Type Annotation Cont.

- Type Annotations in Swift are a really useful way to ***make sure*** that the specified value matched that of the expected value

Data Types

Integers

- Integers are whole numbers with no fractional component signed or unsigned.
- Swift provides ***signed*** and ***unsigned*** integers in 8, 16, 32, and 64 bit forms.

UInt

- The unsigned integer type.
- Has the same size as the current platform's native word size:
  - ✓ On a 32-bit platform, UInt is the same size as UInt32.
  - ✓ On a 64-bit platform, UInt is the same size as UInt64.

Int

- If you don't need to pick a specific size of integer to use in your code, always use ***Int*** for integer values in your code which has the same size as the current platform's native word size:
  - ✓ On a 32-bit platform, Int is the same size as Int32.
  - ✓ On a 64-bit platform, Int is the same size as Int64.

Integers-review

- There are following important points related to Integer types:
  - ✓ On a 32-bit platform, Int is the same size as Int32.
  - ✓ On a 64-bit platform, Int is the same size as Int64.
  - ✓ On a 32-bit platform, UInt is the same size as UInt32.
  - ✓ On a 64-bit platform, UInt is the same size as UInt64.
  - ✓ Int8, Int16, Int32, Int64 can be used to represent 8 Bit, 16 Bit, 32 Bit and 64 Bit forms of signed integer.
  - ✓ UInt8, UInt16, UInt32 and UInt64 can be used to represent 8 Bit, 16 Bit, 32 Bit and 64 Bit forms of unsigned integer.

Integers-ranges

Type	Typical Bit Width	Typical Range
Int8	1byte	-127 to 127
UInt8	1byte	0 to 255
Int32	4bytes	-2147483648 to 2147483647
UInt32	4bytes	0 to 4294967295
Int64	8bytes	-92233720368547758O8 to 92233720368547758O7
UInt64	8bytes	0 to 18446744073709551615
Float	4bytes	1.2E-38 to 3.4E+38 (~6 digits)
Double	8bytes	2.3E-308 to 1.7E+308 (~15 digits)

Numeric Literals

- Integer literals can be written as:
  - ✓ A *decimal* number, with no prefix
  - ✓ A *binary* number, with a 0b prefix
  - ✓ An *octal* number, with a 0o prefix
  - ✓ A *hexadecimal* number, with a 0x prefix

```
let decimalInteger = 17

let binaryInteger = 0b10001       // 17 in binary notation

let octalInteger = 0o21           // 17 in octal notation

let hexadecimalInteger = 0x11     // 17 in hexadecimal notation
```

Numeric Literals Cont.

- Numeric literals can contain extra formatting.
- Both integers and floats can be padded with extra zeros.
- Can contain underscores to help with readability.

```
let paddedDouble = 000123.456

let oneMillion = 1_000_000

let justOverOneMillion = 1_000_000.000_000_1
```

Type conversion

- Swift doesn't convert numeric values that covers different ranges or exceeds the range:

```
let cannotBeNegative: UInt8 = -1
// UInt8 cannot store negative numbers, and so this will report an error
let tooBig: Int8 = Int8.max + 1
// Int8 cannot store a number larger than its maximum value,
// and so this will also report an error
```

Type conversion Cont.

- You can't add two variables with different types without casting

```
1   let twoThousand: UInt16 = 2_000
2   let one: UInt8 = 1
3   let twoThousandAndOne = twoThousand + UInt16(one)
```

Floating-Point Numbers

- Swift provides two signed floating-point number types:
  - ✓ ***Double*** represents a 64-bit floating-point number
    used when floating-point values must be very large.
  - ✓ ***Float*** represents a 32-bit floating-point number
    used for numbers with smaller decimal points.

Floating-Point Numbers Cont.

- Double has a precision of at least 15 decimal digits, whereas the precision of Float can be as little as 6 decimal digits

Data types - Bool

- In swift we use Bool type to define a boolean values

- We can assign two values to type Bool
  - ✓true
  - ✓false

  Instead of (yes , no) in Objective-C

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

Data types - Character

- In Objective-C we had char type used to define variables of single character

- In swift we use Character

Strings

- To ***initialize*** an empty string this can be done by:
  - ✓ Assign an empty string literal to a variable.
  - ✓ Initialize a new String instance with initializer syntax:

```
1  var emptyString = ""                // empty string literal
2  var anotherEmptyString = String()   // initializer syntax
3  // these two strings are both empty, and are equivalent to each other
```

Strings Cont.

- Find out whether a String value is empty by checking its Boolean isEmpty property:

```
1  if emptyString.isEmpty {
2      print("Nothing to see here")
3  }
```

Strings Cont.

- String values can be constructed by passing an array of Character values as an argument to its initializer

```
let catCharacters: [Character] = ["C", "a", "t", "!", "😺"]

let catString = String(catCharacters)

print(catString)

// Prints "Cat!😺"
```

String Manipulation

- String **concatenation** is as simple as adding together two strings with the + operator

```
1   let string1 = "hello"
2   let string2 = " there"
3   var welcome = string1 + string2
```

String Manipulation Cont.

- You can also **append** a String value using += operator

```
1  var instruction = "look over"
2  instruction += string2
```

String Manipulation Cont.

- String literals can include the following special characters:
  - ✓ The escaped special characters
  - ✓ \0 (null character)
  - ✓ \\ (backslash)
  - ✓ \t (horizontal tab)
  - ✓ \n (line feed)
  - ✓ \r (carriage return)
  - ✓ \" (double quote)
  - ✓ \' (single quote)

String Manipulation Cont.

- You can append a character value to a string variable with append() method.

```
1  let exclamationMark: Character = "!"
2  welcome.append(exclamationMark)
```

String Manipulation Cont.

- To retrieve a **count** of the Character values in a string, use the count property of the string's characters property

```
let unusualMenagerie = "Koala , Snail , Penguin , Dromedary "

print("unusualMenagerie has \(unusualMenagerie.characters.count)
  characters")

// Prints "unusualMenagerie has 40 characters"
```

String Interpolation

- String interpolation is a way to construct a new String value from a mix of constants, variables, literals, and expressions by including their values inside a string literal.

```
let multiplier = 3

let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"

// message is "3 times 2.5 is 7.5"
```

Comparing Strings

- String and character equality is checked with the “equal to” operator (**==**) , the “not equal to” operator (**!=**) and (**>**) or (**<**)

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
```

Tuples

- Tuples group multiple values into a single compound value.
- The values within a tuple can be of any type and do not have to be of the same type as each other.

```
let http404Error = (404, "Not Found")

// http404Error is of type (Int, String), and equals (404, "Not Found")
```

- It can be described as a tuple of type (Int,String).

Tuples Cont.

- You can decompose a tuple's contents into separate constants or variables, which you then access as usual:

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// prints "The status code is 404"
print("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

Tuples Cont.

- If you only need some of the tuple's values, ignore parts of the tuple with an underscore (_) when you decompose the tuple

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// prints "The status code is 404"
```

Tuples Cont.

- Access the individual element values in a tuple using index numbers starting at zero:

```
1  print("The status code is \(http404Error.0)")
2  // prints "The status code is 404"
3  print("The status message is \(http404Error.1)"
4  // prints "The status message is Not Found"
```

Tuples Cont.

- You can name the individual elements in a tuple when the tuple is defined:

```
let http200Status = (statusCode: 200, description: "OK")
```

```
1  print("The status code is \(http200Status.statusCode)")
2  // prints "The status code is 200"
```

Arrays

- The type of a Swift array is written in full as Array<Element>
- You can also write the type of an array in shorthand form as [Element]
  - ✓ Element is the type of values the array is allowed to store

```
let someArray: Array<String> = ["Alex", "Brian", "Dave"]

let someArray: [String] = ["Alex", "Brian", "Dave"]
```

Creating an Array with a Default Value

- You pass this initializer with the number of items to be added to the new array (called count) and a default value of the appropriate type (called repeatedValue)

```
var threeDoubles = [Double](count: 3, repeatedValue: 0.0)

// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

Creating an Array by Adding Two Arrays

```
var anotherThreeDoubles = [Double](count: 3, repeatedValue: 2.5)

// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]
```

```
var sixDoubles = threeDoubles + anotherThreeDoubles

// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5,

  2.5, 2.5]
```

Modifying an Array

- You can append an array of one or more compatible items with the addition assignment operator (+=)
- You can add a new item to the end of an array by calling the array's append method:

```
var shoppingList = ["Eggs", "Milk"]
```

```
shoppingList.append("Flour")
// shoppingList now contains 3 items
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
```

Accessing & Modifying Array Elements

- Retrieve a value from the array by passing the index of the value you want to retrieve

```
var firstItem = shoppingList[0]

// firstItem is equal to "Eggs"
```

- change a range of values at once

```
shoppingList[4...6] = ["Bananas", "Apples"]

// shoppingList now contains 6 items
```

- It will remove elements from index 4 ,5 ,6 and replace it by bananas and apples

Iterating Over Array

- You can iterate over the entire set of values in an array with the for-in loop:

```
for item in shoppingList {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

Dictionary Type

- You can define dictionary in a way that is similar to the array

```
let someDictionary: [String: Int] = ["Alex": 31, "Paul": 39]

let someDictionary: Dictionary<String, Int> = ["Alex": 31, "Paul": 39]
```

Type Safe and Type Inference

Type Safety:

- Swift provide type checking while compiling.

Type Inference:

- Means if you assign a literal value of 42 to a new constant without saying what type it is, Swift ***infers*** that you want the constant to be an Int, because you have initialized it with a number

```
1  let meaningOfLife = 42
2  // meaningOfLife is inferred to be of type Int
```

Type Safe and Type Inference Cont.

