GENERAL ASSEMBLY

MOBILE DEVELOPMENT

CLASSES AND OBJECTS

LEARNING OBJECTIVES

- Define your own types, called *classes*.
- Instantiate those classes into *instances*.
- *Set* and *get* properties on classes.
- Describe how to define and call methods on an instance.
- Build *derived* classes from existing classes.

DEFINING CLASSES

WHAT IS A CLASS?

Swift, out of the box, provides us with some very basic data types. Ints, Doubles, Bools, etc.

These are very useful, but they're also very raw. A Double may be used to represent temperatures in one place or distance in another.

WHAT IS A CLASS?

Our apps will generally be more complex than apps with numbers and text.

For example, in a task-manager app, we'll need to represent tasks, projects, due dates, etc.

In a map app, we'll have to describe pins, annotations, locations, and so forth.

Is there a more natural way to do this than group together a bunch of Strings and Doubles?

WHAT IS A CLASS?

Yes, there is indeed a way, through *classes*.

A *class* is a mechanism that enables us to represent higher-level concepts, like tasks and projects, people and dogs, etc., using the basic building blocks we've already been using.

Strictly speaking, classes are *user-defined types*.

WHAT IS A CLASS?

- A class consists of
  - *state (values, held in "properties") and*
  - *behavior (functions, called "methods").*
- A class is a *template* for creating individual objects with specific values.

WHAT IS A CLASS?

- One “defines” a class using the “class” keyword.
- After, one creates “instances” of classes through a process called “instantiation.”
  - Dog could be a class.
  - Toshi, a specific dog, is an *instance* of the class Dog.
- Note that we’ll use the word “object” somewhat interchangeably with “instance.”

EXAMPLES OF CLASSES

EXAMPLES : DOG

A class called Dog.

- Instances are individual dogs, like Toshi, Neeko, and Layla.
- State (properties): They can have a name, breed, weight, etc.
- Behavior (methods): They can bark, run, sit, etc.

EXAMPLES : TASK

A class called Task.

- Instances are individual todo items.
- State (properties): They can have title, notes, due date, priority, owner, etc.
- Behavior (methods): They can be completed, rescheduled, deferred, delegated.

EXAMPLES : TASK

A class called Window.

- Instances are individual windows in a house.
- State (properties): They have a type, height, width, R-rating, etc.
- Behavior (methods): They can be opened, closed, installed, cleaned, etc.

