iOS F2
Day 1

- Course Intro
- Coding Basics:
  - Variables
  - Functions
  - Algorithms
- OOP:
  - Classes
  - Method & Scope
  - Properties & Variables
  - Inheritance
- Github Intro

Spoiler: You will need Xcode for this class

Download Xcode from the app store on your mac, but don't do it now because our internet isn't great

Course Format

- 2 hours of lectures each class, split by a 10 minute water/restroom break in between.
- Each class will begin with review of the last class, except for today since its the first class.
- Homework is graded twice, once in the middle of the course and then at the very end. But homework is assigned after each class.
- All slides and sample code will be posted to the class Github repository after class.
- Homework is posted on Canvas. We also have a Slack chatroom for the class. Email me (brad@codefellows.com) if you haven't received an invite to these.
- **On Thursday from 5:30-7:00pm, there is office hour here on the first floor. This is your time to come in and get hands-on help from myself and the TAs.**

What are we going to learn?

OOP

Abstraction
Classes Objects
Inheritance

Swift

Syntax Optionals
Types Conditionals
Optional Bindings
Control Flow

UIKit

UIView UILabel
UITextField
UIImageView
UITableView
UIViewController
UITableViewCell
UINavigationController

Coding Fundamentals

Variables Functions
Algorithms

Programming Design Patterns

Delegation MVC
Target-Action

Xcode

Navigation
Storyboards
Segues
Auto Layout

Source Control

git github

Dev Accelerator

- Moving directly from the F2 course to Dev Accelerator, without a break in-between, isn't for everyone.
- Usually you will take a few months in between to do some guided self study.
- If you are interested in taking the dev accelerator, you should basically be spending all or most of your free time working on apps.
- At the end of this F2 class, you can apply for the upcoming DA and we will sit down and figure out if you are ready.

Tips for doing well


Ask all the questions
In class and
on Slack too!

Solid Note Taking

- All slides and code are posted immediately after lectures are over.
- So instead of blindly typing everything we type or everything on the slides, try to just absorb the information into your brain.
- Scientific Studies have shown students typically have the best results if they take notes on a notepad during class and keep their laptops closed.

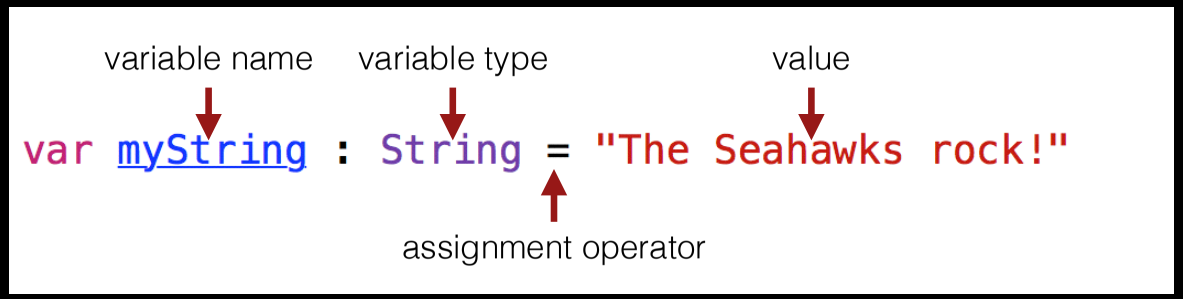
Don’t get stuck

- Remember to always consult with the TA’s, your instructor, and fellow students if you are stuck on something or if something is unclear. Hit us up on Slack!
- You can always look at the sample code from the lectures as well.
- Knowing how to Google your coding problems is a very important skill as a developer, so always try Googling as well. Chances are someone has asked that exact same question on stack overflow or another forum.

Coding Fundamentals

Variables

- Variables act as ‘storage locations’ for data in your apps.
- They are basically a way for naming information for later use
- Heres how we do it in Swift:



Common Types of Variables

- Every variable in Swift has a type. This describes to Xcode (and you, the developer), what this variable is capable of.
- Here are some of the most common types you will encounter:
  - **String** : An ordered collection of characters. Used most commonly to store words or sentences.
  - **Int** : An integer is any whole number
  - **Float** : Numbers with a fractional component (3.14156 or 0.1)
  - **Bool** : A logical type, can only be true or false
  - **UIImage**: Represents an Image
  - **Array** : An ordered collection of things
  - **Dictionary**: a collection of key-value pairings of things

Inferring the Type

- You actually don't need to assign the type explicitly, as Xcode can infer the type of the variable based on the value you give it:

```
var anotherString = "Booo 49ers suck"
```

is the same as

```
var anotherString : String = "Boooo 49ers suck"
```

Type inference is only available with Swift, not Objective-C.

Constants vs Variables

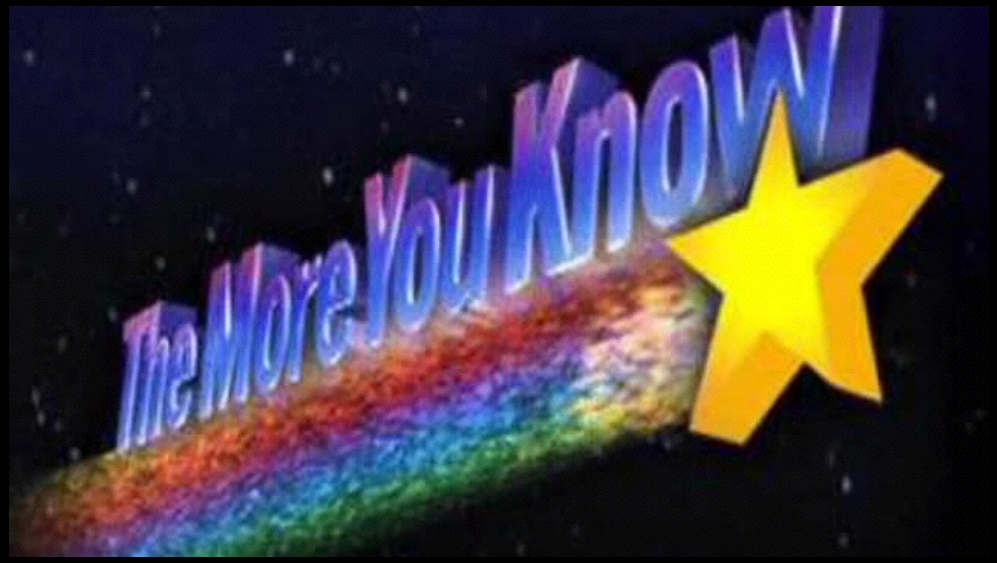
- Use the let keyword to make a constant
- Constants are just like regular variables, except they can only be assigned a value exactly once.

```
let myName = "Brad"
myName = "Bradley"

var city = "Seattle"
city = "Portland"
```