- Swift always chooses Double (rather than Float) when inferring the type of floating-point numbers.

```
let pi = 3.14
```

- Swift inference that pi will be double.

Type Aliases

- Type aliases define an alternative name for an existing type.
- You define type aliases wit the type alias keyword.

```
typealias AudioSample = UInt16
```

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

nil

- Swift's nil is not the same as nil in Objective-C
  - ✓ In Objective-C, nil is a pointer to a non existent object
  - ✓ In Swift, nil is not a pointer - it is the absence of a value of a certain type

- You can set an optional variable or constants values to nil to indicates the meaning of no value

Note:

- ✓ You can't use nil with non optional variables or constants.

What is the Problem ?!

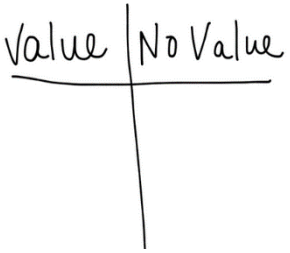
- In C, it is possible to create a variable without giving it a value

```
Int x;
```

- If you were to try to use the value before assigning it a value, you would get undefined behavior !!

Optional Cont.

- Swift create the type Optional



- You use optional in situations where a value may be absent

- The type Optional<Wrapped> is an enumeration with two cases
  - ✓ None ***(nil)***
  - ✓ Some(Wrapped) ***value***

```
enum Optional {
    case None
    case Some(T)
}
```

Optional Cont.

- You declare an optional by using ? Or Optional keyword followed by the type

```
var optionalInteger: Int?

var optionalInteger: Optional<Int>
```

Note:

- There is no whitespace may appear between the type and the ?

Optional Cont.

- Any type can be explicitly declared to be (or implicitly converted to) an optional type not only objects.

Note:

✓ Default value for an optional variable or property is nil

```
1  var surveyAnswer: String?
2  // surveyAnswer is automatically set to nil
```

Unwrapping optional

- Before you use the value from an Optional you must first ***"unwrap"*** it.
- You can unwrap an optional in both a ***"safe"*** and ***"unsafe"*** way.
- The safe way is to use Optional Binding
- Once you're sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation mark (!) to the end of the optional's name.

Optional Binding

- You use optional binding to check if the optional contains a value or not.
- If it does contain a value, unwrap it and put it into a temporary constant or variable.
- Optional binding can be used with if and while statements to check for a value inside an optional.
- Then extracting that value into a constant or variable, as part of a single action.

Optional Binding Cont.

```
if let actualNumber = Int(possibleNumber) {
    print("\'\(possibleNumber)\' has an integer value of \
  (actualNumber)")
} else {
    print("\'\(possibleNumber)\' could not be converted to an integer")
}
// prints "'123' has an integer value of 123"
```

Forced Unwrapping

- Sometimes you know for sure that a variable holds an actual value and you can assert that with ***forced Unwrapping*** by using an exclamation point (!)

```
let possibleString : String? = "Hello"

println(possibleString!)
```

Implicitly Unwrapped Optionals

- Sometimes it is clear from a program's structure that an optional will always have a value.
- In these cases, it is useful to remove the need to check and unwrap the optional's value every time it is accessed.
- These kinds of optionals are defined as implicitly unwrapped optionals.

Implicitly Unwrapped Optionals Cont.

- Implicity Unwrapped optional is an optional that does not need to be unwrapped because it is done implicitly.

- Implicitly unwrapped optionals are useful when an optional's value is confirmed to exist.

- You write an implicitly unwrapped optional by placing an exclamation mark (String!) rather than a question mark (String?).

Implicitly Unwrapped Optionals Cont.

```
let possibleString: String? = "An optional string."

let forcedString: String = possibleString! // requires an exclamation

  mark
```

```
let assumedString: String! = "An implicitly unwrapped optional string."

let implicitString: String = assumedString // no need for an exclamation

  mark
```

Control flow

What is a new ?!

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
```

Note:

- ✓ There is no ( ) surrounding the condition !

for - in

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

for - in Cont.

- If you don't need each value from a sequence, you can ignore the values by using an underscore in place of a variable name

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
// prints "3 to the power of 10 is 59049"
```

repeat - while

- The repeat-while loop in Swift is analogous to a do-while loop in other languages

```
repeat {
    statements
} while condition
```

switch

- Every switch statement consists of multiple possible cases.
- Swift provides several ways for each case to specify more complex matching patterns.
- In contrast with C and Objective-C , switch statements in Swift do not fall through the bottom of each case. So break is not required in Swift.

switch Examples

```
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}
// prints "e is a vowel"
```

Switch Examples Cont.

- Values in switch cases can be checked for their inclusion in an interval

```
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
default:
    naturalCount = "many"
}
```

Functions

Functions

- In Swift language Function is defined by the "func" keyword.

- Return values are followed by ->

Function definition & calling

```
func sayHello(personName: String) -> String {
    let greeting = "Hello, " + personName + "!"
    return greeting
}
```

```
print(sayHello("Anna"))
// prints "Hello, Anna!"
print(sayHello("Brian"))
// prints "Hello, Brian!"
```

Function definition & calling Cont.

- Function without parameters

```
func sayHelloWorld() -> String {
```

- Function with multiple parameters

```
func sayHello(personName: String, alreadyGreeted: Bool) -> String {
```

- Function without return value

```
  func greet(person: String) {
```

Function definition & calling Cont.

- Function with Multiple Return Values

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
```

```
return (currentMin, currentMax)
```

Function with optional return

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Functions Cont.

- You can define a default value for any parameter in a function.
- Functions can also be written within other functions to encapsulate useful functionality within a nested function scope.
- Parameters of function can have internal, external names.

Default Parameter Values

- You can define a default value for any parameter in a function by assigning a value to the parameter after that parameter's type

```
func someFunction(parameterWithDefault: Int = 12) {
```

```
someFunction(6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12
```

Functions with Internal and External names

- If you provide an external parameter name for a parameter, that external name must always be used when you call the function

```
func someFunction(externalParameterName localParameterName: Int) {
    // function body goes here, and can use localParameterName
    // to refer to the argument value for that parameter
}
```

Functions with Internal and External names Cont.

```
func sayHello(to person: String, and anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}
```

```
print(sayHello(to: "Bill", and: "Ted"))
// prints "Hello Bill and Ted!"
```

Omitting External Parameter Names

- If you do not want to use an external name for the second or subsequent parameters of a function, write an underscore (_) instead of an explicit external name

```
func someFunction(firstParameterName: Int, _ secondParameterName: Int) {
```

- Calling:

```
someFunction(1, 2)
```

Functions Type

- Each function has a type
- Type of a function consists of it's arguments types and the return type
- We can assign function type into variables or constants

Functions Type

- Function type for the previous two functions is:

```
(Int, Int) -> Int.
```

```
func printHelloWorld() {
    print("hello, world")
}
```

- Function type printHelloWorld() is:

```
() -> Void
```

Using Function Types

- You use function types just like any other types in Swift.
- you can define a constant or variable to be of a function type.
- Function type can passed as return type to another function .

Using Function Types

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

This can be read as:

- Define a variable called mathFunction, which has a type of a function that takes two Int values, and returns an Int value.
- Set this new variable to refer to the function called addTwoInts

```
print("Result: \(mathFunction(2, 3))")

// prints "Result: 5"
```

Function Types as parameter types

- You can use a function type such as `(Int, Int) -> Int` as a parameter type for another function.

```
func printMathResult(mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int)
 {
```

- You can pass any function of that type as the argument for this first parameter.

Function Types as parameter types Cont.

```
func printMathResult(mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int)
  {
     print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "Result: 8"
```

Function Types as return types

- You can use a function type as the return type of another function

- You do this by writing a complete function type immediately after the return arrow (->) of the returning function

Function Types as return types Cont.

```
func stepForward(input: Int) -> Int {
    return input + 1
}
func stepBackward(input: Int) -> Int {
    return input - 1
}
```

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
```

Nested Functions !

- Nested functions are hidden from the outside world by default
- It still can be called and used by their enclosing function
- An enclosing function can also return one of its nested functions

Nested Functions ! Cont.

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
```

Variadic Parameters

- A variadic parameter accepts zero or more values of a specified type.
- use a variadic parameter to specify that the parameter can be passed a varying numbers of input values when the function is called.

```
func arithmeticMean(numbers: Double...) -> Double {
```

- Calling:

```
arithmeticMean(1, 2, 3, 4, 5)
arithmeticMean(3, 8.25, 18.75)
```

In-Out parameters

- Variable parameters, as described above, can only be changed within the function itself
- If you want a function to modify a parameter's value after the function call has ended
- You write an in-out parameter by placing the inout keyword at the start of its parameter

In-Out parameters Cont.

- You place an ampersand (&) directly before a variable's name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

- An in-out parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value

In-Out parameters Cont.

