Swift

448460
Fall 2015
09/21/2015
Kyoung Shin Park
Multimedia Engineering
Dankook University

Swift

- Constants, Variables, Type Inference
- String, Character
- Array<T>, Dictionary<K, V>, Range<T>
- Functions, Closures
- Optional, ImplicitlyUnwrappedOptional, Enum
- Data Structure (Class, Struct, Enum)
- Methods, Properties
- Subscript, Inheritance
- Initializer, Deinitializer
- AnyObject, Introspection, Type Casting (is, as, as?)
- Type conversion, Assertion

Basic Rules

- **Semicolons are optional.** You can just press Enter to finish a line of code.
- No more separate header and implementation files, just one file with a **.swift** extension.

Constants, Variables, Type Inference

- **Constants and variables must be declared before they are used.**
- You declare constants with the **let** keyword and variables with the **var** keyword.
- In Swift, constants are preferred over variables. You should only declare with var if it will be changed.

```
// constants
let languageName: String = “Swift”
// variables
var version: Double = 1.0
// Swift has type interference
let x = 1          // inferred as Int (same as let x : Int = 1)
var isUsed = true // inferred as Bool (same as var isUsed: Bool = true)
```

String

- A *String* is a collection of *Character* values
- **String.Index**
  - Each *String* value has an associated index type, **String.Index**, which corresponds to the position of each *Character* in the *String*
  - Since different characters can require different amounts of memory to store, Swift *String* cannot be indexed by integer values.
  - Use the **startIndex** property to access the position of the first *Character* of a *String*.
  - The **endIndex** property is the position after the last *Character* in a *String*.
  - **String.Index** value can access its immediately preceding / succeeding index by calling the **predecessor()** / **successor()** method
  - You advance forward with **advancedBy(_:)**

String

- Initialize/Create a string

```
var str1 = “” // empty string
var str2 = String() // empty string
var hello = “Hello”
var world = “World”
var hello2 = String(“Hello”)
var hhh = String(count: 3, repeatedValue: Character(“h”))
var smile = “ ” // Character is also declared with double quotes
var letters: [Character] = [“c”, “a”, “f”, “e”]
var cafe = String(letters)
print(cafe) // cafe
let b: Character = “B” // Character B
let acuteAccent: Character = “\u{0301}” // Character é
```

String

- String mutability

```
var h = hello + “ ” + world  // “Hello World”
h += “, Swift” // “Hello World, Swift”
h.append(Character(“!”)) // “Hello World, Swift!”
var str = h.stringByAppendingString(smile) // Hello World, Swift!
```

- Number of characters in a string (NSString length)

```
var len = hello.characters.count // “Hello” 5
var len2 = hello.startIndex.distanceTo(hello.endIndex) // “Hello” 5
```

- Characters in a string

```
for char in hello.characters { // character print
   print(char)
}
```

String

- String interpolation (stringWithFormat in Objective-C)

```
var name =  “Swift”
var total = 100
var string1 = “Hello \(name). Total is \(total).”
// Hello, Swift. The total number is 100.
```

- Lowercase/Uppercase/Capitalized String

```
print(string1.lowercaseString) // hello swift. Total is 100.
print(string1.uppercaseString) // HELLO SWIFT. TOTAL IS 100.
print(string1.capitalizedString) // Hello Swift. Total Is 100.
```

String

- Split string into an array
  - **To separate substrings by character**

```
var cities = "Seoul, New York, Tokyo LA"
var array1 = cities.componentsSeparatedByString(",")
// returns ["Seoul", " New York", " Tokyo LA"]
```

  - **To separate substrings with more than one character**

```
var array2 = cities.componentsSeparatedByCharactersInSet(
                    NSCharacterSet (charactersInString: ", "))
// returns ["Seoul", "", "New",  "York", "", "Tokyo",  "LA"]
```

String

- Access/Insert/Remove/Substring

```
var s = "Hello!"
s[s.startIndex] // "H"
s.insert("~",  atIndex: s.endIndex) // insert character "Hello!~"
let startIndex = s.startIndex.advancedBy(2) // index for "l"
let endIndex = s.startIndex.advancedBy(5) // index for "!"
// Insert/Remove
s.insert("a", atIndex: startIndex) // s = "Heallo!~"
s.insertContentsOf("bc".characters, at: startIndex.successor())  //
s="Heabcllo!~"
s.removeAtIndex(s.endIndex.predecessor()) // s="Heabcllo!"
// Substring
let substring = s[startIndex..<endIndex] // substring="abc"
var sub = s.substringWithRange(Range<String.Index>(start:
startIndex, end: endIndex)) // substring="abc"
```