Invalid

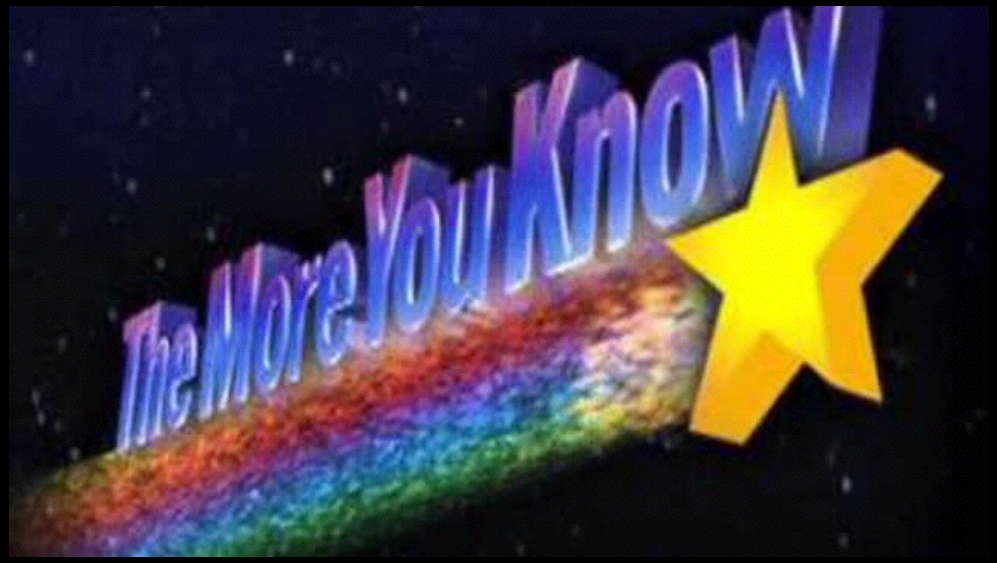
Valid

iOS Best Practice #1



- Make everything a constant, unless it absolutely needs to be a variable (aka you need to change its value, more rare than you might think)

iOS Best Practice #2



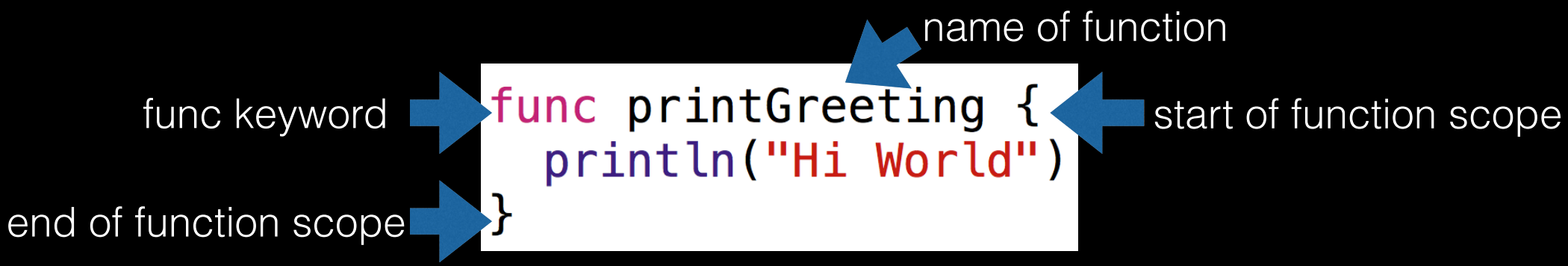
- Use camel case for all variable, constant, function, and method names
- ex: mySuperAwesomeVariable

Functions

- Think of a function as a procedure or a routine
- They ‘encapsulate’ a task, so you can perform that task with one line of code (by typing the name of the function) rather than having to type out the entire task over multiple lines of code

Defining Functions in Swift

- To declare a function in Swift, you use the keyword func, and then write the name of the function (which is completely up to you)
- You then use {}'s to define the scope of the function
- Everything inside of these two brackets is considered the body of the function. It's the code that will execute when the function is called



Calling Functions in Swift

- When you declare a function, you are essentially declaring its blueprint.
- Defining a function is one thing, but you need to actually call the function for the code inside of the function to run
- To call a function in swift, simply type out the name of the function, followed by ()
- Xcode should help you autocomplete it as you type the name

Functions can have input and output

- Often times, you will want to pass something into a function
- In these cases, you need to add parameter(s) to the function
- In addition, you can make the function spit something back out when it is done, by adding a return value to it
- In this sense, functions can have both input and output.

Defining your input AKA parameters

- Parameters are listed inside of parentheses at the end of the function's name
- The format of a parameter is: parameterName : parameterType
- There is no limit to the number of parameters a function can have
- To have more than one, use a comma separated list:

```
func doSomethingWithString(string : String, number : Int) {
  println(string)
  println(number)
}
```

Defining your output AKA return type

- When you want to have your function spit something out, you need to specify that it has a return value
- You do this by adding a -> at the end of the function's name (or parameter list if it has parameters)
- After the ->, just put the type of the return value (no need for a name)
- Finally, somewhere in your function, you must actually return something. Do this by using the return keyboard, and then some value

```
func doSomethingWithString(string : String, number : Int) -> String {
  println(string)
  println(number)
  return "Blah"
}
```

Algorithmic Thinking

- When people discuss programming and writing applications and such, the word algorithm comes up a lot
- Algorithm is just a fancy word for a bunch of functions (or a single function!) used to solve a problem
- So how do you get started writing your own algorithms?

