Types

Int	`1, 25, 589, 300_000`
Float	`1.6, 6.89, 4.6789`
Double	`3.1415925359`
Bool	`true, false`
String	`“This is a String”`

Variables

Constants cannot be redefined

```
let someConstant : Int = 18
```

Variables can be redefined later (mutable)

```
var someVariable : Int = 10
    someVariable = 46 //Valid!
```

Swift can infer primitive variable types

```
let someType = "Swift knows I'm a String"
```

Optionals

Optionals are containers that can be nil

```
var optionalDef : Int? = 10
var optionalNil : Int? = nil
```

Force-unwrapping automatically unwraps without checking for nil, and is usually not a good idea

```
let forceDef = optionalDef! //unwraps as Int
let forceNil = optionalNil! //runtime error
```

The Nil Coalescing Operator allows you to define a fallback value

```
let number = optionalInt ?? 0
```

Optional chaining is usually the best option and lets you handle `nil`

```
if let val = optionalValue {
    print("val contains \(number)!") }
else {
    print("val doesn't contain anything")
}
```

Strings

Concatenation uses the + operator

```
let concat = string1 + string2
```

Convert Strings to Int using `.toInt`

```
let numString = “2”
let intVersion = numString.toInt
```

Arrays

```
var myArray = ["soda",
"evans", "cory"]                Define array of Strings
myArray[1] = "wheeler"          Updates “evans” to “wheeler”
myArray.count                   Returns number of elems (3)
myArray.append("dwinelle")      Appends “dwinelle” to end
myArray += ["bechtel"]          Appends “bechtel” to end
```

Array initialization (both are the same)

```
var empty1 = [String]()
var empty2: [String] = []
```

Array slicing

```
var nums = [0, 1, 2, 3]
nums[1...3]                     Returns [1,2,3] (... means inclusive)
nums[1..<3]                     Returns [1,2] (..< means exclusive)
nums[0..<                       Returns all but last value
     (nums.endIndex-1)]
```

Functions

Function definitions require specifying input and output types

```
func addTwo(a:Int, b:Int) -> Int {
    return a + b }
```

Call it with `addTwo(a:2, b:4)` `//Returns 6`

External Parameters can be used to differentiate between functions with the same name

```
func update(withNewData data: [String]) -> Bool {
    originalData += data
    return true }
```

Call it with `update(withNewData: [4, 5])`

Control Structures (If, For, and Switch)

```
if someInt == 0 { return true } else { return false }
```

For loops share inclusive/exclusive range defs with array splicing:

```
for i in 0...4 {print(i)} //Prints 0 1 2 3 4
for i in 0..<4 {print(i)} //Prints 0 1 2 3
```

```
switch someVariable {
    case 1: "Hello"
    case 2: "Good Bye"
    default: "Nothing" }
```

Closures

Take this function, for example:

```
func isGood(string: String) -> Bool {
   return string == "OK" }
```

Call with: `isGood(string: "OK")`

Closures are self-contained functions, like Python lambdas

```
let isGoodClosure = { string -> Bool in
 return string == "OK" }
```

Call with: `isGoodClosure("OK")`

Structs

Use structs if you just need to store data in a structured object.

```
struct Resolution {
    var width = 0
    var height = 0 }

let hd = Resolution(width: 1920, height: 1080)
```

Classes

```
class Square {
    var side: Int

    init(side: Int) { self.side = side }

    //Custom getter and setter property
    var perimeter: Int {
        get { return 4 * side }
        set (newSide) { side = newSide / 4 } }

    func getArea() -> Int { return pow(side, 2) }
}

var mySquare = Square(side: 5)
print(mySquare.getArea())   //prints 25
mySquare.perimeter = 36
print(mySquare.side)   //prints 9
```