String

- RangeOfString/ReplaceRange/RemoveRange
  - The **rangeOfString** method returns Range<String.Index>?

```
if let range = hello.rangeOfString("ello") { // "Hello" contains "ello"
    print("ello is exited in hello")
}
```

  - To get whole number part of a string representing a double

```
var num = "56.25"
if let range = num.rangeOfString(".") {
   let wholeNum = num[num.startIndex..<range.startIndex]
}
```

  - To replace substring

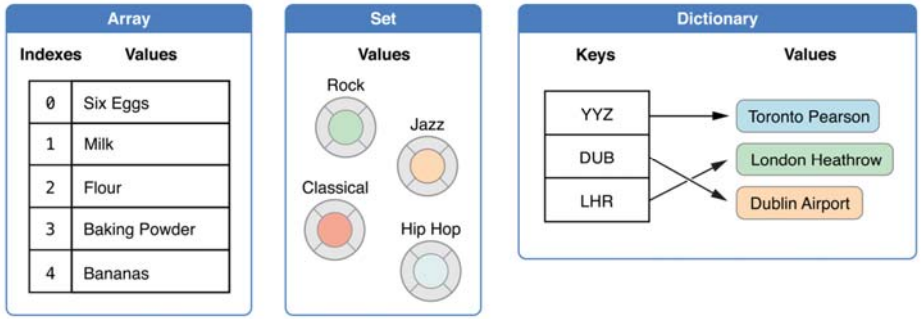
```
num.replaceRange(num.startIndex..<range.startIndex, with: "32") //
num=32.25
```

  - To remove the whole number part using **removeRange** method

```
num.removeRange(num.startIndex..<range.startIndex) // num=.25
```

Collection Types

- Arrays store ordered collections of values
- Sets are unordered collections of unique values
- Dictionaries are unordered collection of key-value associations



Array

- Array stores values of the same type in an ordered list
- Create/Initialize array

```
var a = Array<String>()        // empty array
var b = [String]() // empty array
var c = [Int] (count: 3, repeatedValue: 1) // [1, 1, 1]
let animals = ["Giraffe", "Cow", "Dog", "Cat"]
```

- Access/Combine/Append/Replace

```
//animals.append("Bird") // won't compile animals is immutable (because of let)
var animal = animals[1] // "Cow"
//var animal = animals[5] // crash (array index out of bound)
b.append("Egg"); // b=["Egg"]
b += ["Cocoa", "Milk", "Flour"] // b=["Egg","Cocoa","Milk","Flour"]
b[1...3] = ["Apple", "Butter"] // b=["Egg", "Apple", "Butter"]
```

Array

- Some Array<T> Methods

```
// enumerating an Array
for animal in animals {
    print(animal)
}
for (index, value) in animals.enumerate() {
    print("animals[\(index)]=\(value)")
}
c.insert(4, atIndex: 1) // c = [1, 4, 1, 1]
c.insertContentsOf([5, 6], at: 2) // c = [1, 4, 5, 6, 1, 1]
c.removeAtIndex(1) // c = [1, 5, 6, 1, 1]
c.removeRange(0..<2) // c = [6, 1, 1]
c.replaceRange(0...1, with: [8, 9, 7]) // c = [8, 9, 7, 1]
let sorted = c.sort {$0 < $1} // sorted = [1, 7, 8, 9]
```

Array

- More Array<T> Methods
  - **filter** creates a new array with any "undesirables" filtered out. The function passed as the argument returns false if an element is undesirable

```
filter(includeElement: (T) -> Bool) -> [T]
```

  - **map** creates a new array by transforming each element to something different. The thing it is transformed to can be of a different type than what is in the Array

```
map(transform: (T) -> U) -> [U]
let stringified: [String] = [1, 2, 3].map { "\($0)" }
```

  - **reduce** reduces an entire array to a single value

```
reduce(initial: U, combine: (U, T) -> U) -> U
let sum: Int = [1, 2, 3].reduce(0) { $0 + $1 } // adds up the numbers in the Array
```

Dictionary

- Dictionary stores associations between keys of the same type and values of the same type in a collection with no defined ordering.

```
var planets = Dictionary<String, Int>() // empty dictionary
var planets = [String:Int]() // empty dictionary
planets = ["Mercury":1, "Venus":2] // assign
planets["Mars"] = 3 // append
planets["Mars"] = 4 // replace
let earth = planets["Earth"] // earth is an Int? (would be nil)
// use a tuple with for-in to enumerate a Dictionary
for (key, value) in planets {
        print("\(key) = \(value)")
}
```