Writing an Algorithm

- First, write an English description of what you want to happen
- Then, transform it into code:
  - Use nouns for variables
  - Use verbs for functions
- Lets look at an example

Object Oriented Programming

Object Oriented Programming

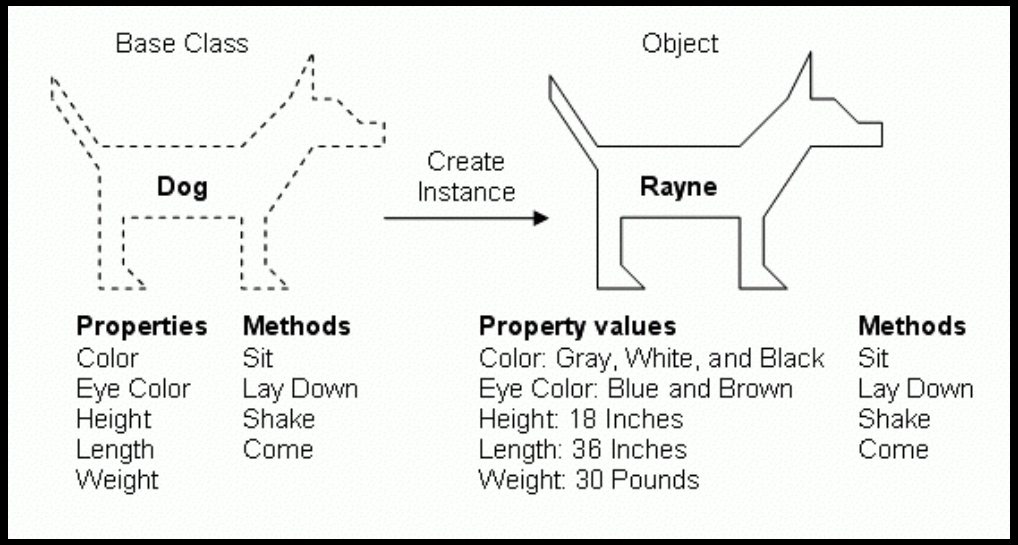
- In life, we are constantly bombarded with a ridiculous amount of information and impressions.
- We need to make sense of these things, or at least try to. One of our primary methods of making sense of the world is through abstractions.
- For example, if someone forgets your name on multiple occasions, we abstract those experiences into the idea that this person is rude.
- Its just easier for our mind to recall this idea, vs recalling all the individual experiences.

Object Oriented Programming

- Now think of how complex computers and their programs are. Millions of 1's and 0's being moved around and operated on.
- Object Orientation provides an abstraction of that data, and it provides a concrete grouping between the data and operations you can perform with that data.
- This is abstraction is provided by the concept of Classes & Objects.

Classes & Objects

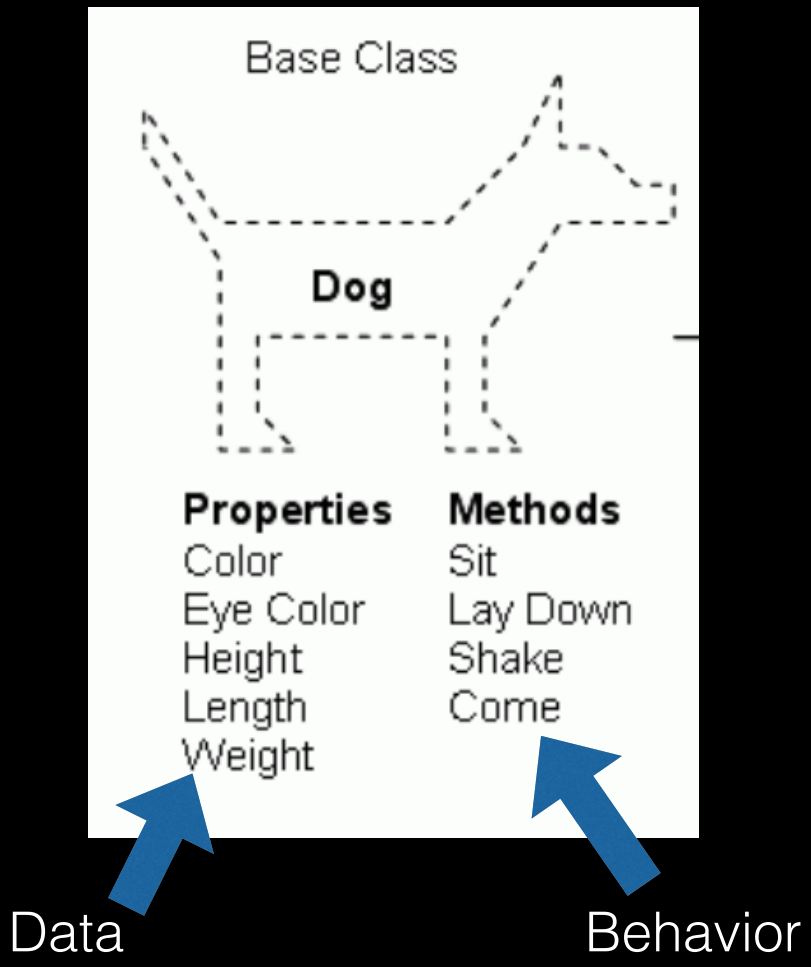
- A class is like a template
- And an object is a single instance of a class



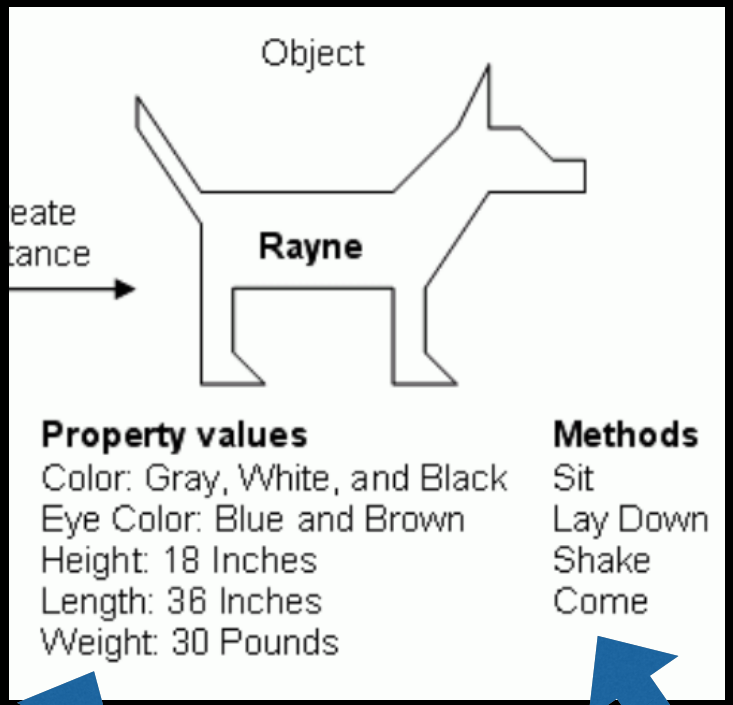
We have our class called ‘Dog’ on the left. That is our template. On our right we have an instance of the Dog class, her name is Rayne.

