Contents

Chapter 1

Basic syntax

1.1 Variables

Variables are declared as

```
[strong/ weak] var [variable-name] : [variable-type] [? / !]
```

Optional variables marked with ? are `optional`.

Variables have to be initialized upon declaration. Optional variables are automatically initialized as `nil`.

Declare optional type as ! rather than ? means automatic unwrap (so don't have to add ! every time we want to unwrap).

Computed property:

```
var displayValue : Double {
    get {
        return NSNumberFormatter().numberFromString(display.text!)!.doubleValue;
    }
    set {
        // newValue is by default the new value that displayValue is set to
        display.text = "\(newValue)" // convert stuff into String
    }
}
```

1.2 Functions

Functions are declared as

```
func [function[name] ([param 1 name] : [param 1 type], ... )
```

Pass function type as parameter type:

```
func performOperation(operation: (Double, Double) -> Double) {
   var result = operation(..., ...)
}
// FIRST WAY TO CALL performOperation
func multiply(op1: Double, op2: Double) -> Double {
   return op1 * op2
}
performOperation(multiply)

// SECOND WAY TO CALL performOperation
performOperation({ (op1: Double, op2: Double) -> Double in
   return op1 * op2
})

// THIRD WAY TO CALL performOperation (use type inference)
performOperation({ (op1, op2) in
    op1 * op2 // return keyword is optional
})

// 4TH WAY TO CALL performOperation
performOperation({
    \$0 * \$1 // does not need to specify parameter name
})

// MOST CONCISE WAY
performOperation { \$0 * \$1} // function as last argument can be declared outside of (
   ). if no other argument, () can also be omitted
```

Constructors are called `init()`.

1.3 Basic data structures

1.3.1 Array

```
var operandStack :  Array<Double> = Array<Double>()
```

Note that type can be omitted because Swift has type inference.

Alternative and preferred: `[Double]()`.

Array is a `struct`.

1.3.2 Dictionary

```
var knownOps = Dictionary<String, Op>()
```

Alternative and preferred: `[String :  Op]()`. Dictionary is a `struct`. Enumerate Dictionary using tuples:

```
pacl10teamRankings = ["Stanford" : 1, "Cal": 10]
let ranking = pacl10teamRankings["Ohio State"]
for (key,value) in pack10teamRankings {
   println("\(key) = \(value)")
}
```

1.3.3 Range

Two end points of a sensible type.

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

Specifying a Range:

```
let array = ["a", "b", "c", "d"]
let subArray1 = array[2..3] // 2 to 3 inclusive
let subArray2 = array[2..<3] // 2 to 3 not including 3
for i in [27..104] {
}
```

1.4 Enum

Can have functions. Only computed properties.

```
enum Op{
   case Operand(Double)
   case UnaryOperation(String, Double -> Double)
   case BinaryOperation(String, (Double, Double) -> Double)
}

var opStack = [Op]()
var knownOps = [String, Op]()

func pushOperand(operand: Double){
   opStack.append(Op.Operand(operand))
}

func performOperation(symbol: String){
   // let operation = ..., if operation != nil ...
   if let operation = knownOps[symbol] {
      opStack.append(operation)
   }
}

func evaluate() -> Double? {

}
```

Chapter 2

Other language-specific notes

2.1 Private vs Public

- If no keyword `private`, variables are public inside your program. Only use `public` when you ship a framework.

2.2 Value vs Reference

- `struct` does not have inheritance and, like `enum`, is passed by value, not reference. There's an implicit `let` in front of parameter declaration, so parameters passed by value are immutable.(can specify `var`).
- 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 function that can mutate a struct/enum with the keyword `mutating`.
- Constant pointers to a class (`let`) still can mutate by calling methods and changing properties.

2.3 Methods

- All parameters to all functions have **internal** name and an **external** name. The internal name is 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){
   let local = internal
}
func bar(){
  let result = foo(external: 123)
}
```

- You a use bar to specify ”dont’ care”.

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

- The bar is default for the first parameter (only) in a method, but not for `init` methods. You can force the first parameter’s external name to be the internal name with `#`. For other (not the first) parameters, the internal name is by default the external name.

2.4 Properties

- You can observe any changes to any property with `willSet` and `didSet`. One very commong thing to do in an observer in a Controller is to update the user-interface.