Range

- Range is just two end points of a sensible type
- Range is generic (e.g. Range<T>)
- This is sort of a pseudo-representation of Range

```
struct Range<T> {
    var startIndex : T
    var endIndex : T
}
```

- An Array's range would be a **Range<Int>** (since Arrays are indexed by Int)
- Waring: A String subrange is **not** Range<Int> (it is Range<String.Index>

Range

- There is special syntax for specifying a Range: either **... (inclusive)** or **..< (open-ended)**

```
let array = ["a", "b", "c", "d"]
let subArray1 = array[2...3] // subArray1 will be ["c", "d"]
let subArray2 = array[2..<3] // subArray1 will be ["c"]

// Range is enumerable, like Array, String, Dictionary
for i in 27...104 {
}
```

Range

```
let value = weight / (height * 0.01) / (height * 0.01)
var bmi : BMI
switch value {
    case 0.0 ..< 20.0:
        bmi = BMI.Underweight
    case 20.0 ..< 24.0:
        bmi = BMI.Normal
    case 24.0 ..< 30:
        bmi = BMI.Overweight
    default :
        bmi = BMI.Obesity
    }
```

Functions

- Functions are self-contained chunks of code that perform a specific task.
- Every function has a type, consisting of the function's parameter types and return type.

```
// function sayHello(_:)
func sayHello(personName: String = "World") -> String {
    let greeting = "Hello, " + personName + "!"
    return greeting
}
print(sayHello("Anna")) // "Hello, Anna!"
print(sayHello()) // "Hello, World!"
```

Closures

- Closures are **self-contained blocks of functionality**
- Similar to **blocks** in C and Objective-C and to ***lambdas*** in other programming languages
- Closures expression syntax

```
{ (<parameters>) -> <return-type> in
    <closure body statement>
}
```

Closures

```
let names = [“Chris”, “Alex”, “Dan”]
func backwards(s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversed = names.sort(backwards)
reversed = names.sort({ (s1: String, s2: String) -> Bool in
    return s1 > s2 // closures as parameters
})
reversed = names.sort( { s1, s2 in return s1 > s2 } ) // inferring type from context
reversed = names.sort( { s1, s2 in s1 > s2 } ) // implicit returns from single expression closures
reversed = names.sort( { $0 > $1 } ) // shorthand argument names
reversed = names.sort(>) // operator functions
reversed = names.sort() { $0 > $1 } // trailing closure
```

Closures

```
func repeat(count: Int, task: ()->()) {
    for i in 0..<count {
        task()
    }
}
repeat(2, {
    print(“Hello”) // closures as parameters – Hello Hello
})
repeat(2) {
    print(“Hello”) // trailing closures – Hello Hello
}
```

Optional

- An **Optional** is just an enum

```
enum Optional<T> : Reflectable, NilLiteralConvertible {
    case None
    case Some(T)
    // 중간생략..
}
var z: Int? // same as var z: Optional<Int> (and automatically set to nil)
z = 42
print(z!) // Use ! to unwrap an optional (run-time error if z is nil)
let x: String? = nil        // same as let x = Optional<String>.None
let x: String? = “Hello”// same as let x = Optional<String>.Some(“Hello”)
var y = x! // is same as switch x {
                //      case Some(let value): y = value
                //      case None: // raise an exception
                // }
```

Optional

- Use **Forced unwrapping** or **Optional binding** to unwrap an optional

```
let planets = [“Mercury”:1, “Venus”:2, “Earth”:3] // assign dictionary
let planetID: Int? = planets[“Earth”]
if planetID != nil { // check if planetID is nil to prevent run-time error
    print(“Earth = \(planetID!)”) // Forced unwrapping (using !) Earth = 3
} else {
    print(“Earth is not found”)
}
// Optional binding (unwrapping an optional)
if let planetID = planets[“Earth”] { // return true if planetID has valid value
    print(“Earth = \(planetID)”)      // Earth = 3
}
```

ImplicitlyUnwrappedOptional

- Use ImplicitlyUnwrappedOptional in all the same places in your code that you can use optionals.
- Should mark a variable as ImplicitlyUnwrappedOptional if you can guarantee that **it will NOT be nil at the time it’s called**.

```
enum ImplicitlyUnwrappedOptional<T>: Reflectable,
NilLiteralConvertible {
    case None
    case Some(T) // 중간생략..
}
var w: Int! = 1 // same as var w: ImplicitlyUnwrappedOptional<Int> = 1
print(w) // you don’t need to put !
var z: Int! // same as var z: ImplicitlyUnwrappedOptional<Int> = nil
print(z) // RUN-TIME ERROR (because z is nil)
w = nil // OK
print(w) // RUN-TIME ERROR (because w is nil) while unwrapping
```