Classes: Data and Behavior

- Looking at just the class for a sec, we can see two different categories of ‘things’
- The properties of the class, and the methods of the class.
- Aka, the data of the class, and the behavior of the class.



Objects: Data and Behavior

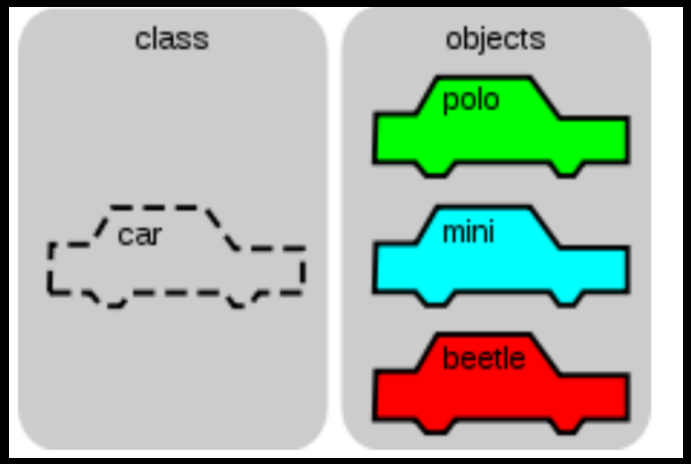


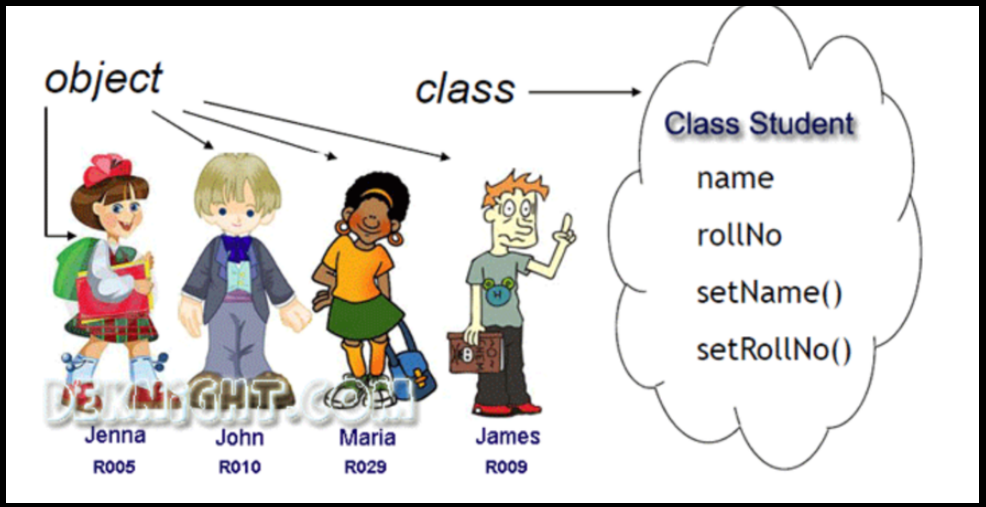
Data

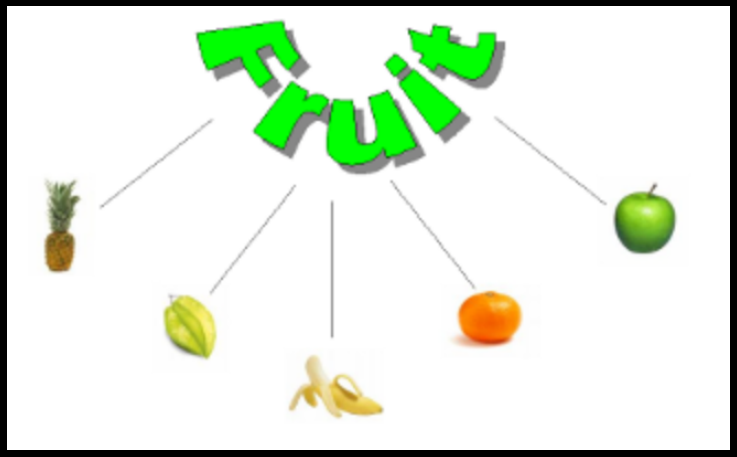
Behavior

- Now looking at an object.
- Which is just a single instance of the class.
- This object now contains real data, like an Eye Color of blue and brown.
- So the class is really just a template for a dog, an object is an actual dog with set attributes.