```
func swap (a1 : inout Int , b1 : inout Int){


    let temp = a1
    a1 = b1
    b1 = temp


}
```

```
var num1 = 5
var num2 = 6

swap(&num1, &num2)

println(num1)  // 6
println(num2)  // 5
```

Assignments

1.Factorial

- Using playground write a function called: getFactorial that takes an Int value and return the factorial of that number.

2.Power

- Using playground Write a function called: clacPower that takes two numbers base and power and returns the base to that power as an Int using two ways (recursion , loops).

3.Sorting

- Using playground write a function called: sortArray that takes an array and returns the sorted version of it

Note:

- The parameter of the function is constant by default if you want to make it variable write the keyword var before it while declaring your function type

4.Min & Max

- Using playground write a function called: calMinAndMax that takes an array and returns the min and max numbers in it.

Note:

- Use return of the function should be tuple

5.Swap

- Using playground write a function called: swap that takes two numbers and swap them using inout parameters.

Lecture 2

Agenda

- Classes & Structures
- Subscripts
- Properties
  - ✓ Stored properties
  - ✓ Lazy stored property
  - ✓ Computed property
  - ✓ Property observer
- Inheritance
  - ✓ Overriding methods
  - ✓ Overriding properties
  - ✓ Preventing Overriding
- Initialization & Deinitialization

Classes & Structures

Classes & Structures

- In Objective C, you create separate interface (.h) and implementation (.m) files for classes.
- Swift does not require you to create separate interface and implementation files.
- You define a class or a structure in a single file, and the external interface to that class or structure is automatically made available.

Comparing Classes and Structures

- Classes and structures in Swift have many things in common. Both can:
  - ✓ Define properties to store values.
  - ✓ Define methods to provide functionality.
  - ✓ Define subscripts to provide access to their values using subscript syntax.
  - ✓ Define initializers to set up their initial state.
  - ✓ Be extended to expand their functionality beyond a default implementation.
  - ✓ Conform to protocols to provide standard functionality of a certain kind.

Comparing Classes and Structures Cont.

- Classes have additional capabilities that structures do not:
  - ✓ Inheritance enables one class to inherit the characteristics of another.
  - ✓ Type casting enables you to check and interpret the type of a class instance at runtime.
  - ✓ Deinitializers enable an instance of a class to free up any resources it has assigned.
  - ✓ Reference counting allows more than one reference to a class instance.

Comparing Classes and Structures Cont.

- Structure are value type but classes are reference type.
- A value type is a type whose value is copied when it is assigned to a variable or constant, or a function.
- Reference types are not copied when they are passed. Rather than a copy, a reference to the same existing instance is used instead.

Definition Syntax

- Classes and structures have a similar definition syntax.
- You introduce classes with the class keyword and structures with the struct keyword.

```
class SomeClass {

    // class definition goes here

}

struct SomeStructure {

    // structure definition goes here

}
```

Definition Syntax Cont.

- Example of a structure definition and a class definition:

```
1   struct Resolution {
2       var width = 0
3       var height = 0
4   }
5   class VideoMode {
6       var resolution = Resolution()
7       var interlaced = false
8       var frameRate = 0.0
9       var name: String?
10  }
```

Class and Structure Instances

- The syntax for creating instances is very similar for both structures and classes:

```
1   let someResolution = Resolution()
2   let someVideoMode = VideoMode()
```

Accessing Properties

- You can access the properties of an instance using ***dot syntax***. In dot syntax, you write the property name immediately after the instance name, separated by a period **(.)**, without any spaces.

```
print("The width of someResolution is \(someResolution.width)")

// prints "The width of someResolution is 0"
```

Accessing Properties Cont.

- You can drill down into sub-properties, such as the width property in the resolutionproperty of a VideoMode:

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")

// prints "The width of someVideoMode is 0"
```

Memberwise Initializers for Structure Types

- All structures have an automatically-generated memberwise initializer, which you can use to initialize the member properties of new structure instances

```
let vga = Resolution(width: 640, height: 480)
```

Subscripts

Subscripts

- Accessing the element members of a collection, sequence and a list in Classes, Structures and Enumeraions are carried out with the help of subscripts

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

Subscripts Cont.

```
class daysofaweek {
    private var days = ["Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "saturday"]
    subscript(index: Int) -> String {
        get {
            return days[index]
        }
        set(newValue) {
            self.days[index] = newValue
        }
    }
}
var p = daysofaweek()

println(p[0])
println(p[1])
println(p[2])
println(p[3])
```

```
Sunday
Monday
Tuesday
Wednesday
```

Properties

Properties

- *Properties* associate values with a particular class, structure, or enumeration
- Types of properties :
  - ✓ Stored
  - ✓ Computed

Stored Property	Computed Property
Store constant and variable values as instance	Calculate a value rather than storing the value
Provided by classes and structures	Provided by classes, enumerations and structures

Stored Properties

- Swift introduces stored property concept to store the instances of constants and variables
- A stored property is a constant or variable that is stored as part of an instance of a particular class or structure
- Stored properties are defined by the let keyword for constants and var for variables

Stored Properties Cont.

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)

rangeOfThreeItems.firstValue = 6
```

Stored (lazy) property

- A lazy stored property is a property whose won't calculate the initial values when the variable is initialized for the first time until the first time it is used

- You indicate a lazy stored property by writing the lazy modifier before its declaration

Stored (lazy) property Cont.

- The DataImporter instance for the importer property has not yet been created

```
let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
```

Stored (lazy) property Cont.

- Now when we use the property, It will be created

```
1  print(manager.importer.fileName)
2  // the DataImporter instance for the importer property has now been created
3  // Prints "data.txt"
```

Computed Property

- Rather than storing the values computed properties provide a getter and an optional setter to retrieve and set other properties and values indirectly

- A computed property with a getter but no setter is known as a read-only computed property. A read-only computed property always returns a value, and can be accessed through dot syntax

Computed Property example

```
class sample {
    var no1 = 0.0, no2 = 0.0
    var length = 300.0, breadth = 150.0

    var middle: (Double, Double) {
        get{
            return (length / 2, breadth / 2)
        }
        set(axis){
            no1 = axis.0 - (length / 2)
            no2 = axis.1 - (breadth / 2)
        }
    }
}

var result = sample()
println(result.middle)
result.middle = (0.0, 10.0)

println(result.no1)
println(result.no2)
```

```
(150.0, 75.0)

-150.0

-65.0
```

Read Only Computed Property example

- You can simplify the declaration of a read-only computed property by removing the get keyword and its braces:

```
1  struct Cuboid {
2      var width = 0.0, height = 0.0, depth = 0.0
3      var volume: Double {
4          return width * height * depth
5      }
6  }
7  let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
8  print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
9  // prints "the volume of fourByFiveByTwo is 40.0"
```

Property Observer

- Property observers observe and respond to changes in a property's value
- Property observers are called every time a property's value is set
- You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass

Property Observer Cont.

- You have the option to define either or both of these observers on a property:
  - ✓willSet is called just before the value is stored
  - ✓didSet is called immediately after the new value is stored.

Property Observer Cont.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
```

Property Observer Cont.

```
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

Inheritance

Inheritance

- A class can inherit methods, properties, and other characteristics from another class
- Swift helps to ensure your overrides are correct by checking that the override definition has a matching superclass definition.
- Classes can add property observers to inherited properties in order to be notified when the value of a property changes

Defining a Base Class

- Any class that does not inherit from another class is known as a base class

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necessarily make a
  noise
    }
}
```

Defining a sub Class

- To indicate that a subclass has a superclass, write the subclass name before the superclass name, separated by a colon.

```
class Bicycle: Vehicle {

    var hasBasket = false

}
```

Overriding Methods

- To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the override keyword
- The override keyword also prompts the Swift compiler to check that your overriding class's superclass

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

Overriding Methods Cont.

- It is sometimes useful to use the existing superclass implementation as part of your override

- You access the superclass version of a method, property, or subscript by using the super prefix
  - ✓ super.someMethod()
  - ✓ super.someProperty
  - ✓ super[someIndex]

Overriding Properties

- You can override an inherited instance or type property to provide your own custom getter and setter for that property

- Add property observers to enable the overriding property to observe when the underlying property value changes

Overriding Properties Cont.

- You can provide a custom getter (and setter, if appropriate) to override *any* inherited property.

- You must always state both the name and the type of the property you are overriding, to enable the compiler to check that your override matches a superclass property with the same name and type.

Overriding Properties Cont.

- You can present an inherited read-write property as a read-only property by providing a getter method only in your subclass

- You cannot, however, present an inherited read-only property as a read-write property.

Preventing Overriding

- You can prevent a method, property, or subscript from being overridden by marking it as final. Do this by writing the final modifier before the method, property, or subscript.

- You can mark an entire class as final by writing the final modifier before the class keyword in its class definition.

Initialization

Initialization

- Initialization is the process of preparing an instance of a class, structure, or enumeration for use.
- You implement this initialization process by defining initializers.
- Swift initializer differs from Objective-C that it does not return a value.
- Instances of class types can also implement a deinitializer.

Initialization Cont.

- Classes and structures must set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created.

- You can provide initialization parameters as part of an initializer's definition.

Initializers

- Initializers are called to create a new instance of a particular type

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
```

```
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// prints "The default temperature is 32.0° Fahrenheit"
```

Initializers Cont.

External Name

Internal Name

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

Initializers Cont.

- Swift provides an automatic external name for every parameter in an initializer if you don't provide an external name yourself.

- Swift provides a default initializer for any structure or class.

Optional Property Types

- If you have stored property that is logically allowed to have “no value” declare the property with an optional type

```
class ShoppingListItem {

    var name: String?

    var quantity = 1

    var purchased = false

}
```

- Properties of optional type are automatically initialized with a value of nil

Member wise Initializers for Structure

- Structure types automatically receive a *member wise initializer* if they do not define any of their own custom initializers

- The memberwise initializer is a shorthand way to initialize the member properties of new structure instances

Member wise Initializers for Structure

- The Size structure automatically receives an init(width:height:) memberwise initializer, which you can use to initialize a new Size instance

Initializer Inheritance and Overriding

- Unlike subclasses in Objective-C, Swift subclasses do not inherit their superclass initializers by default

- If you want a custom subclass to present one or more of the same initializers as its superclass, you can provide a custom implementation of those initializers within the subclass

Deinitialization

- A deinitializer is called immediately before a class instance is deallocated
- You write deinitializers with the deinit keyword
- Deinitializers are only available on class types

