# Objects and Class

* Classes are always passed by reference

## Declaring Class

* **class Shape {**
* **var numberOfSides = 0**
* **func simpleDescription() -> String {**
* **return "A shape with \(numberOfSides)**
* **}**
* **}**

## Creating instance of class

* **var shape = Shape()**
* **shape.numberOfSides = 7**
* **var shapeDescription = shape.simpleDescription()**

## Property

* A property in a class is written the same way as a constant or variable declaration.
* Properties can also have a getter and setter
* **struct Animal {**
* **var name: String = ""**
* **var heightInches = 0.0**
* **var heightCms = Double {**
* **get {**
* **return 2.54 \* heightInches**
* **}**
* **set (newHeightCms) {**
* **heightInches = newHeightCms/2.54**
* **}**
* **}**
* **}**
* **var dog = Animal (name: "dog", heighInches: 50)**
* **dog.heightInches // This will output 50**
* **dog.heightCms // This will output 127. We never declare this but this is automatically calculated**
* **dog.heightCM = 254**
* **dog.heightInches // This will output 100. This was automatically set by swift**
* **dog.heightCms // This will output 254**

## Initializers

* Swift would not allow to create a class without an initializer
* **class CarWithCups {**
* **var cupHolder: String**
* **}**
* The above class will not be allowed
* You can do two things:
  + **class CarWithCups {**
  + **var cupHolder: String = "two holder"**
  + **}**
  + When you do the above, swift creates a default initializer for you
* The other think you can do is
  + **class CarWithCups {**
  + **var cupHolder: String**
  + **init(cupHolder: String) {**
  + **self.cupHolder = cupHolder**
  + **}**
  + **}**
* There are two types of initializers. First is required initializers
  + **class CarWithCups {**
  + **var cupHolder: String**
  + **required init(cupHolder: String) {**
  + **self.cupHolder = cupHolder**
  + **}**
  + **}**
* Second is convenience initializers
  + class CarWithCups {
  + var cupHolder: String
  + convenience init() {
  + self.init(cupHolder: "Cool:)
  + }
  + }
* Convenience initializer always has to call required initializer
  + **let car = Car(cupHolder: "Cool")**
  + **let car = Car()**
* If the class is a subclass then the initializer always has to call the superclass initializer
  + **class CarWithCups: NSString {**
  + **var cupHolder: String**
  + **required init(cupHolder: String) {**
  + **self.cupHolder = cupHolder**
  + **super.init()**
  + **}**
  + **}**
* Every convenience has to call required. Every subclass has to call superclass initializer.

## Deinitializers

* Called when object is going to be deallocated
* There is only one deinitializer
* **class CarWithCups: NSString {**
* **var cupHolder: String**
* **required init(cupHolder: String) {**
* **self.cupHolder = cupHolder**
* **super.init()**
* **}**
* **deint {**
* **}**
* **}**

## Inheritance

* **class SuperNumber: NSNumber {**
* **override func getValue(valueL UnsafeMutablePointer<Void>) {**
* **super.getValue(value)**
* **}**
* **}**
* SuperNumber inherited from NSNumber.
* It overrides a function of the super class and called the super function again.
* Basically, the overridden function did nothing new.

## Extension

* What if rather than overriding we want to add extra functionality
* **extension NSNumber {**
* **func superCoolGetter() -> Int {**
* **return 5**
* **}**
* **}**
* This is really powerful because it can work for system classes as shown
* **let n = NSNumber(int: 4)**
* **n.superCoolGetter() // This will give 5**
* This will also work for our own classes:
* **extension SuperNumber {**
* **func superCoolGetter() -> Int {**
* **return 5**
* **}**
* **}**
* **let n = NSNumber(int: 4)**
* **n.superCoolGetter() // This will give 5**

## Protocol

* Protocol is an interface to a class
* Let’s say we want to create a series of object that can dance
  + **protocol danceable {**
  + **func dance()**
  + **}**
* You don’t have to implement it but only define the functionality of the class
* Now a class can conform to a protocol
  + **class Person: danceable {**
  + **func dance() {**
  + **// something**
  + **}**
  + **}**
* A struct can also conform to a protocol
  + **struct Person: danceable {**
  + **func dance() {**
  + **// something**
  + **}**
  + **}**
* An enum can also conform to a protocol
  + **enum Person: danceable {**
  + **func dance() {**
  + **// something**
  + **}**
  + **}**
* Syntax is similar to subclass. A class can be a subclass and also conform to multiple protocols.
  + **class Person: NSNumber, danceable, NSObjectProtocol {**
  + **func dance() {**
  + **}**
  + **}**