More borderline-silly Images to help you understand

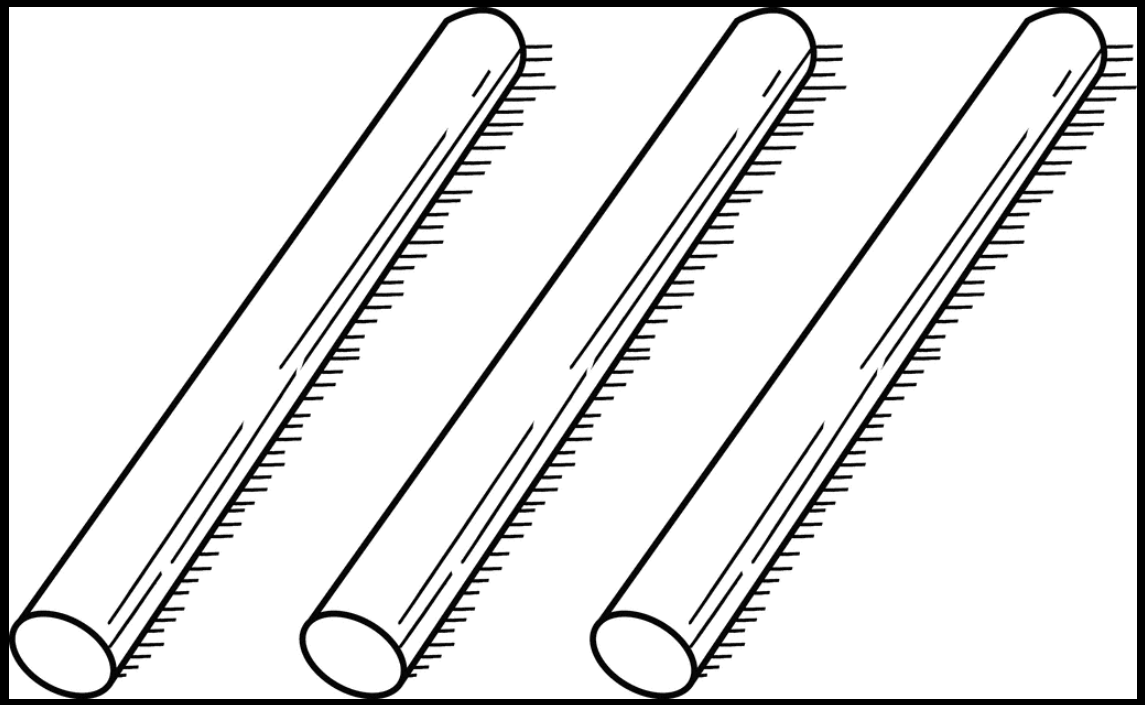






Object Oriented Programming

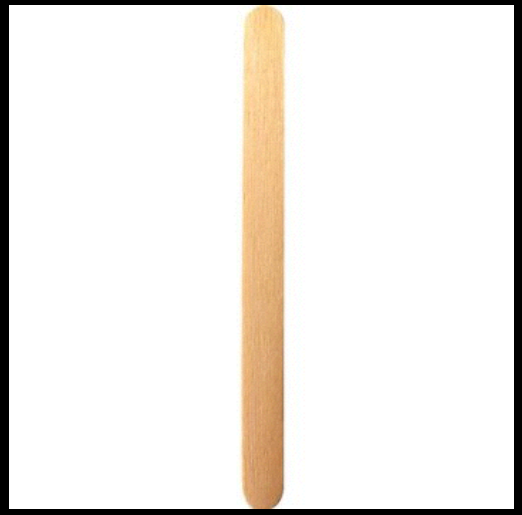
- So lets create some example classes and objects, and we will begin to see how OOP can make programming fun and easy. Well, kind of easy
- To do this, it will help if we all use the same examples to base our classes and objects off of.
- And there is no better example than…..



Sticks

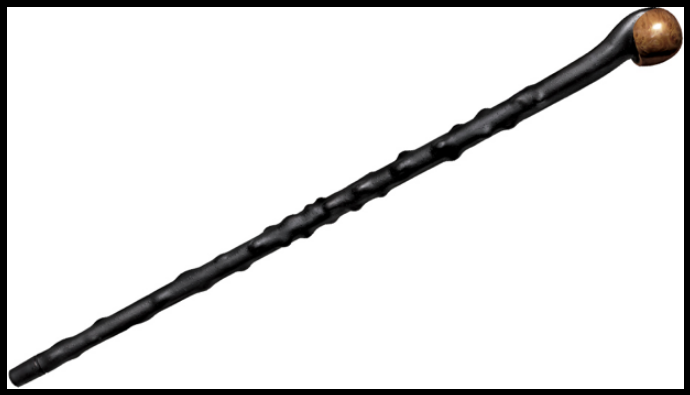
yes, sticks.

Cinnamon sticks



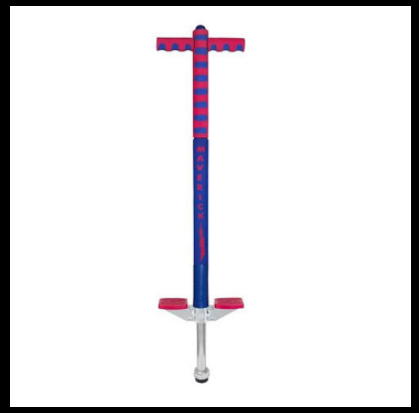
Popsicle sticks

Drum sticks



Walking sticks

Hockey sticks



Pogo sticks

Selfie sticks!!!!

Class Declaration & Scope

- You denote the beginning of a class with the keyword class, and then the name of your class
- What a class contains, all of its methods and properties, is considered the scope of the class
- In Swift (and in Objective-C!) the scope begins and ends with curly brackets {}.

```
class Stick {
  //an empty class
}
```

Here we have an empty class called Stick

Class Scope

```
class Stick {

  let length = 10
  let width = 1
  let color = "Blue"
  var faceUp = true

  func roll() {
    faceUp = !faceUp
  }

}
```

Scope begins

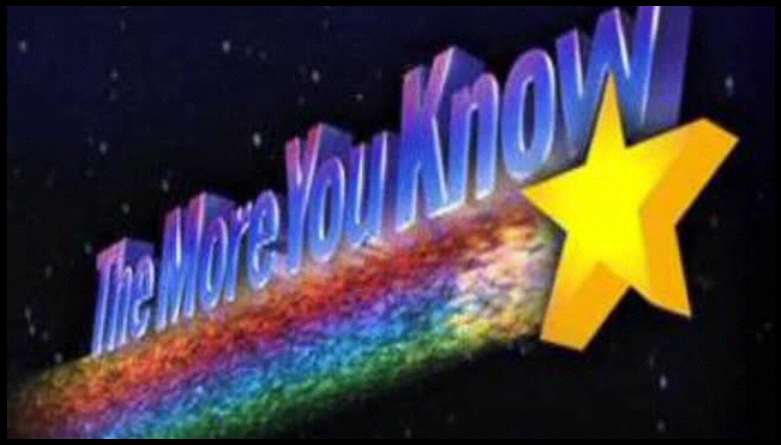
Scope ends

Demo

Properties

- Properties store a class's data.
- A Properties declaration looks a lot like a variables/constants declaration. Properties are variables, **but properties are globally available in all of the Class's methods.**
- Properties stay alive in memory as long as the object stays alive in memory. They differ from local variables because properties don't die at the end of a method (more on this in a bit)
- There are two ways of accessing properties, which you will see in the next few slides.
- Properties must be given default values, or marked as optionals (more on optionals a bit later in the course)

iOS Best Practice #3



- Properties are always declared at the top of a class.

```
class Stick {

  let length = 10
  let width = 1
```

Properties

Properties

- You can explicitly declare a property's type by adding a colon after the name, and then the name of the type.

```
class Stick {

  let length : Int = 10
  let width = 1
```