```
deinit {
    // perform the deinitialization
}
```

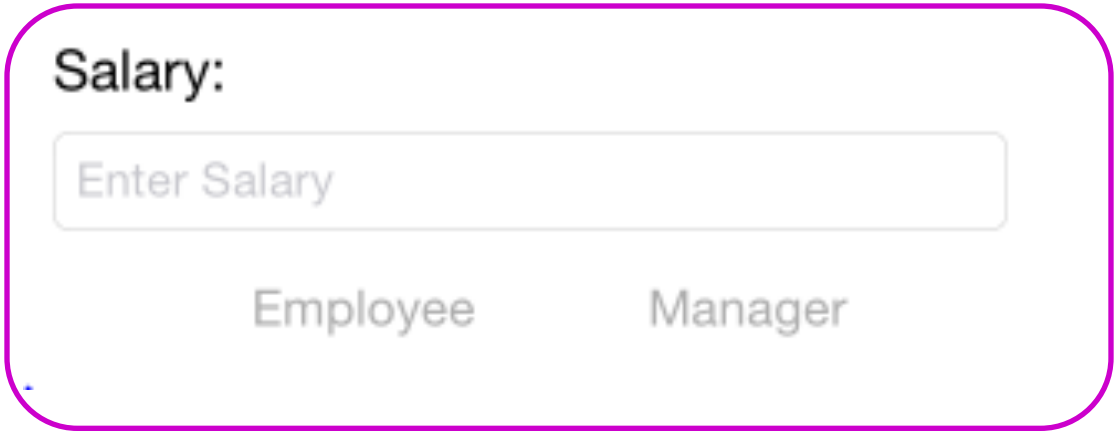
Deinitialization Cont.

- Class definitions can have at most one deinitializer per class
- The deinitializer does not take any parameters and is written without parentheses
- Deinitializers are called automatically, just before instance deallocation takes place

Assignments

1.Employee-Manager

- Make a class Person which have a getSalary func
- Make class Manager and Employee that inheret from Person and override getSalary method
- Make UI to calculate the salary of both



2.Movie List

- Make Movie TableViewController that shows 5 static Movies and when select specific movie you should present its data in another viewContrller.
- Movie Attributes:
  - ✓title : String
  - ✓image : String
  - ✓rating : Folat
  - ✓releaseYear : Int
  - ✓genre : [String]

Lecture 3

Agenda

- Closures
- Protocols
- Delegation
- Enumerations
- Extensions

Closures

Closures

- Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.
- Closures can capture and store references to any constants and variables from the context in which they are defined.
- Function is special case of closures.

Closures Syntax

- Closures are typically enclosed in curly braces { } and are defined by a function type ( ) -> ( )
- The -> separates the arguments and the return type, followed by the in keyword which separates the closure header from its body

```
{ (parameters) -> return type in
    statements
}
```

Closures Cont.

- Instead of representing the whole function declaration and name constructs we can use closures.

```
let devide = {(val1 : Int , val2 : Int) -> Int in

    return val1 / val2

}
```

Closures Example

- The sort(_:) method accepts a closure that takes two arguments of the same type as the array's contents, and returns a Bool value to say whether the first value should appear before or after the second value once the values are sorted.
- The sorting closure needs to return true if the first value should appear before the second value, and false otherwise.

Closures Example Cont.

- This example is sorting an array of String values, and so the sorting closure needs to be a function of type (String, String) -> Bool.

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

```
1  func backwards(s1: String, _ s2: String) -> Bool {
2      return s1 > s2
3  }
4  var reversed = names.sort(backwards)
5  // reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

Closures Example Cont.

- The example below shows a closure expression version of the backwards(_:_:) function from earlier:

```
1  reversed = names.sort({ (s1: String, s2: String) -> Bool in
2      return s1 > s2
3  })
```

- Because the body of the closure is so short, it can even be written on a single line:

```
reversed = names.sort( { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

Closures Example Cont.

Inferring Type From Context

- Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns.

```
reversed = names.sort( { s1, s2 in return s1 > s2 } )
```

Implicit Returns from Single-Expression Closures

- Single-expression closures can implicitly return the result of their single expression by omitting the return keyword from their declaration, as in this version of the previous example:

```
reversed = names.sort( { s1, s2 in s1 > s2 } )
```

Closures Example Cont.

Shorthand Argument Names

- Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names $0, $1, $2, and so on.

```
reversed = names.sort( { $0 > $1 } )
```

Closures Example Cont.

Demo

Protocols

Protocol

- A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.
- The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements.
- Any type that satisfies the requirements of a protocol is said to ***conform*** to that protocol.

Protocol Syntax

- You define protocols in a very similar way to classes, structures, and enumerations:

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Protocol Syntax Cont.

- Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

```
1  struct SomeStructure: FirstProtocol, AnotherProtocol {
2      // structure definition goes here
3  }
```

Protocol Syntax Cont.

- If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

```
1  class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
2      // class definition goes here
3  }
```

Property Requirements

- A protocol can require any conforming type to provide an instance property or type property with a particular name and type.
- The protocol doesn't specify whether the property should be a stored property or a computed property, it only specifies the required property name and type.
- The protocol also specifies whether each property must be gettable or gettable *and* settable.

Property Requirements Cont.

- Property requirements are always declared as variable properties, prefixed with the var keyword.
- Gettable and settable properties are indicated by writing { get set } after their type declaration, and gettable properties are indicated by writing { get }.

```
1  protocol SomeProtocol {
2      var mustBeSettable: Int { get set }
3      var doesNotNeedToBeSettable: Int { get }
4  }
```

Method Requirements

- Protocols can require specific instance methods and type methods to be implemented by conforming types.
- These methods are written as part of the protocol's definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body.

```
1   protocol SomeProtocol {
2       static func someTypeMethod()
3   }
```

Method Requirements Cont.

- The following example defines a protocol with a single instance method requirement:

```
1   protocol RandomNumberGenerator {
2       func random() -> Double
3   }
```

Method Requirements Cont.

- Here's an implementation of a class that adopts and conforms to the RandomNumberGenerator protocol.

```
1  class LinearCongruentialGenerator: RandomNumberGenerator {
2      var lastRandom = 42.0
3      let m = 139968.0
4      let a = 3877.0
5      let c = 29573.0
6      func random() -> Double {
7          lastRandom = ((lastRandom * a + c) % m)
8          return lastRandom / m
9      }
10 }
```

Protocol Inheritance

- A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits.
- The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
1  protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
2      // protocol definition goes here
3  }
```

Delegation

Delegation

- *Delegation* is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type.
- This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated.

Delegation Cont.

- Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

Enumerations

Enumerations

- An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.
- Enumerations in Swift are much more flexible, and do not have to provide a value for each case of the enumeration.
- If a value (known as a “raw” value) *is* provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type

Enumeration Syntax

- You introduce enumerations with the enum keyword and place their entire definition within a pair of braces:

```
1   enum SomeEnumeration {
2       // enumeration definition goes here
3   }
```

Enumeration Syntax Cont.

- Here's an example for the four main points of a compass:

```
1  enum CompassPoint {
2      case North
3      case South
4      case East
5      case West
6  }
```

Enumeration Syntax Cont.

- The values defined in an enumeration (such as North, South, East, and West) are its enumeration cases. You use the case keyword to introduce new enumeration cases.

- Multiple cases can appear on a single line, separated by commas:

```
1   enum Planet {
2       case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
3   }
```

Enumeration Syntax Cont.

- Each enumeration definition defines a brand new type.
- Like other types in Swift, their names (such as CompassPoint and Planet) should start with a capital letter.
- Give enumeration types singular rather than plural names, so that they read as self-evident:

```
var directionToHead = CompassPoint.West
```

Enumeration Syntax Cont.

- The type of directionToHead is inferred when it is initialized with one of the possible values of CompassPoint. Once directionToHead is declared as a CompassPoint, you can set it to a different CompassPoint value using a shorter dot syntax:

```
directionToHead = .East
```

Matching Enumeration Values with a Switch Statement

- You can match individual enumeration values with a switch statement:

```
1  directionToHead = .South
2  switch directionToHead {
3  case .North:
4      print("Lots of planets have a north")
5  case .South:
6      print("Watch out for penguins")
7  case .East:
8      print("Where the sun rises")
9  case .West:
10     print("Where the skies are blue")
11 }
12 // prints "Watch out for penguins"
```

Implicitly Assigned Raw Values

- When you're working with enumerations that store integer or string raw values, you don't have to explicitly assign a raw value for each case.
- When you don't, Swift will automatically assign the values for you.
- For instance, when integers are used for raw values, the implicit value for each case is one more than the previous case.
- If the first case doesn't have a value set, it's value is 0.

Implicitly Assigned Raw Values Cont.

- The enumeration below is a refinement of the earlier Planet enumeration, with integer raw values to represent each planet's order from the sun:

```
1  enum Planet: Int {
2      case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
3  }
```

- In the example above, Planet.Mercury has an explicit raw value of 1, Planet.Venus has an implicit raw value of 2, and so on.

Implicitly Assigned Raw Values Cont.

- When strings are used for raw values, the implicit value for each case is the text of that case's name.
- The enumeration below is a refinement of the earlier CompassPoint enumeration, with string raw values to represent each direction's name:

```
1   enum CompassPoint: String {
2       case North, South, East, West
3   }
```

- In the example above, CompassPoint.South has an implicit raw value of "South", and so on.

Implicitly Assigned Raw Values Cont.

- You access the raw value of an enumeration case with its rawValue property:

```
1  let earthsOrder = Planet.Earth.rawValue
2  // earthsOrder is 3
3
4  let sunsetDirection = CompassPoint.West.rawValue
5  // sunsetDirection is "West"
```

Extensions

Extensions

- *Extensions* add new functionality to an existing class, structure, enumeration, or protocol type.
- This includes the ability to extend types for which you do not have access to the original source code.
- Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions do not have names.)

Extensions Cont.