DEFINING AND INSTANTIATING

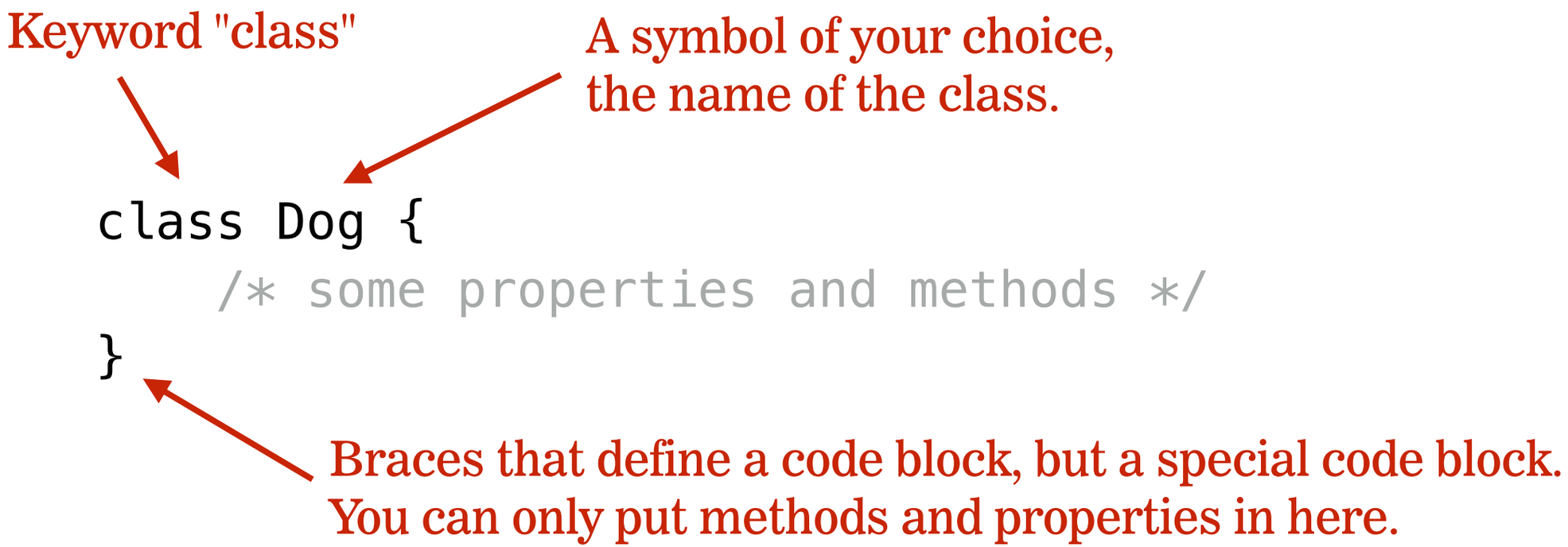
SYNTAX: DEFINING A CLASS

This is the base syntax for defining a class.

```
class Dog {
    /* some properties and methods */
}
```

SYNTAX: DEFINING A CLASS

This is the base syntax for defining a class.



SYNTAX: CREATING AN INSTANCE

This creates (i.e. “instantiates”) a single instance of the class “Dog” and assigns it to a variable.

```
var toshi = Dog()
```

The “type” of the value contained by “toshi” is Dog. The instance doesn’t do anything yet, because the class has not properties and no methods.

METHODS

SYNTAX: DEFINING A CLASS

To add methods to our class, we can define functions inside the braces, like this:

```
class Dog {
    func bark() {
        println(“Woof!”)
    }
}
```

SYNTAX: CALLING A METHOD

Using “dot notation,” we can reference the method (function) “bark” within the instance and then *call* it like the functions we’ve written before.

```
var toshi = Dog()
toshi.bark()
```

SYNTAX: DEFINING A CLASS WITH A METHOD

There is a special method called an “initializer” that is only called when the class is instantiated. It’s symbol is “init” and it typically looks like this:

```
class Dog {
    init() {
        println(“New Dog!”)
    }

    func bark() {
        println(“Woof!”)
    }
}
```

SYNTAX: CREATING AN INSTANCE

Now try instantiating an instance of a Dog.

```
var mine = Dog()
```

You should see that “New Dog!” appears in your playground’s console.

PROPERTIES

SYNTAX: DEFINING A CLASS WITH A PROPERTY

Let’s add a property to the class so its instances can hold some state.

```
class Dog {
    var name : String

    init() {
        println("New Dog!")
```

But we're not done!

SYNTAX: DEFINING A CLASS WITH A PROPERTY

Instances *must* have values for each property (or be Optional) upon instantiation.

```
class Dog {
    var name : String

    init() {
        println(“New Dog!”)
        self.name = “Toshi”
```

Thus we have to give the property "name" a value by initializing it in the class's initializer. Note the keyword "self", which enables an instance to *refer* to itself and its own properties and methods.

SYNTAX: ACCESSING A PROPERTY

Using “dot notation,” we can also reference a property that contains a value specific to the instance.

```
var toshi = Dog()
println(toshi.name)
```

SYNTAX: CLASSES WITH PARAMETERIZED INITIALIZER

Initializers can accept arguments. We can those arguments to initialize the properties of the instance:

```
class Dog {
    var name : String

    init(name:String) {
        self.name = name
        println(“New Dog!”)
    }
}
```

SYNTAX: CLASSES WITH PARAMETERIZED INITIALIZER

Now, we can specify the name of the dog at initialization time:

```
var myDog = Dog(name:"Toshi")
print(myDog.name)

// You can "set" the property like this.
myDog.name = "Toshi Martin"
print(myDog.name)
```

Accessing the value of a property is called "getting." Giving a property a new value is called "setting."