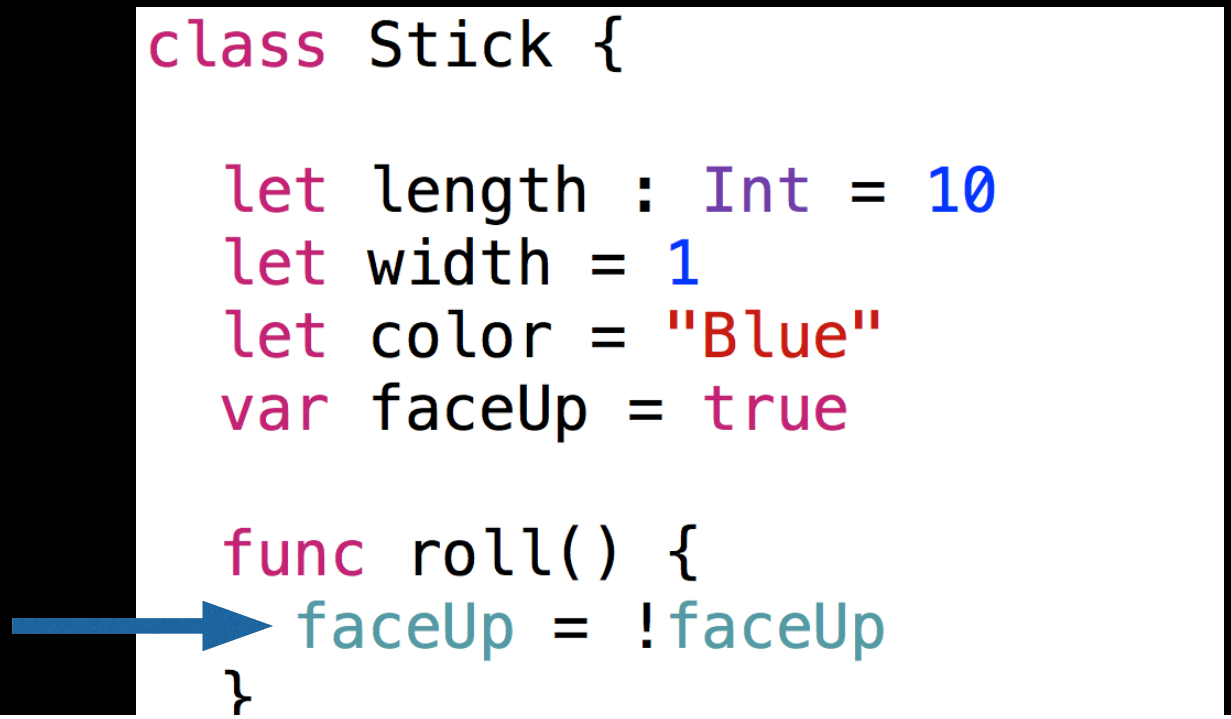
Type Inference

- But because we are giving these properties default values, Swift & Xcode can infer what type the property is! Thats why we didn't give level a type and it still works. Later on we will see how not giving a property a default value changes how we setup a class.

Demo

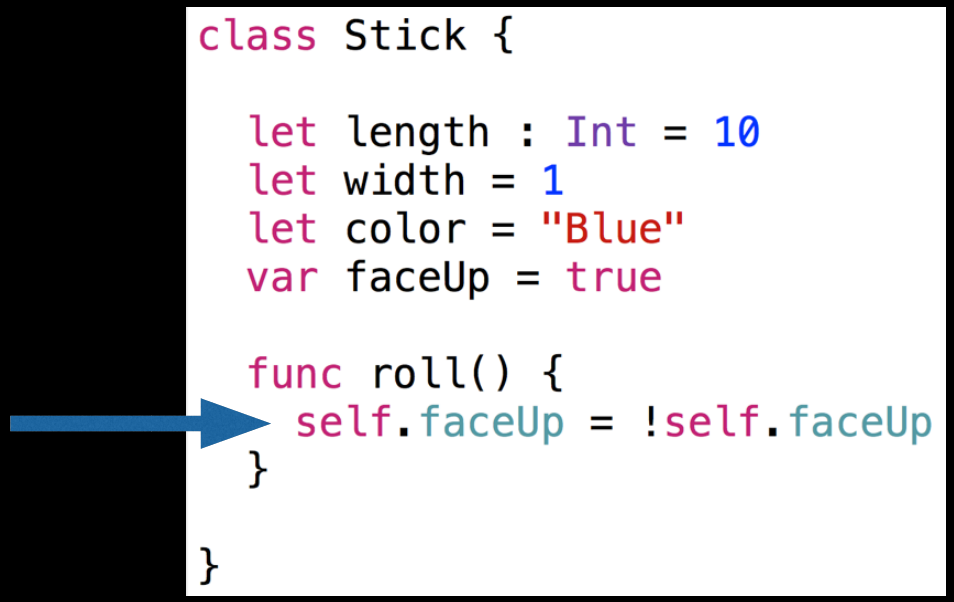
Accessing properties

- There are two syntactically different ways of accessing properties, both do the exact same thing.
- The first way is to just use the name of the property:



Accessing properties

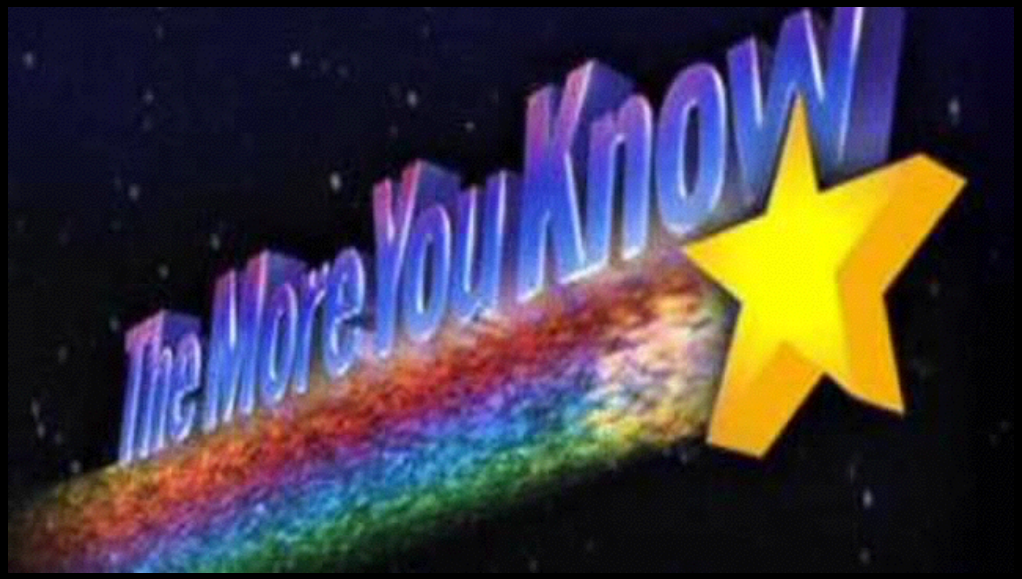
- The second way is to use to the implicit variable self, and then a dot, and then the name of the property.



```
class Stick {

  let length : Int = 10
  let width = 1
  let color = "Blue"
  var faceUp = true

  func roll() {
    self.faceUp = !self.faceUp
  }

}
```

self refers to the instance of the object this method is being called on.
This is a very common pattern of OOP languages, sometimes they call it **this**, instead of **self**

iOS Best Practice #4



- Never use self when you are accessing properties, unless required

(like in closures —an advanced topic)

Properties vs local variables

- There are **two primary differences** between properties and local variables:
  - Properties can be accessed by all the Class's methods. Local variables are limited to the method they are created in.
  - Properties are kept alive in memory, as long as the object that owns it is also kept alive. Local variables are only kept alive until the method they were created in is done executing.