- Extensions in Swift can:
  - ✓ Add computed instance properties and computed type properties
  - ✓ Define instance methods and type methods
  - ✓ Provide new initializers
  - ✓ Define subscripts
  - ✓ Define and use new nested types
  - ✓ Make an existing type conform to a protocol

Extensions Syntax

- Declare extensions with the extension keyword:

```
1   extension SomeType {
2       // new functionality to add to SomeType goes here
3   }
```

Extensions Syntax Cont.

```
extension String{

    func myMethod()->String{

        return "Extension Method"

    }

}
```

```
var str : String = ""
println(str.myMethod())  // Extension Method
```

Assignments

1.Movie List

- Update your Movie list app to add **+** button on the navigation bar which present view allow you to add new movie and when you press **Done** button in this view the list view updated with the new movie.

Lecture 4

Agenda

- Networking using URLSession
- NetworkActivityIndicator

Networking using URLSession

URLSession

- URLSession is a replacement for NSURLConnection, and similarly it is both a distinct class and also a group of related APIs.
- The URLSession class and related classes provide an API for downloading content.
- It able to do everything NSURLConnection could do but also adds improvements on top.

URLSession Cont.

URLSession Vs NSURLConnection

- URLSession and NSURLConnection have a lot in common in terms of how they work, at a fundamental level, they are quite different

- Apple created URLSession to resemble the general concepts of NSURLConnection

Why Use URLSession?

- Background uploads and downloads
  - ✓You can start a download of a large image or file, close the app and the download will continue until it completes.
- Ability to pause and resume networking operations.
- Uploads and downloads through the file system.
- Improved authentication handling.

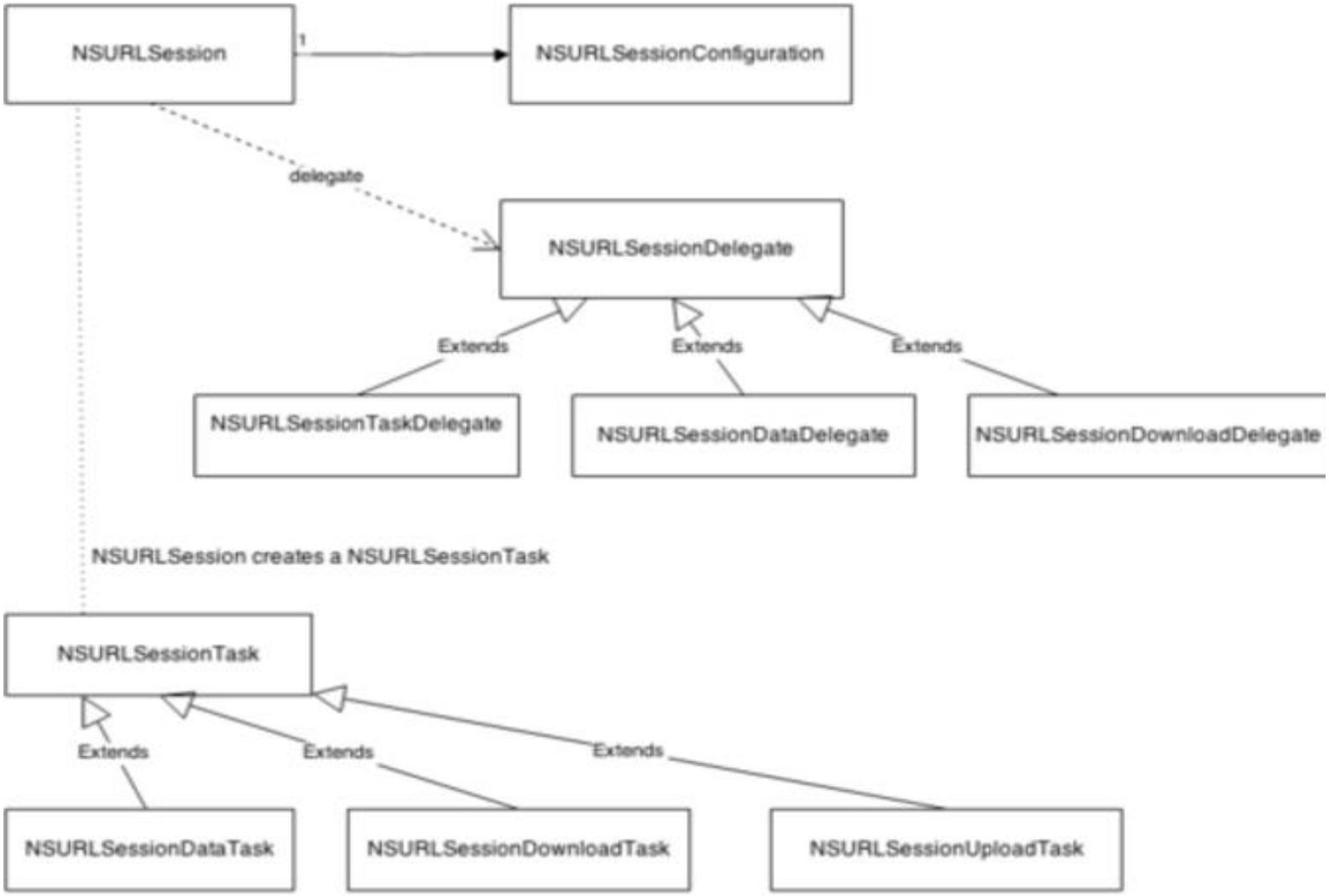
Why Use URLSession? Cont.

- Configurable container
  - ✓URLSessionConfiguration is the first class you'll probably work with in the URLSession API.
  - ✓ It is a class that allows you to do exactly what its name suggests: configure your URLSession with connection/HTTP policies, caching behavior, maximum number of connections, custom headers, etc.

Networking with URLSession API

- URLSession is the key component sending requests and receiving responses.
- The configuration of the session object, is handled by an instance of the URLSessionConfiguration class
- The basic unit of work when working with NSURLSession is the task an instance of URLSessionTask

Networking with URLSession API Cont.



Networking with URLSession API Cont.

- The basic unit of work when working with URLSession is the task.

- There are three types of tasks
  - ✓ URLSessionDataTask
  - ✓ URLSessionUploadTask
  - ✓ URLSessionDownloadTask

Networking with URLSession API Cont.

- URLSessionDataTask: Use this task for HTTP GET requests to retrieve data from servers to memory.

- URLSessionUploadTask: Use this task to upload a file from disk to a web service, typically via a HTTP POST or PUT method.

- URLSessionDownloadTask: Use this task to download a file from a remote service to a temporary file location.

Data task example

```
let url = URL(string: "http://api.androidhive.info/
    json/movies.json")

let request = URLRequest(url: url!)

let session = URLSession(configuration:
    URLSessionConfiguration.default)

let task = session.dataTask(with: request) { (data,
    response , error) in

    do{

        var json = try  JSONSerialization.jsonObject
            (with: data!, options: .allowFragments) as!
            Dictionary<String , String>

        // Manipulate your JSON
    }catch{

        print("Error")
    }

}

task.resume()
```

1

2

3

4

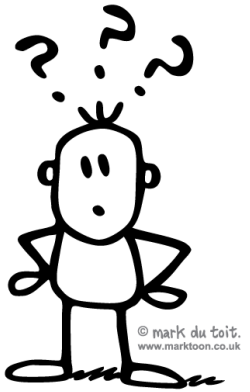
5

6

Data task example Cont.

Note:

You can't do UI things inside your task because you are in a different thread



Data task example Cont.

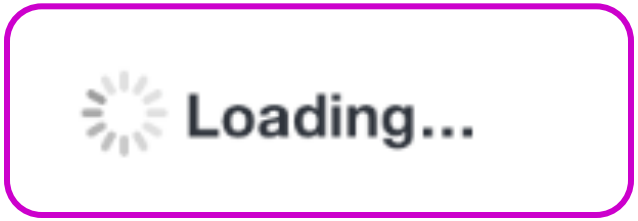
- You can simply dispatch back to the main queue

```
DispatchQueue.main.async {

    //update your GUI here

}
```

NetworkActivityIndicator

Display Activity Indicator

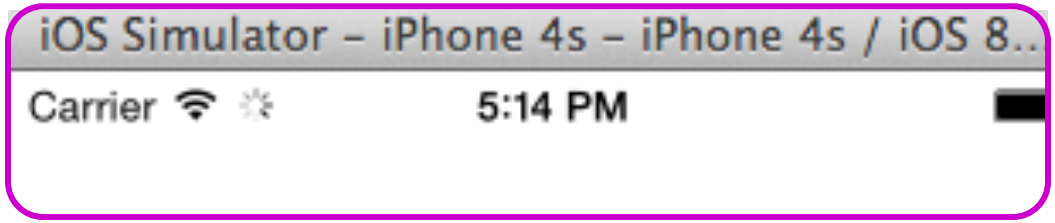
- The Activity Indicator can be displayed in the status bar to inform the user that some kind of network activity is in progress.



Display Activity Indicator Cont.

- You can simply activate the indicator by single line of code

```
UIApplication.sharedApplication().networkActivityIndicatorVisible = true
```



- To stop the indicator just change the boolean value to false

Display Activity Indicator Cont.

- You can customize coloring , positioning , style of the indicator

```
var networkIndicator = UIActivityIndicatorView
    (activityIndicatorStyle: UIActivityIndicatorViewStyle.Gray)
    networkIndicator.center = view.center
    networkIndicator.startAnimating()
    view.addSubview(networkIndicator)
```

- To stop the activity indicator

```
networkIndicator.stopAnimating()
```

Assignments

1.Movie List

- Update your Movie list app to get movies from the following webservice:

  http://api.androidhive.info/json/movies.json

- Display the movie detail in another view controller when select it in the table view.