Nil Coalescing Operator

- Nil coalescing operator (a **??** b) unwraps an optional a if it contains a value, or returns a default value b if a is nil.

```
a ?? b // same as a != nil ? a! : b

let defaultColorName = “Red”
var userDefinedColorName: String? // set to nil
// colorNameToUse = “Red” (because userDefinedColorName = nil)
var colorNameToUse = userDefinedColorName ?? defaultColorName

// if planets[“Earth”] = nil then it will return the default value 0)
var earthID = planets[“Earth”] ?? 0 // var earthID: Int? = 3
```

Optional Chaining

- Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil.

```
class Person {
   var contact: Contact? // automatically set to nil
}
class Contact {
   var address: String? // automatically set to nil
   var telephone: String? // automatically set to nil
}
let p = Person()
var phone = p.contact!.telephone! // run-time error(because contact=nil)
```

Optional Chaining

```
if let contact = p.contact { // optional binding to read optional property
   if let phone = contact.telephone {
      print(phone)
   } else {
      print("phone is nil")
   }
} else {
   print("contact is nil")
}
// optional chaining
var phone = p.contact?.telephone? // phone=nil (because contact=nil)
// use optional chaining & optional binding together
if let phone = p.contact?.telephone? {
//…
}
```

Enum

- Enum defines a common type for a group of related values

```
enum Gender {
    case Female
    case Male
}

var gender = Gender.Female   // (Enum Value)
gender = .Male               // (Enum Value)
switch gender {
    case .Female:
        print("FEMALE")
    case .Male:
        print("MALE")
}
```

Enum

- Multiple member values can appear on a single line, separated by commas

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

let somePlanet = Planet.Earth // Earth
switch somePlanet {
    case .Earth:
        print("Mostly harmless") // Mostly harmless
    default:
        print("Not a safe place for humans")
}
```

Enum

- Enumeration Associated Values

```
enum TrainStatus {
    case OnTime
    case Delayed(Int)
}

var status: TrainStatus = .Delayed(5)  // Delayed(5)
code = .OnTime                         // OnTime
switch status {
    case .OnTime:
        print("Train is on time")
    case .Delayed(let minutes):
        print("Train is delayed by \(minutes) minutes")
}
```

Enum

- Enum initialization

```
enum Gender {
    case Female, Male
    init() { // enum initialization
        self = .Female
    }
    init(_ name: String) {
        switch name {
        case "Female", "여자":  self = .Female
        case "Male", "남자": self = .Male
        }
    }
}
var g = Gender() // called init() Female
g = Gender("남자") // called init(name: ) Male
```

Enum

- Enum description

```
enum Gender {
    // 중간생략…
    var description: String {
        switch self {
        case .Female: return "FEMALE~"
        case .Male: return "MALE~"
        }
    }
}
var gender = Gender.Male
print(gender.description) // MALE~
gender = Gender("여자")
print(gender.description) // FEMALE~
```

Enum

- Enum **rawValue initializer** is a failable initializer.

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

let earthID = Planet.Earth.rawValue      // earthID is 3

let possiblePlanet = Planet(rawValue: 7) // possiblePlanet is Planet? Uranu
switch possiblePlanet! { // unwrapping optional
    case .Earth: print("Mostly harmless")
    default: print("Not a safe place for humans") // Not a safe…
}
```

Enum

- Enum **rawValue initializer** is a failable initializer.

```
if let aPlanet = Planet(rawValue: 9) { // optional binding
    switch aPlanet {
    case .Earth: print("Mostly harmless")
    default: print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position 9") // There isn't…
}
```

Data Structures in Swift

- Classes, Structures and Enumerations
  - There are 3 fundamental building blocks of data structures in Swift
- Similarities
  - Declaration syntax ..

```
class CalculatorBrain {
}

struct Vertex {
}

enum Op {
}
```

Data Structures in Swift

- Classes, Structures and Enumerations
  - There are 3 fundamental building blocks of data structures in Swift
- Similarities
  - Properties and Functions ..

```
func doit(argument: Type) -> ReturnValue { // method
}

var storedProperty = <initial value> (not enum) // stored property

var computedProperty: Type { // computed property (must define Type)
    get { … }
    set { … }
}
```

Data Structures in Swift

- Classes, Structures and Enumerations
  - There are 3 fundamental building blocks of data structures in Swift
- Similarities
  - Initializers ..

```
init (argument1: Type, argument2: Type, ..) {
}
```