```
  var someStoredProperty : Int = 42 {
    willSet {
      //newValue is the new value
    }
    didSet {
      // oldValue is the old value
    }
  }
```

- A lazy property does not get initialized

2.5 Initializer

- You can set any property’s value, even those with default values. Constant properties declared with `let` can also be set.

- By the time any `init` is done, all properties must have values (optionals can have the value `nil`).

- You must initialize all properties introduced by your class before calling a superclass's `init`.
- You must call a superclass's `init` before you assign a value to an inherited property.
- A designated init can only call a desginated init in its immediate subclass.
- A class can mark one or more of its init methods as `required`. Any subclass must implement such init methods.

2.5.1 Inheriting init

- If you do not implement any designated inits, you will inherit all of your superclass's designateds.
- If you override all of your superclass's designated inits, you will inherit all its convenience inits.

2.5.2 Creating objects

Create an object by calling it's initializer via the type name or by calling type methods in classes

```
let x = CalculatorBrain()
let z = [String]()
let button = UIButton.buttonWithType(UIButtonType.System)
```

Sometimes other objects will create objects for you.

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

2.5.3 Conversion between types with `init()`

```
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")
let s = String(["a", "b", "c"])
let s = "\(37.5)"
```

2.6 Casting

Casting is done using the keyword as.

```
// crash if destinationViewController is not a CalculatorViewController
let calcVC = destinationViewController as CalculatorViewController
```

To protect against a craash, use if let with as?, which returns an optional.

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

Or we can check the type.

```
if destinationViewController is CalculatorViewController { ... }
```

Can also cast a whole array.

```
var toolbarItems: [AnyObject]
// first way
for item in toolbarItems {
   if let toolbarItems = item as? UIBarButtonItem {
      // do something
   }
}
// second way. note no ? here
for toolbarItem in toolbarItems as [UIBarButtonItem] {
   // do something
}
```

2.7 Useful functions

- Array: say var arr = [a,b,c].

```
+= [T] // add another array to the end
first -> T? // optional
last -> T? // optional
append(T)
insert(T, atIndex: Int)       // arr.insert(d, atIndex: 1) yields arr = [a,d,b,c]
splice(Array<T>, atIndex: Int)   // arr.splice([d,e], atIndex: 1) yields arr =
    [a,d,e,b,c]
removeAtIndex(Int)            // arr.removeAtIndex(1) yields arr = [a,c]
```

```
    removeRange(Range)            // arr.removeRange(0..<2) yields arr = [c]
    replaceRange(Rance, [T])      // arr.replaceRange(0..1, with: [x,y,z]) yields arr =
        [x,y,z,b]
    sort(isOrderedBefore:(T,T) -> Bool) // arr.sort({$0 < $1})
    filter(includeElement: (T) -> Bool) -> [T] // if includeElement returns false, T
        is removed
    map(transform:(T) -> U) -> [U]   // let stringified: [String] = [1,2,3].map{"\($0)"}
    reduce(initial: U, combine: (U,T) -> U) -> U // let sum: Int = [1,2,3].reduce(0)
        {$0 + $1}
```

2.8 String

Substrings and indexes are all about the global `advance` function.

```
  var s = "hello"
  let index = advance(s.startIndex, 2) // index is a String.Index to the 3rd glyph, "l"
  s.splice("abc", index)           // s = "heabcllo"

  let startIndex = advance(s.startIndex, 1)
  let endIndex = advance(s.startIndex, 6)
  let substring = s[index..<endIndex]  // substring will be "eabcl"
```

Use `Range` to get substring.

```
let num = "56.25"
  if let decimalRange = num.rangeOfString(".") { // decimalRange is Range<String.Index>
    let wholeNumberPart = num[num.startIndex..<decimalRange.startIndex]
  }
```

2.9 Assertions

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

```
assert(() -> Bool, "message")
assert(validation() != nil, "the validation functions returned nil")
```

2.10 NSUserDefaults

A Property List is a collection of objects which are ONLY one of

`NSString, NSArray, NSDictionary, NSNumber, NSData, NSDate`

`NSUserDefaults` is a tiny database that stores Property List data, which persists between launchings of your application. Great for things like ”settings” and such.

```
setObject(AnyObject, forKey: String)  // AnyObject must be a Property List
objectForKey(String) -> AnyObject?
arrayForKey(String) -> Array<AnyObject>? // returns nil if value is not set or not an
    array

setDouble(Double, forKey: String)
doubleForKey(String) -> Double        // not an optional, return 0 if no such key
```

Example code:

```
let defaults = NSUserDefaults.standardUserDefaults()
let plist : AnyObject = defaults.objectForKey(String)

// Be sure changes are saved at any time by synchronizing
if !defaults.synchronize() {
  // failed, not much you can do about it
}
```

2.11 NSAttributedString

```
func setAttributes(attributes: Dictionary, range: NSRange)
func addAttributes(attributes: Dictionary, range: NSRange)
```

This is a pre-Swift API. `NSRange` is not a `Range`.
Attributes examples:

```
NSForeGroundColorAttributeName : UIColor
NSStrokeWidthAttributeName : CGFloat
NSFontAttributeName : UIFont
```

To get the right font:

```
preferredFrontForTextStyle(UIFontTextStyle) -> UIFont

// some of the styles

UIFontTextStyle.Headline

UIFontTextStyle.Body

UIFontTextStyle.Footnote
```

Chapter 3

View

The top of the usable view hierarchy is the Controller's `var view: UIView`. This view is the one whose bound will change on rotation, and likely the one you will programmatically add subviews to. Instead of initializing, you can call `awakeFromNib()` (only for views that come out of a storyboard).

3.1 Coordinate system

To work with view's coordinate system, use `CGFloat`. Double and Float can be converted to CGFloat by `let cfg = CGFloat(aDouble)`.

`CGRect` is a struct with a `CGPoint` and a `CGSize` in it.

