A class contains instance (member) variables, or **properties** as they're called in Objective-C

- A person object will have properties for their first name, last name, birthday, etc.

- And we can create a single person object using our class

Custom Classes

- In Objective-C, classes are defined with two files: the **header** file and the **implementation** file

- The **header** file, or the **.h** file, is where we *declare* our properties/instance variables and functions that our class contains

- The **implementation** file, or the **.m** file, is where we actually *define* our functions (like a **.cpp**)

Custom Classes

- Anything we declare in our header file will be **public**
  - Available to other classes
- So if there's a private "helper" function we want to write that is only applicable to that class, just declare it in the **.m** implementation file

.h and .m

- Think of the .h (header) and .m (implementation) files like a textbook
- At the end of a book, there is an index– a list of all the terms that are defined in the book. The .h file is like that index. It tells us what is available, but doesn't actually give us the definition.
- The .m is where the actual contents of our dictionary (or function) are defined

Custom Classes

- Remember: a class will contain **properties** (aka member/instance variables in C++)
- Basic structure is as follows:

```
@property (nonatomic, strong) ClassName*
propertyName;
```

Questions

Model-View-Controller

- Central concept of iOS Development
- MVC is a design pattern that is used to structure our iOS apps
- At its most basic, MVC says is that we must keep 3 types of things separate: our **models**, our **views**, and our **controllers**

Models in MVC

- Our models represent objects, like the Person class we just created
- That “modeled” what a person is
- Class of it’s own– not defined inline in some other class or file

Models in MVC

- Most importantly: your model **only** knows about **itself** and its data

- It knows what a “Person” *is*, but doesn’t, and shouldn’t, not how it is *presented* to the user

  - Do we list people in a table? How big? What font? Which view

  - Your model doesn’t care about these questions

Views in MVC

- Views are what represent the user interface (like a Storyboard)
    - All the screens, buttons, labels, etc. that the user **sees** and **uses**
- Views know how they are meant to look/be presented, but don't know anything about the data
- Views aren't responsible for retrieving, storing, or processing data– **just how to present it**

Controllers in MVC

- What ties the model and view together then? The controller

- The controller is the glue that has access to both the Model and the View files, and acts as the intermediary between them

- The controller could ask the model for data, then ask the view to show it on the screen

Model-View-Controller

- Remember: all the Model-View-Controller design pattern means is that everything we create in iOS is either going to be a **model**, **view**, or **controller**

- Sometimes that line gets hazy (as we'll see shortly with a class called UIViewController), but it acts as a good guideline of how we should design our programs

MVC Relationships

- We also have to be extremely explicit in defining the relationships between our views and models

- Xcode doesn't know how we want that relationship

- When we tap a button, we need to explicitly write what should happen, like changing the text of a label

MVC Relationships

- Tapping that button would be an **action**, and for Xcode to change the text of a label, it needs to know that label exists as a **property** or **outlet**

- Let's open up Xcode and see what this looks like in practice