Data Structures in Swift

- Classes, Structures and Enumerations
  - There are 3 fundamental building blocks of data structures in Swift
- Differences
  - **Inheritance (class only)**
  - **Introspection and casting (class only)**
  - Value type (struct, enum) vs Reference type (class)

Value vs Reference

- Value (**struct** and **enum**)
  - **Copied** when passed as an argument to a function
  - **Copied** when assigned to a difference variable
  - **Immutable** if assigned to a variable with **let**
  - Remember that function parameters are, by default, constants
  - You can put the keyword **var** on an parameter, and it will be mutable, but it's still a **copy**
  - You must note any **func** that can mutate a struct/enum with the keyword **mutating**
- Reference (**class**)
  - Stored in the heap and **reference counted** (automatically)
  - Constant **pointers** to a class (let) still can mutate by calling methods and changing properties
  - When passed as an argument, does not make a copy (just **passing a pointer** to same instance)

Methods

- **Override** methods/properties
  - Precede your **func** or **var** with the keyword **override**
  - **A method** can be marked **final** which will prevent subclasses from being able to override
  - **Classes** can also be marked **final**

Methods

- **Type** vs **Instance** methods/properties
  - For this example, lets consider the struct Double

```
var d: Double = …
if d.isSignMinus {
    d = Double.abs(d)
}
```

  - **isSignMinus** is an instance property of a Double (you send it to a particular Double)
  - **abs** is a type method of Double (you send it to the type itself)
  - You declare a type method or property with a **static** prefix (or **class** in a class)

```
static func abs(d: Double) -> Double
```

Methods

- Parameters Names
  - All parameters to all functions have an **internal** name and an **external** name
  - The internal name is the name of the local variable you use inside the method
  - The external name is what callers will use to call the method

```
func foo(external internal: Int) -> Int {
    let local = internal
    return local
}
func bar() {
    let result = foo(external: 123)
}
```

Methods

- Parameters Names
  - You can put _ if you don't want callers to use an external name at all for a given parameter
  - An _ is the default for the first parameter (only) in a method (but not for init methods)

```
func foo(_ internal: Int) -> Int {
    let local = internal
    return local
}
func bar() {
    let result = foo(123)
}
```

Methods

- Parameters Names
  - You can force the first parameter's external name to be the internal name with #

```
func foo(# internal: Int) -> Int {
    let local = internal
    return local
}
func bar() {
    let result = foo(internal: 123)
}
```

Methods

- Parameters Names
  - For other (not the first) parameters, the internal name is, by default, the external name

```
func foo(first: Int, second: Double) -> Double {
    let local1 = first
    let local2 = second
    return local1 + local2
}
func bar() {
    let result = foo(123, second: 1.5)
}
```

Methods

- Parameters Names
  - Any parameter's external name can be changed

```
func foo(first: Int, externalSecond second: Double) -> Double {
    let local1 = first
    let local2 = second
    return local1 + local2
}
func bar() {
    let result = foo(123, externalSecond: 1.5)
}
```

Methods

- Return Values

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currMin = array[0]; var currMax = array[1]
    for value in array[1..<array.count] {
        if value < currMin {
            currMin = value
        } else if value > currMax {
            currMax = value
        }
    }
    return (currMin, currMax) // returns a Tuple
}
let bounds = minMax([8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
```

Mutating Func

```
struct Frame {
    var x: Int, y: Int // stored property
    var width: Int, height: Int // stored property
    var area: Int { // computed property (don't need to enclose getter)
        return width * height
    }
    mutating func addWidth(width: Int) {// method
        self.width += width // mutating modifies the value of a stored property
    }
}
let f = Frame(x: 5, y: 10, width: 100, height: 100) // member-wise initializer
//f.width = 250 // invalid (can't change the struct properties stored in a let)

var g = Frame(x: 5, y: 10, width: 100, height: 100)
g.addWidth(15) // use mutating func to modify the value of a stored property
print(g.width) // 15

let h = Frame(x: 5, y: 10, width: 100, height: 100)
//h.addWidth(15) // compile error (can't call mutating func of struct stored in a let)
```

Properties

- Property Observers
  - You can observe changes to any property with **willSet** and **didSet**
  - One very common thing to do in an observer in a Controller is to update the user-interface

```
var someStoredProperty: Int = 42 {
    willSet { print("About to set \(newValue)") }
    didSet {
        if someStoredProperty > oldValue {
            print("Changed \(someStoredProperty – oldValue)")
        }
    }
}
override var inheritedProperty {
    willSet { newValue … }
    didSet { oldValue … }
}
```

Properties

- Lazy Initialization
  - A **lazy** property does not get initialized until someone **accesses** it
  - You can allocate an object, execute a closure, or call a method if you want