SYNTAX: MULTIPLE INSTANCES

Now, we can declare two independent instances of the class “Dog” and start to see the value of classes:

```
var myDog = Dog(name:"Toshi")
myDog.name

var mySistersDog = Dog(name:"Layla")
mySistersDog.name
```

These instances have different values for their “name” property.

SYNTAX: A CLASS WITH OPTIONAL PROPERTIES

Optional properties are declared like Optional variables, with a ?

```
class Dog {
    var name : String
    var age : Int?

    init(name:String) {
        self.name = name
    }
}
```

They don't have to be initialized, but they contain the value *nil* if they aren't.

SYNTAX: A CLASS WITH OPTIONAL PROPERTIES

The Optional property can be "set" after we instantiate the class:

```
class Dog {
    var name : String
    var age : Int?

    init(name:String) {
        self.name = name
    }
}
var myDog = Dog(name:”Toshi”)
myDog.age = 2
```

INTRO TO OBJECT-ORIENTED PROGRAMMING

FUNCTIONS THAT ACCEPT CLASSES AS TYPES

Here we can write a function that takes a Dog as the type of it's argument.

```
var myDog = Dog(name:"Toshi")

func prettyPrintDog(oneDog:Dog) {
    println("\(oneDog.name) is a Dog!")
}

prettyPrintDog(myDog)
```

Note that we can use .name to access that property of oneDog.

EXTENDING A CLASS : INHERITANCE

We can make classes from other classes and give them new behaviors and state. This enables us to extend the functionality of our classes and empower us to customize UI elements.

```
class Shiba : Dog {
    var temperament : String
}
```

We say that Shiba “inherits” from Dog. It has all the properties and functions from Dog, plus an additional property, temperament.

We call Shiba a "derived" class.

EXTENDING A CLASS : INITIALIZATION

When we extend a class, sometimes the initialization process needs to be different, especially if we have new properties that need to be initialized:

```
class Shiba : Dog {
    var temperament : String

    override init(name:String) {
        self.temperament = "stubborn"
        super.init(name:name)
    }
}
```

Note the use of "override," which we need if methods have the same name and arguments in the new, "derived" class.

EXTENDING A CLASS + INITIALIZERS

The "super" keyword refers to the "superclass" Dog. So super.init(name:...) is the initializer defined inside the class definition of Dog.

```
override init(name:String) {
    self.temperament = "stubborn"
    super.init(name:name)
}
```

Thus we don't have to repeat the work done inside Dog.init(), namely setting the value of ".name". This way, we can avoid repeating ourselves.

EXTENDING A CLASS + OVERRIDES

You can also redefine the implementation of the methods in derived classes.

```
override bark() {
    print("Shiba's don't bark...often.")
}
```

By using "override", we can give methods new bodies, and thus make the behavior of derived classes different from their superclasses.

POLYMORPHISM

Now, we can use Shiba wherever we can use Dog:

```
var myShiba = Shiba(name:”Toshi”)
prettyPrintDog(myShiba)
```

In this case, we can pass a Shiba into the prettyPrintDog function. All code that has been written with Dog in mind can also take Shiba, since Shiba has all the functions and properties of Dog, and Dog is a superclass of Shiba.

This phenomenon is called “polymorphism.”

ACTIVITY

KEY OBJECTIVE(S)

Work more with classes.

TIMING

25 min	1. Perform the exercises in the provided playground.
5 min	2. Debrief.

DELIVERABLE

A playground with the completed tasks, demonstrating defining classes, creating instances, and writing functions and variables that accept classes as their types.