Properties vs local variables

```
class Stick {

  let length : Int = 10
  let width = 1
  let color = "Blue"
  var faceUp = true

  func roll() {
    faceUp = !faceUp
  }

}
```

A class with no local variables

```
class Stick {

  let length : Int = 10
  let width = 1
  let color = "Blue"
  var faceUp = true

  func roll() {
    let rick = "Rick"
    println(rick)
    faceUp = !faceUp
  }

}
```

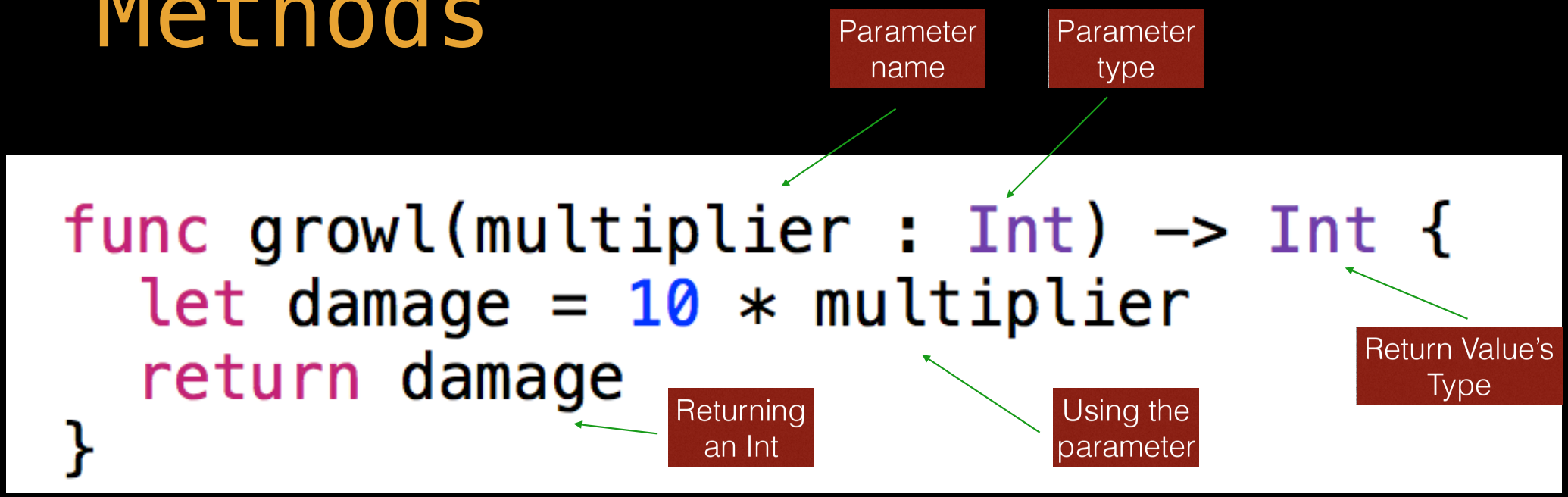
A class with one local variabe, rick

Demo

Methods

- A function that is defined inside of a class is called a method
- Think of a method as an action that instances of your class can perform
- Methods can take in info, referred to as **parameters**, that can then be referenced inside the methods
- Methods can also send info back when they are finished, referred to as **return value**(s)
- So methods can have both an input(parameters) and/or output(return values)

Methods


Parameter name
Parameter type
func growl(multiplier : Int) -> Int {
let damage = 10 * multiplier
return damage
}
Return Value's Type
Returning an Int
Using the parameter

Method Scope

- Similar to a Class, a method's scope begins and ends with the curly braces in Swift and Objective-C.
- Any variables created inside a method are destroyed once that method is finished. These types of variables are called **local variables**.
- Conceptually, think of the method as the owner of the variable. If no other object becomes the owner of the variable, it is then destroyed.

Method Scope

```
func sayHello() {

    var greeting = "Hello"
    println(greeting)
}
```

Scope begins

Scope ends, greeting dies

- our greeting variable is a local variable, since it is created inside of a method
- notice how there is no return type on sayHello. Return types are optional. So are parameters.

Demo

Calling a method

- If you recall from our dog example before, methods are the 'behavior' of a class.
- Calling a method is nearly exactly as the same as calling a function!

```
let myStick = Stick()
myStick.roll()
```

Calling the roll method on a stick

Demo

Storing a return value in another variable

- Often times you will see a function being called and a variable being created on the same line:

```
var myName = self.generateName()
```

- So here you are taking the output of the function generateName and storing it in a variable for use later on.

Demo

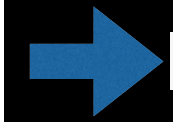
Init

- The way to create an object (an instance of a class) is to call the class's initializer. This is also called a constructor, which is a general term that many programming languages use to describe the way an object is created.
- An initializer is a special form of a method.
- To define an initializer in a class, you use the keyword init.
- It works just like a regular method, but it doesn't need the func keyword.

Init

- Here we have our Stick class, with an init that takes one parameter called startingLength:

```
class Stick {

  var length : Int = 10
  let width = 1
  let color = "Blue"
  var faceUp = true

  init (startingLength : Int) {
    length = startingLength
  }
  func roll() {
    faceUp = !faceUp
  }

}
```



```
let myStick = Stick(startingLength: 20)
```