```
lazy var brain = CalculatorBrain() // nice if CalculatorBrain used lots of resources
lazy var someProperty: Type = {
    // construct the value of someProperty here
    return <constructed value>
} ( )
lazy var myProperty = self.initializeMyProperty( )
```

Subscript

- Classes, structures, enumerations can define **subscripts**, to set and get values by **index**

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        get {
            return multiplier * index
        } // you can also set {}
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
for I in 1…9 {
    “3 x \(i) = \(threeTimesTable[i])”
}
```

Subscript

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

Subscript

```
var matrix = Matrix(rows: 2, columns: 2)

matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
let someValue = matrix[2, 2]

for i in 0...1 {
    for j in 0...1 {
        print("[\(i), \(j)] = \(matrix[i, j])")
    }
}
```

```
grid = [0.0, 0.0, 0.0, 0.0]
```

```
        column
          0     1
row 0 [ 0.0,  0.0,
    1   0.0,  0.0 ]
```

Inheritance

```
class Vehicle {
    var wheels // stored property (inferred as Int; initialized as 0)
    var description: String { // read-only computed property (no getter code)
        return “\(wheels) numberOfWheels”
    }
}
class Car: Vehicle {
    var speed = 10.0
    override init() { // override (designated) init
        super.init()
        wheels = 4 // call the superclass init first before the inherited property
    }
    override var description: String { // override computed property
        return “\(wheels) numberOfWheels with \(speed) speeds”
    }
}
let aVehicle = Vehicle() // called init() aVehicle is an instance of Vehicle
aVehicle.wheels = 6 // valid
aVehicle.description // “6 numberOfWheels”
let aBike = Bicycle() // called init()
aBike.description // “4 numberOfWheels with 10.0 speeds”
```

Initialization

- When is an **init** method needed?
  - **init** methods are not so common because properties can have their defaults set using **=**
  - Or properties might be Optionals, in which case they start out **nil**
  - You can also initialize a property be executing a closure
  - Or use **lazy** instantiation
  - So you only need init when a value can't be set in any of these ways
- You also get some "free" init methods
  - If all properties in a base class have defaults, you get init() for free

```
struct MyStruct {
    var age: Int = 42
    var name: String = "Smith"
    init(age: Int, name: String) // comes for free
}
```

Initialization

- What can you do inside an **init**?
  - You can set any property's value, even those with default values
  - Constant properties (i.e., properties declared with let) can be set
  - You can call other init methods in your own class using **self.init(<args>)**
  - In a class, you can of course also call **super.init(<args>)**
- What are you required to do inside init?
  - By the time any init is done, all properties must have values (optionals can have the value nil)
  - There are two types of inits in a class, **convenience** and **designated**
  - A **designated** init must (an can only) call a designated init that is in its immediate superclass

Initialization

- What are you required to do inside init?
  - You must initialize all properties introduced by your class before calling a superclass init
  - You must call a superclass init before you assign a value to an inherited property
  - A **convenience** init must (and can only) call a designated init in its own class
  - A **convenience** init may call a designated init indirectly (through another convenience init)
  - A **convenience** init must call a designated init before it can set any property values
  - The calling of other inits must be complete before you can access properties or invoke methods

Initialization

- Inheriting **init**
  - If you do not implement any designated inits, you'll inherit all of your superclass designateds
  - If you override all of your superclass designated inits, you'll inherit all of its convenience inits
  - If you implement no inits, you'll inherit all of your superclass inits
  - Any init inherited by these rules qualifies to satisfy any of the rules on the previous slide
- Required **init**
  - A class can mark one or more of its init methods as **required**
  - Any subclass must implement said init methods (though they can be inherited per above rules)

Initialization

- Failable init
  - If an init is declared with a **?** (or **!**) after the word init, it returns an Optional

```
init?(arg1: Type1, …) {
    // might return nil in here
}
```

  - These are rare.
  - Note: The documentation does not seem to show these inits! But you'll be able to tell because the compiler will warn you about the type when you access it.

```
if let image = UIImage(named: "foo") { // image is Optional UIImage
    // image was successfully created
} else {
    // couldn't create the image
}
```

Initialization

- Creating Objects
  - Usually you create an object by calling it's initializer via the type name

```
let x = CalculatorBrain()
let y = ComplicatedObject(arg1: 42, arg2: "age", ..)
let z = [String]()
```