```
struct CGRect {
   var origin: CGPoint
   var size: CGSize
}
// convenient properties and functions
var minX: CGFloat       // left edge
var midY: CGFloat       // midpoint vertically
intersects(CGRect) -> Bool    // intersection
intersect(CGRect)       // create an intersection of 2 CGRects
contains(CGPoint) -> Bool     // contain given point?
```

`var bounds: CGRect` is the rectangle containing the drawing space in its own coordinate system.

`var center: CGPoint` in the center of a `UIView` in its superview's coordinate system.

`var frame: CGRect` is the rect containing a `UIView` in its superview's coordinate system. `frame` is used for positioning, not drawing inside a view's coordinate system.

Views can be rotated, so `frame.size` is not equal to `bounds.size`.

By default, when a `UIView`'s bound changes, there is no redraw. Instead the bits of the existing images are scaled to the new bounds size. This is often not what you want, but there is a `UIVew` property to control this:

```
var contentMode:  UIViewContentMode
```

Or you can just call `.reDraw`.

3.2 Creating views via code

```
let labelRect = CGRect(x: 20, y: 20, width: 100, height: 50)
let label = UILabel(frame: labelRect)
label.text = "hello"
view.addSubview(label)
```

3.3 Custom views

To draw a custom view, just create a `UIView` subclass and override `drawRect`:

```
override func drawRect(regionThatNeedsToBeDrawn: CGRect)
```

NEVER call `drawRect`. If your view needs to be redrawn, let the system know by calling

```
setNeedsDisplay()
setNeedsDisplay(regionThatNeedsToBeDrawn: CGRect)
```

3.4 Drawing images

```
let image: UIImage = ...
image.drawAtPoint(aCGPoint)      // the upper left corner of the image put at CGPoint
image.drawInRect(aCGRect)      // scales the image to fit CGRect
image.drawAsPatternInRect(aCGRect) // tiles the image into a CGRect
```

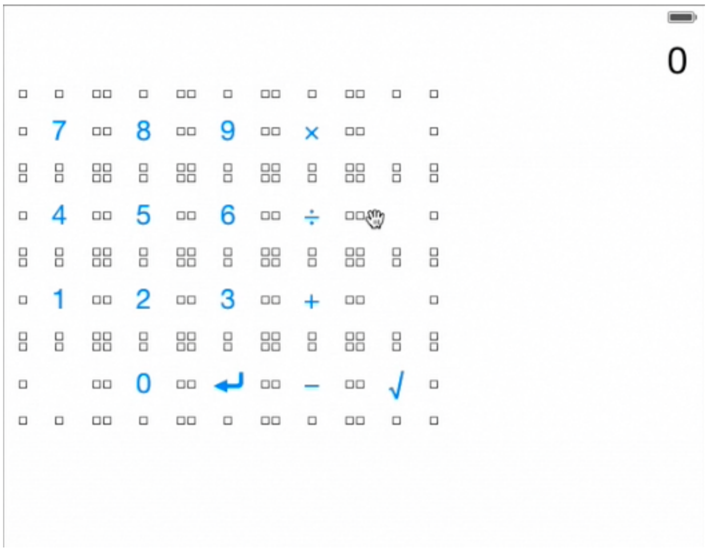
Chapter 4

Autolayout

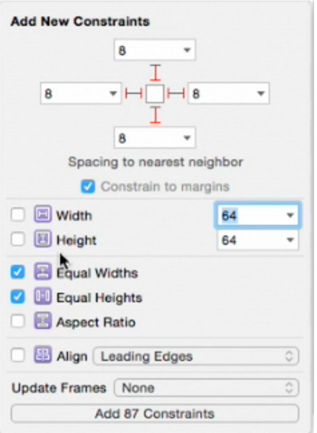
Example 1: Arrange grids so that:

- All buttons have same size.
- Same spacing between neighbors and boundaries.

1st step:



2nd step:



Add New Constraints

8

8 ⊢□⊣ 8

8

Spacing to nearest neighbor

☑ Constrain to margins

□ Width 64

□ Height 64

☑ Equal Widths

☑ Equal Heights

□ Aspect Ratio

□ Align Leading Edges

Update Frames None

Add 87 Constraints