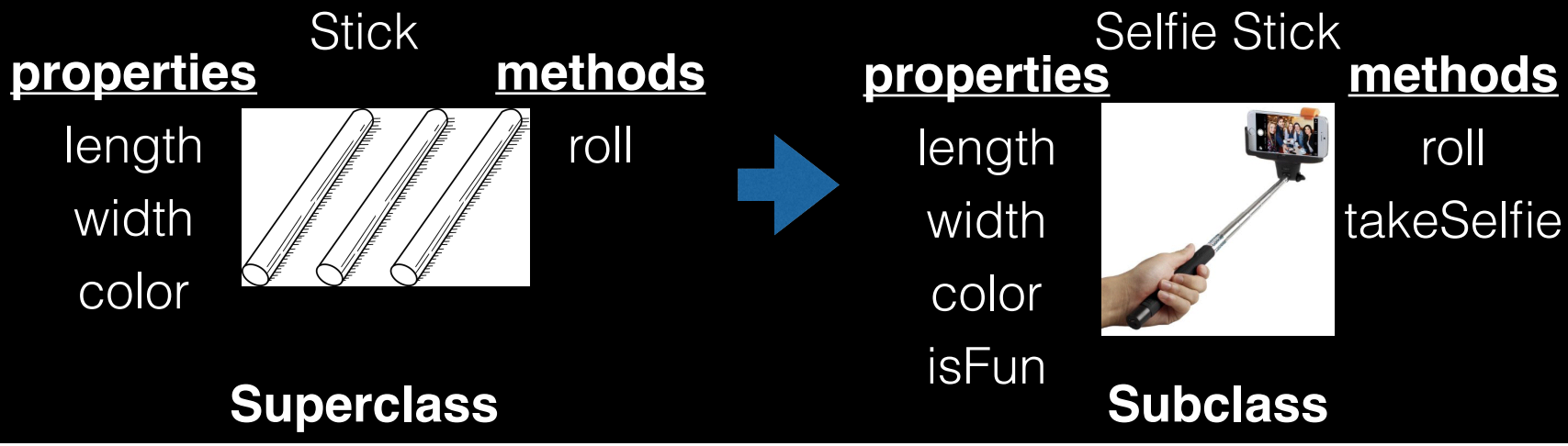
Default Inits

- If you don't create an init yourself, Swift will provide your class with one.
- In order for this to work, your properties must all have values set in their declarations.

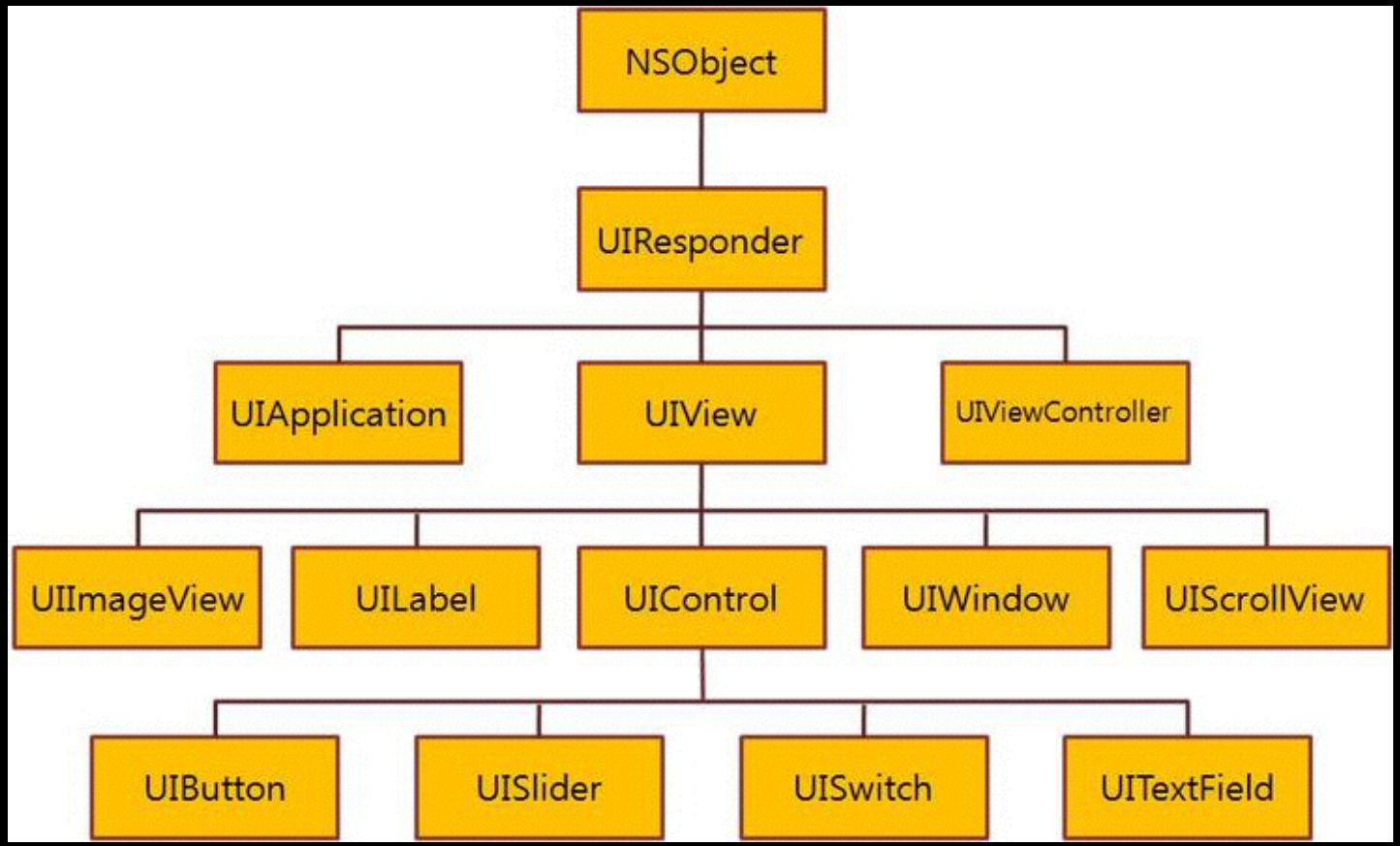
Demo

Inheritance

- Classes can Inherit from other classes. When a Class inherits from another class, it inherits all of its methods and properties.
- The inheriting class is known as the subclass, and the class it inherits from is known as the super class.
- There is no limit to how long your subclassing trail can go. So we could have a GroupieStick class that subclasses SelfieStick, which subclasses Stick.



An Example of Apple's Inheritance



Inheritance

```
class Stick {

  var length : Int = 10
  let width = 1
  let color = "Blue"
  var faceUp = true

  init (startingLength : Int) {
    length = startingLength
  }
  func roll() {
    faceUp = !faceUp
  }

}
```

```
class SelfieStick : Stick {
  func takeSelfie() {
    println("say cheese!")
  }
}
```

```
let mySelfieStick = SelfieStick(startingLength: 20)
mySelfieStick.roll()
```

- Heres where we start to see the power of inheritance.
- We don't have to redefine all the methods we want SelfieStick to inherit from Stick
- They are automatically inherited.

Demo

Github Intro