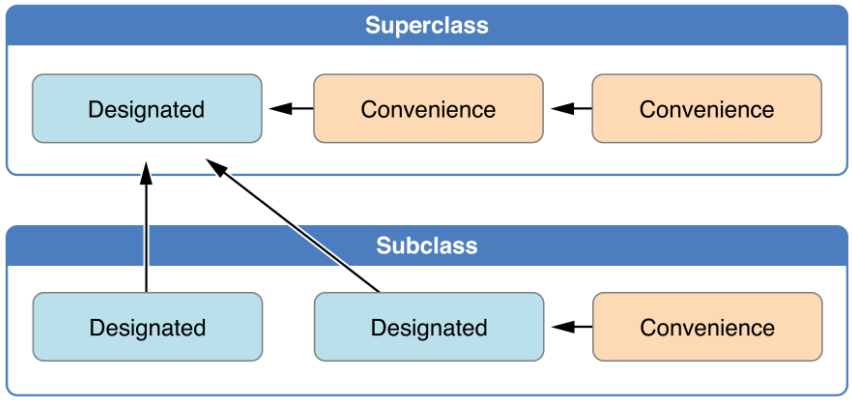
  - But sometimes you create objects by calling type methods in classes

```
let button = UIButton.buttonWithType(UIButtonType.System)
```

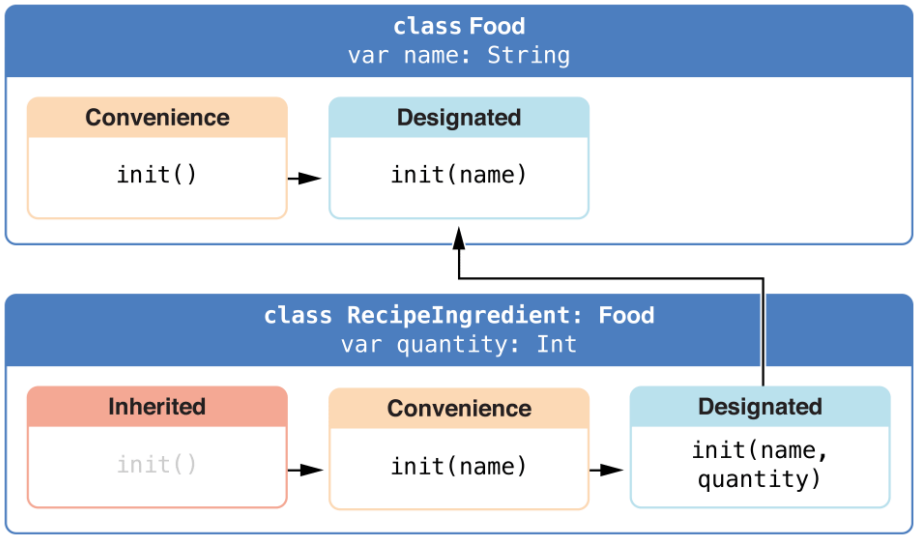
  - Or obviously sometimes other objects will create objects for you

```
let commaSeparatedArrayElements: String = ",".join(myArray)
```

Initialization



Initialization



Initialization

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: “Unnamed”)
    }
}
class RecipeIngredient: Food {
    var quality: Int
    init(name: String, quantity: Int) { // designated init
        self.quantity = quantity // set new property before superclass init
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
let oneBacon = RecipeIngredient(name: “Bacon”)
let sixEggs = RecipeIngredient(name: “Eggs”, quantity: 6)
```

Deinitialization

- Deinitializer is called immediately before a class instance is deallocated. You write deinitializers with the **deinit** keyword.

```
class FileHandler {
    let fileDescriptor: FileDescriptor
    init(path: String) {
        fileDescriptor = openFile(path)
    }
    deinit {
        closeFile(dileDescriptor)
    }
}
```

Type Casting

- Type casting is a way to check the type of an instance. Also type casting to check whether a type conforms to a protocol.
- Type checking operator **‘is’** checks whether an instance is of a certain subclass type (i.e. the downcast is possible).
- Type cast operator **‘as’** casts a value to a different type.
- **‘as?’** returns an optional value and the value will be nil if the downcast is not possible.
- **‘as!’** returns the forced unwrapped optional value (i.e., run-time error if you try to downcast to an incorrect class type).
- Type casting for **AnyObject** and **Any**
  - **AnyObject** can represent an instance of any class type.
  - **Any** can represent an instance of any type at all, including function types and non-class types.

AnyObject

- Special “Type” (actually it’s a Protocol)
  - Used primarily for compatibility with existing Objective-C based APIs
- Where will you see it?
  - As properties (either singularly or as an array of them), e.g.