Lecture 5

Agenda

- Core Data

Core Data

What Is Core Data?

- Core Data is a framework that you use to manage the model layer objects in your application.

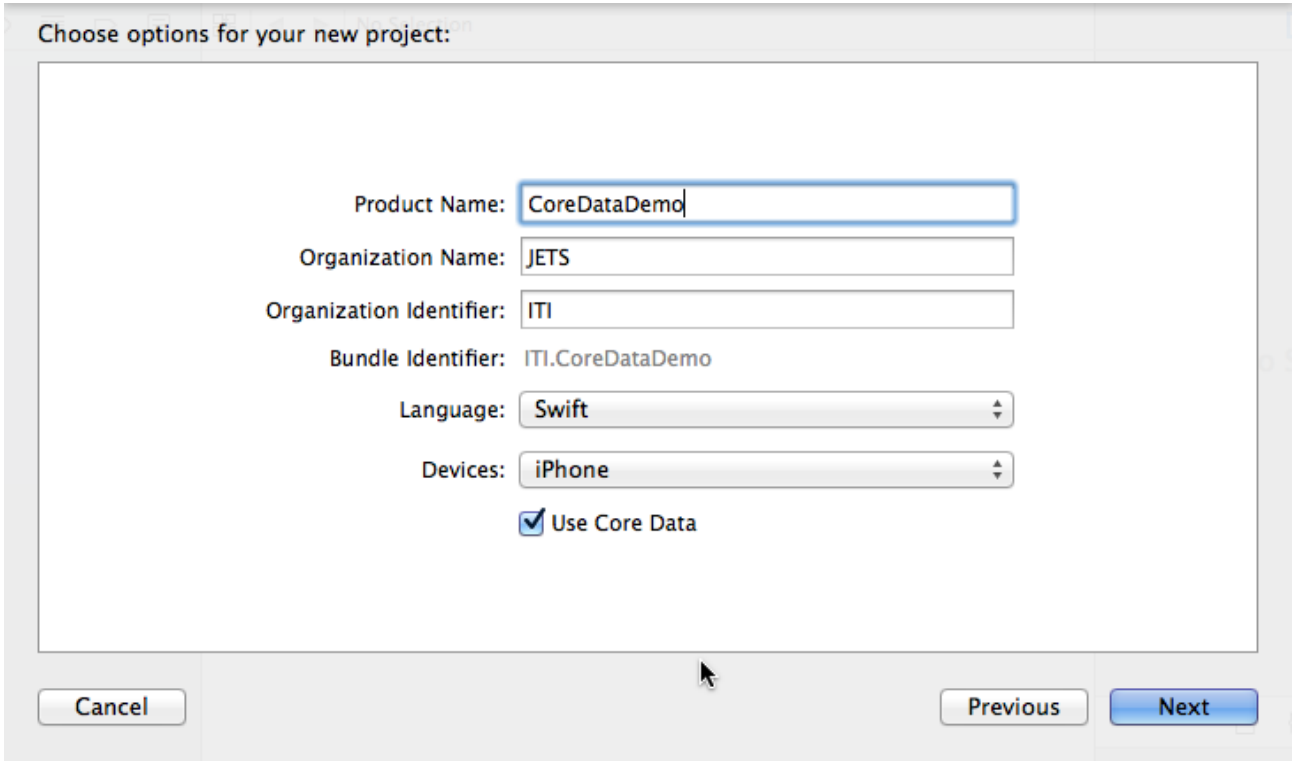
- It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence.

What Is Core Data? Cont.

- Object-oriented database.
- Very powerful framework in iOS.
- It's a way of creating an object graph backed by a database.
- Usually backed by SQL.

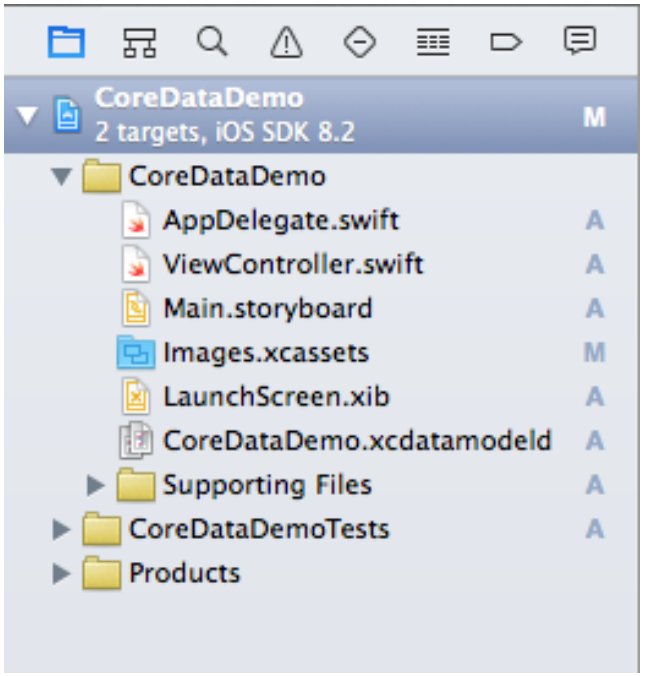
How does it work?

- When you start a new project in Xcode and open the template selection dialog, select the Use Core Data checkbox.



How does it work? Cont.

- A source file for the Core Data model is created as part of the template.
- That source file will have the extension .xcdatamodeld.

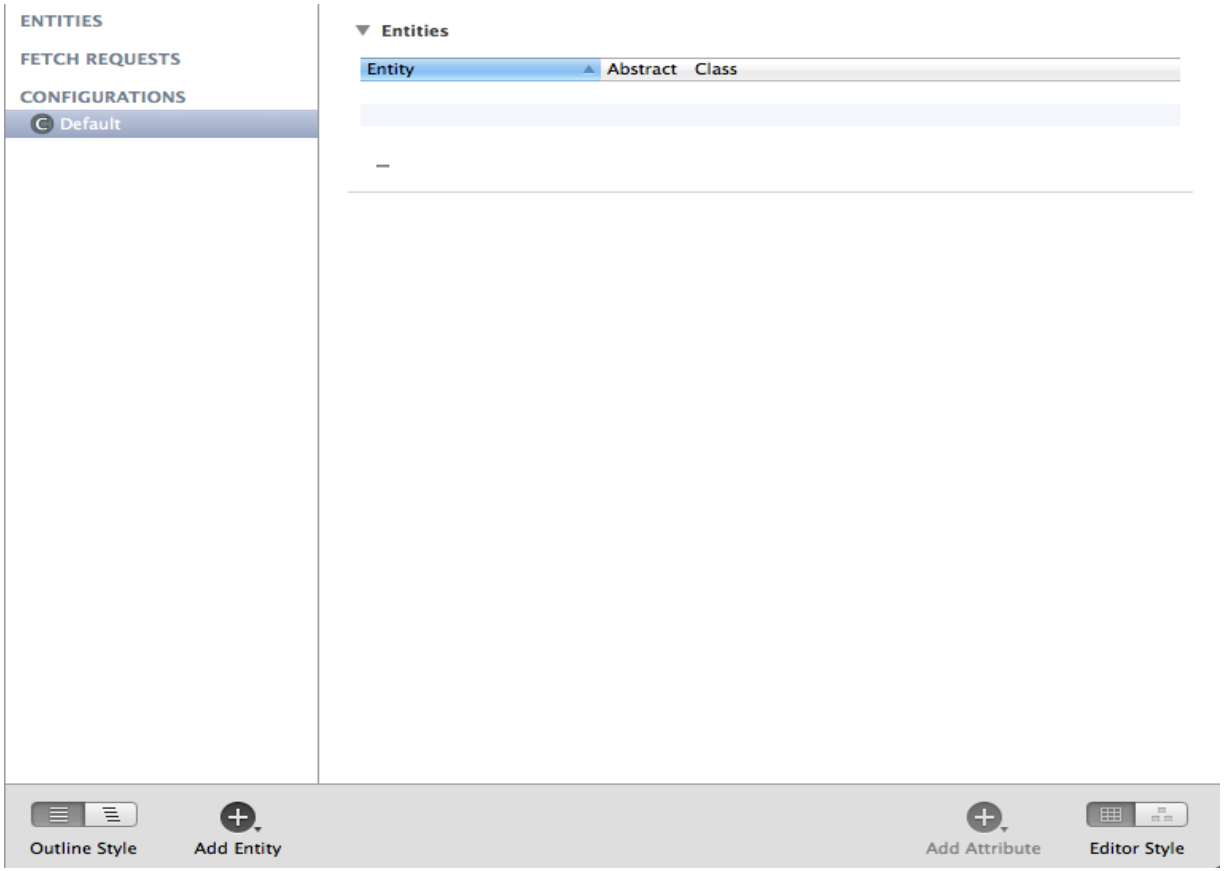


Entity

- An ***entity*** is a class definition in Core Data.
- In our example the entity is a ***Movie***.
- In a relational database, an entity corresponds to a table.