```
var destinationViewController: AnyObject
var toolbarItems: [AnyObject]
```

  - Or as arguments to functions

```
func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject)
func addConstraints(constraints: [AnyObject])
func appendDigit(sender: AnyObject)
```

  - Or even as return types from functions

```
class func buttonWithType(buttonType: UIButtonType) -> AnyObject
```

AnyObject

- How do we use AnyObject?
  - We don't usually use it directly
  - Instead, we convert it to another known type
- How do we convert it?
  - We need to create a new variable which is of a known object type (i.e., not AnyObject)
  - Then we assign this new variable to hold the thing that is AnyObject
  - Of course, that new variable has to be of a compatible type
  - If we try to force the AnyObject into something incompatible, crash!
  - But there are ways to check compatibility (either before forcing or while forcing)

AnyObject

- Casting AnyObject (**is, as, as?**)
  - We "force" an AnyObject to be something else by "**casting**" it using the **as** keyword

```
// this would crash if destinationViewController is not,
// in fact, a CalculatorViewController (or subclass thereof)
var destinationViewController: AnyObject
let calcVC = destinationViewController as CalculatorViewController
```

  - To protect against a crash, we can use **if let** with **as?**

```
if let calcVC = destinationViewController as?
CalculatorViewController { … }
```

  - Or we can check before we even try to do as with the **is** keyword

```
if destinationViewController is CalculatorViewController { … }
```

AnyObject

- Casting Arrays of AnyObject
  - If you're dealing with an **[AnyObject]**, you can cast the elements or the entire array

```
var toolbarItems: [AnyObject]
for item in toolbarItems {
    if let toolbarItem = item as? UIBarButtonItem {
        // do something with the toolbarItem
    }
}
```

  - Or..

```
for toolbarItem in toolbarItems as [UIBarButtonItem] {
        // do something with the toolbarItem
    }
}
```

AnyObject

- Another example
  - Remember when we wired up our Actions in our storyboard?
  - The default in the dialog that popped up was AnyObject
  - We changed it to UIButton
  - But what if we hadn't changed it to UIButton?
  - How would we have implemented appendDigit?

```
@IBAction func appendDigit(sender: AnyObject) {
    if let sendingButton = sender as? UIButton {
        let digit = sendingButton.currentTitle!
        // do something else…
    }
}
```

AnyObject

- Yet another example
  - It is possible to create a button in code using a UIButton type method

```
let button: AnyObject =
    UIButton.buttonWithType(UIButtonType.System)
// The type of this button is AnyObject
// To use it, we'd have to cast button to UIButton
```

  - We can do this on the fly if we want

```
let title = (button as UIButton).currentTitle
// Again, this would crash if button was not, in fact, a UIButton
```

Type Conversion

- Conversion between types with init()
  - A sort of "hidden" way to convert between types is to create a new object by converting

```
let d: Double = 37.5
let f: Float = 37.5
let x = Int(d)          // truncates
let xd = Double(x)
let cgf = CGFloat(d)
let a = Array("abc") // a = ["a", b", "c"], i.e, array of character
let s = String(["A", "b", "c"]) // s = "abc" (the array is of Character, not String)
let s = String(52) // no floats
```

- Int, Double, Float -> String

```
label1.text = "\(slider1.value)"
let s = "\(37.5)"
```

Type Conversion

- String -> Int, Double, Float
  - Convert String to NSString and use convenience methods

```
var str = "37.5"
var iValue : Int = NSString(str).integerValue // 37
var fValue : Float = NSString(str).floatValue // 37.5
var dValue : Double = NSString(str).doubleValue // 37.5
```

  - Use Int(string: String), Float(string: String), Double(string: String)

```
if let value = Int(str) { iValue = value } else { print("error") }
```

  - Extending String, and then use it

```
extension String {
    func toDouble() -> Double {
        if let unwrapNum = Double(self) { return unwrapNum }
        else { print("Error converting ₩"" + self + "₩" to Double")
                return 0.0     } } }
var num = str.toDouble()
```

Assertions

- Debugging Aid
  - Intentionally crash your program if some condition is not true (and give a message)

```
assert(() -> Bool, "message")
```

  - The function argument is an "autoclosure" however, so you don't need the {}

```
assert(validation() != nil, "the validation function returned nil")
// will crash if validation() returns nil (because we are asserting that validation() does not)
// the validation() != nil part could be any code you want
```

  - When building for release (to the AppStore), asserts are ignored completely

References

- https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html
- https://developer.apple.com/swift/blog/?id=30
- http://mattquiros.com/blog/2014/07/30/a-quick-guide-to-swift-introduction/
- Lecture 4 Slide from Developing iOS8 Apps with Swift (Winter 2015) @Stanford University