Create an entity

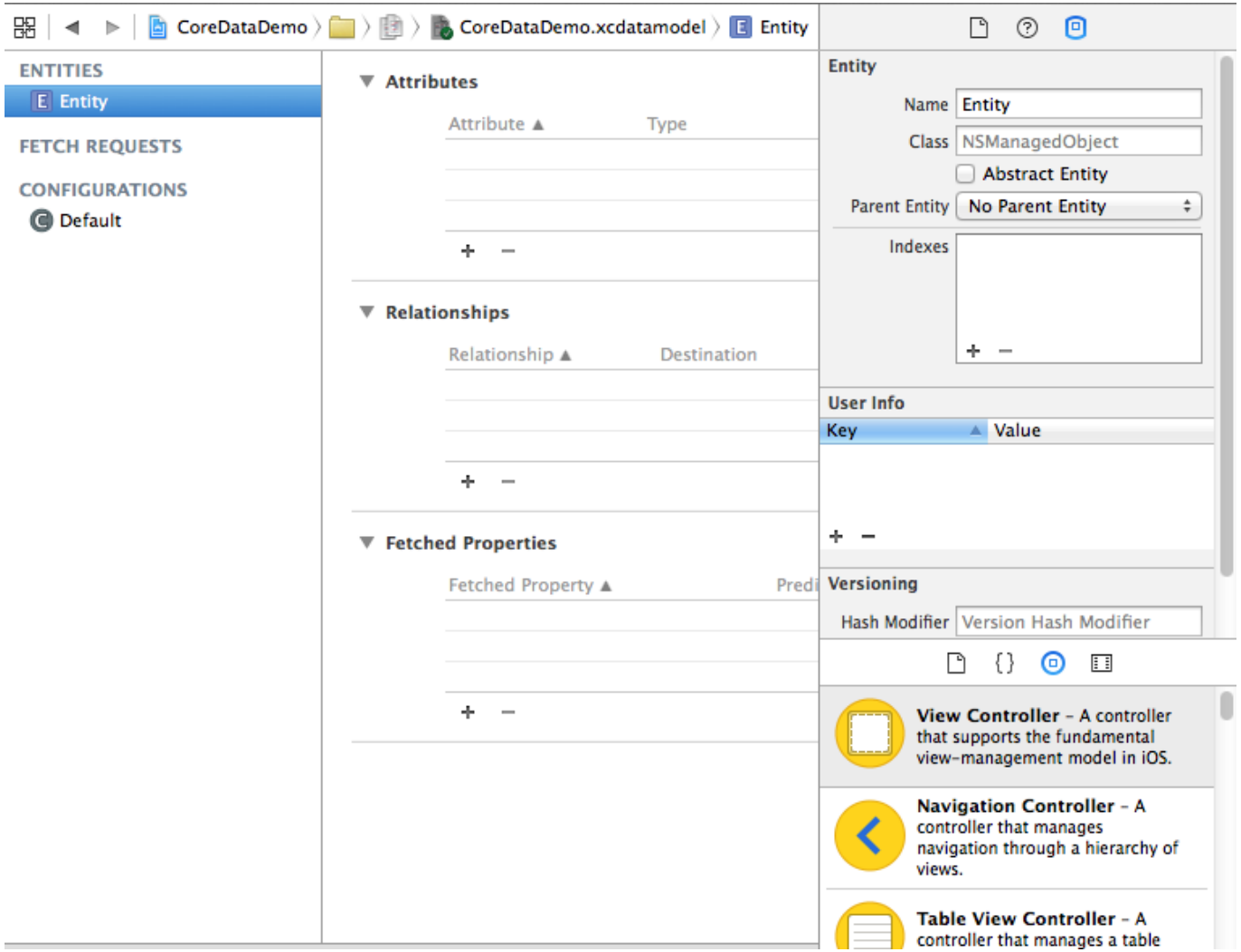
- Select the file in the navigator area to display the Core Data model editor



Create an entity Cont.

- Click Add Entity.
- A new untitled entity will appear in the Entities list in the navigator area.
- Select the new untitled entity.
- In the Entity pane of the Data Model inspector, enter the name of the entity, and press Return.

Create an entity Cont.



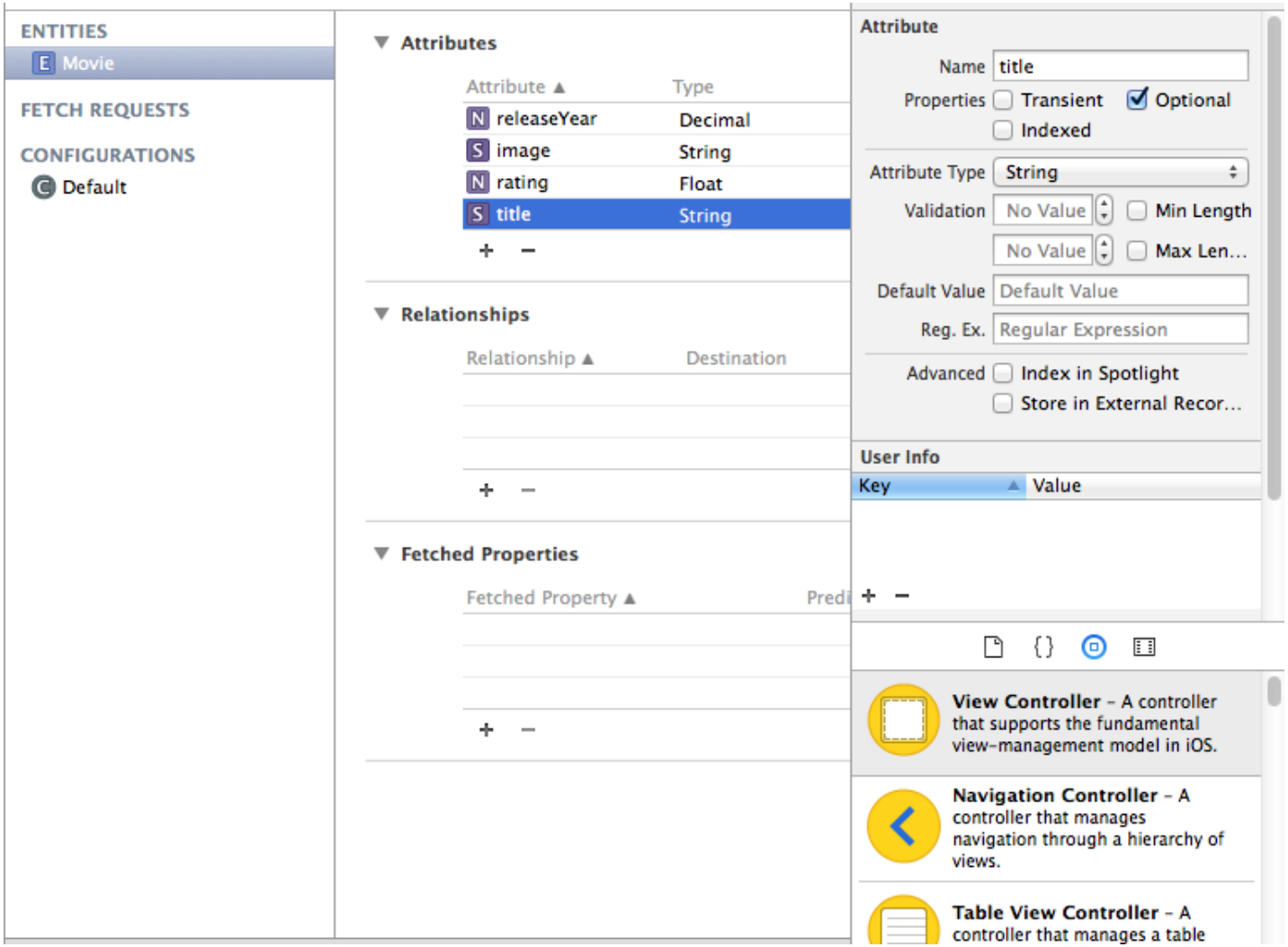
Attribute

- An ***attribute*** is a piece of information attached to a particular entity.
- For example, a ***Movie*** entity could have attributes for the movie's title, image, rating and releasedYear.
- In a database, an attribute corresponds to a particular field in a table.

Create attributes for the entity

- With the new entity selected, click the plus(+) sign at the bottom of the appropriate section.
- A new untitled attribute is added under the Attributes section of the editor area.
- Select the new untitled property.
- The property settings are displayed in the attributes pane of the Data Model inspector.
- Give the property a name and press Return.
- The attribute information appears in the editor area.

Create attributes for the entity Cont.



Saving to Core Data

- Import the Core Data module at the top of ***ViewController.swift***
- You may have had to link frameworks manually in your project's Build Phases if you've worked with Objective-C frameworks.
- In Swift, a simple import statement is all you need to start using Core Data APIs in your code.

```
import UIKit
import CoreData
```

Saving to Core Data Cont.

```
//1
let appDelegate = UIApplication.shared.delegate as!
    AppDelegate

//2
let managedContext = appDelegate.persistentContainer.
    viewContext

//3
let entity = NSEntityDescription.entity(forEntityName:
    "Movie", in: managedContext)

//4
let movie = NSManagedObject(entity: entity!,
    insertInto: managedContext)

movie.setValue("Movie", forKey: "title")
movie.setValue(2.3, forKey: "rating")
movie.setValue(2007, forKey: "releaseYear")

do{
    //5
    try managedContext.save()
}catch let error as NSError{

    print(error)

}
```

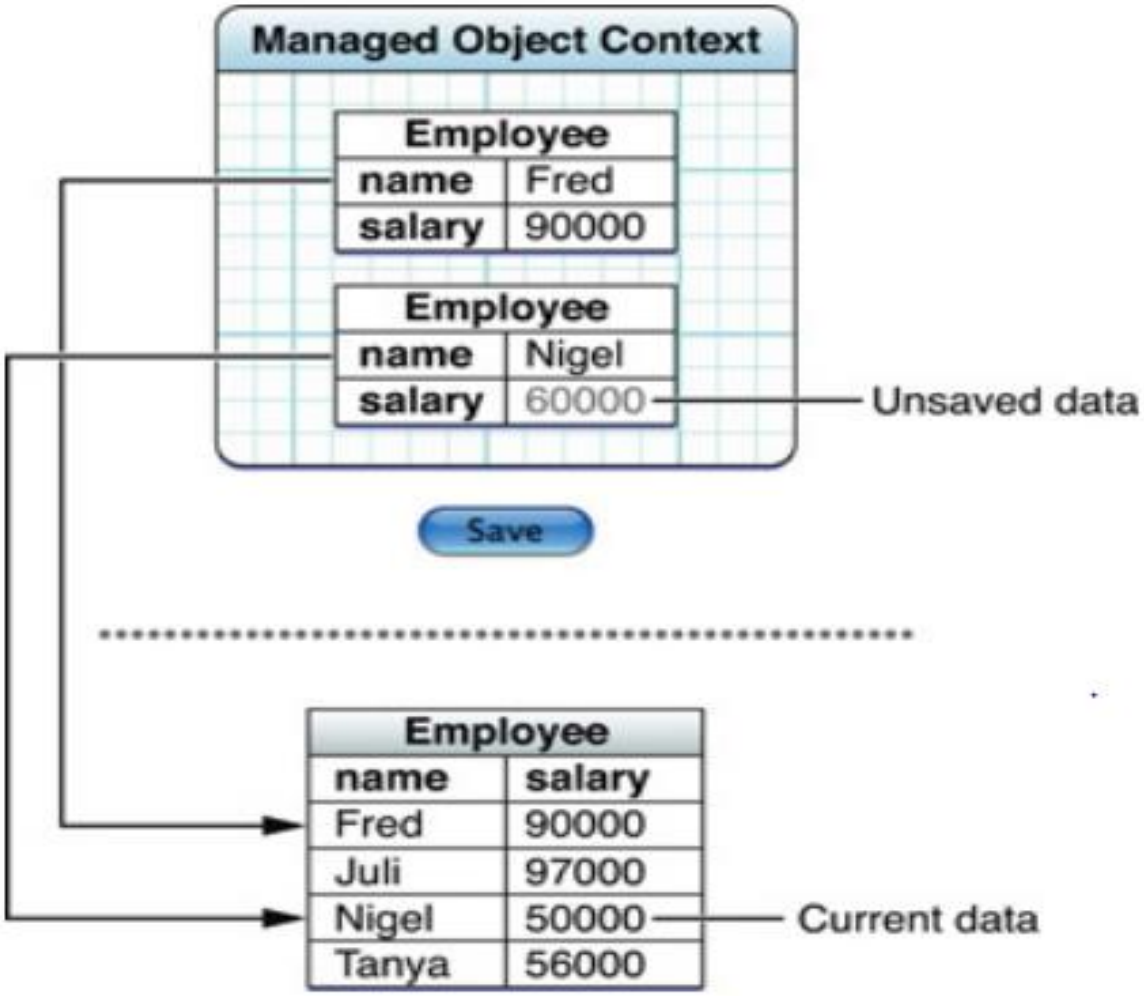
Managed Object Context

- A managed object context is an instance of NSManagedObjectContext.
- Its primary responsibility is to manage a collection of managed objects.
- These managed objects represent an internally consistent view of one or more persistent stores.

Managed Object Context Cont.

- The context is the central object in the Core Data stack.
- It's the object you use to create and fetch managed objects.
- A context is connected to a parent object store.
- the context asks its parent object store to return those objects that match the fetch request.
- Changes that you make to managed objects are not committed to the parent store until you save the context.

Managed Object Context Cont.



Fetching from Core Data

```
//1
let appDelegate = UIApplication.shared.delegate as!
    AppDelegate

//2
let managedContext = appDelegate.persistentContainer.
    viewContext


//3
let fetchRequest = NSFetchRequest<NSManagedObject>
    (entityName: "Movie")

//4
let myPredicate = NSPredicate(format: "title == %@",
    "Movie 1")
fetchRequest.predicate = myPredicate

do{

     //5
     movies = try managedContext.fetch(fetchRequest)
}catch let error as NSError{

    print(error)
}
```

Deletion from Core Data

```
//1
let appDelegate = UIApplication.shared.delegate as!
    AppDelegate

//2
let managedContext = appDelegate.persistentContainer.
    viewContext

//3
managedContext.delete((movies?[indexPath.row])!)

movies?.remove(at: indexPath.row)


do{
    //4
    try managedContext.save()
}catch let error as NSError{

    print(error)

}
```

Assignments

1.Movie List

- Update your Movie list app to add **+** button on the navigation bar which present view allow you to add new movie and when you press **Done** button, The movie will store in core data and the list view updated with the new movie.
- When run this application again the list display the stored movies.

2.Movie List

- Update your Movie list app to get movies from the following webservice:

  http://api.androidhive.info/json/movies.json
- Save all retrieved data using core data.
- If the application is online the data retrieved from the webservice and update the stored data.
- If the application is offline the data retrieved from the core data.
- Display the movie detail in another view controller when select it in the table view.

Lecture 6

Agenda

- Constraints.
  - ✓ Auto Layout.
  - ✓ Constraints Attributes.
  - ✓ Constraints Issues.

- Bridging
- Static Table view.

Constraints

Auto Layout

- Auto Layout dynamically calculates the size and position of all the views in your view hierarchy, based on constraints placed on those views.
- For example, you can constrain a button so that it is horizontally centered with an ImageView and so that the button's top edge always remains 8 points below the image's bottom. If the image view's size or position changes, the button's position automatically adjusts to match.

Auto Layout Cont.

Constraints

- Use Auto Layout to define relationship constraints for your app's user interface so that when one item changes its size or position, that item and its neighboring items adjust their sizes and positions appropriately.

Constraints Cont.

Constraints Cont.

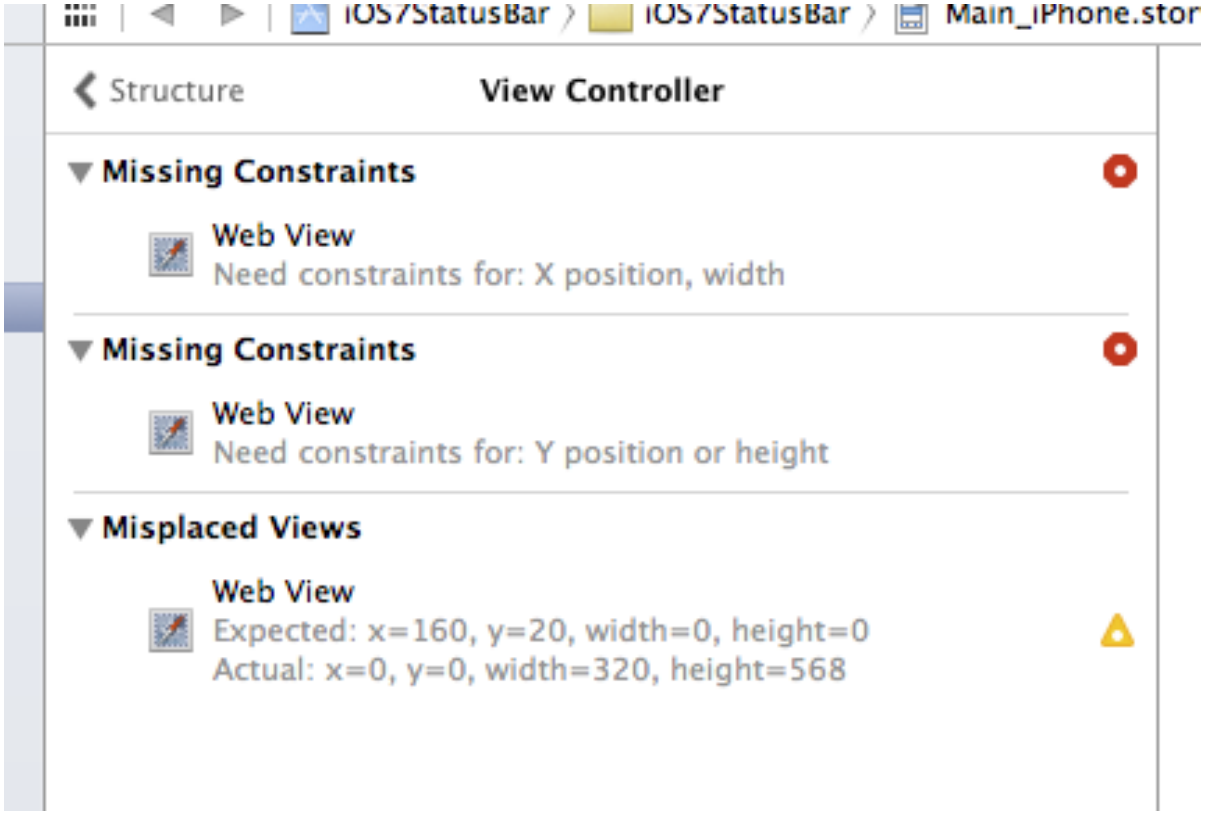
Constraints Attributes

- Every constraint for an item includes an attribute that relates to the size or position of the item's alignment rectangle.
  - ✓ **Horizontal positioning:** Leading, Trailing, Left, Right, and Center.
  - ✓ **Vertical positioning:** Top, Bottom, Center, and Baseline.
  - ✓ **Size:** Width, Height, and Aspect Ratio.

Constraints Issues

- Sometimes after you create the constraint, the Interface Builder outline view shows a disclosure arrow.
- The red arrow indicates that there are conflicts or ambiguities.
- Click the arrow, and you'll see a list of the issues.
- Typical issues include **missing constraints**, **conflicting constraints** and **misplaced views.**

Constraints Issues Cont.



Demo

Bridging

Demo

Static Table view

Demo

Assignments

1- registration Form

- Make a registration form that contains the following fields :
- (as label and text fields)
  - User Name
  - Password
  - Email
- Image View centered
- Submit Button

Notes:

✓ Make the view scrollable using static table view

2.Movie List

- Update your Movie list UI to fit on all devices using constrains specially the screen of adding new Movie.

Notes:

- ✓ Make the view scrollable using static table view.