Swift Cheatsheet

1. Declaring Variables of Simple Types

```
var answer = 42              // Makes Int variable with value 42
var temperature : Int        // Makes unitialized Int variable

var isAlive = true           // Bool

var shortPi = 3.1412         // Double - use rather than Float, better precision

var floatyPi: Float = 3.1412  // Float

var greeting = "Hullo"       // String
```

If you are not going to change the value of a variable, you are strongly encouraged to make it a constant so that the compiler can optimise it. You do this by using "let" instead of "var".

```
let pi = 3.1412              // Makes a constant Double
```

2. Common Operators

Binary operators

+	Add	// e.g. y = a+b	Works for strings too
-	Subtract		
*	Multiply		
/	Divide		
%	Remainder		

Binary shorthand

+=	Add and assign	// e.g. y += 21	Equivalent to y = y+21
-=	Subtract and assign		
*=	Multiply and assign		
/=	Divide and assign		
%=	Remainder and assign		

Unary operators

-	Minus	//e.g. y = -x	Gives the negative of a number
--	Decrement	// e.g. y--	Subtracts one from y
++	Increment	// e.g. y++	Adds one to y

Boolean tests - return true or false

<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&&	Logical AND
||	Logical OR

e.g. following will return true when age = 42, name = "Jim" and working = true
age > 27 && (name == "Julia") || working)
Brackets are not needed, but behaviour would change without the bracket after the &&.
Behaviour is X and (Y or Z), but without brackets it would be (X and Y) or Z.

3. Conditionals, loops

If, else else if

```
if comparison1
{
  //called if comparison 1 is true
}
else if comparison2
{
   //called if comparison 2 is true
}
else
{
   //called if nothing else is true
}
```

Curly brackets around consequences are mandatory. Round brackets around condition are optional.
e.g.

```
if  temperature > comfyTemp {
   uncomfortable = true
   println( "Someone turn the fan on" )
} else {
   uncomfortable = false
   println( "Feeling fine" )
}
```

For loops

for initialisation; condition; increment { statements }
e.g.

```
var sum = 0
for i=0; i < 5; i++{ sum = sum + i }        // Gives the answer 10
```

for item in range { statements }
e.g.

```
var sum = 0
for i in 1...5 { sum = sum + i }          // Gives the answer 15
sum = 0
for i in (1..<5) { sum = sum + i }         // Gives the answer 10 as last value not included in range
```

for item in collection { statements }
e.g.

```
var listOfAttendees = [ "Bill", "Jane", "Jim", "Fred", "Ann" ]
for name in listofAttendees{
    println( name )
}
```

While loops

while condition { statements }
e.g.

```
var i = 0
while (i < 100)
{
   println( "I told you so" )
}
```

4. Functions

Functions can have zero or more named parameters, and zero or more results.

```
func fahrenheitToCentrigrade( fTemp: Double ) -> Double {
   return (fTemp - 32.0) / 1.8
}
var temp = fahrenheitToCentrigrade( 32.0 )   //returns 0


func ageAndPensioned( birthYear: Int) -> (age: Int, pensioned: Bool) {
   let age = 2015 - birthYear
   return (age, age > 60)
}
var age: Int
var pensioned: Bool
(age, pensioned) = ageAndPensioned(1922)
// can then use age and pensioned values
```

5. Enumerated Types and Switch statements

Enumerated types

```
//Simple example
enum TempType {
   case degF
   case degC
}
var tempType = TempType.degC
if tempType == .degF   //  would return false


// Example below uses Int as a base type and gives a start value
enum daysOfWeek: Int {
   case Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}

println( daysOfWeek.Sunday.rawValue )   // Will be 7

var today = daysOfWeek.Monday
var whatToDo = ""
switch today{
case .Saturday, .Sunday:
   whatToDo = "Chill - it's the weekend"
default:
   whatToDo = "Apply nose to grindstone"
}

println( whatToDo ) // Will print “Apply nose to grindstone”

today = .Saturday // As we know type of today now, I don’t need to say “daysOfWeek”




// Example below associates a different structure with each possibility
// You can then use that structure to deal with the specific item you have
enum carReg {
   case New( String, Int, String )
   case Old( String, String, String )
   case Custom( String, Int )
}

var myCar = carReg.New("CP", 57, "BHO")
myCar = .Old( "ABB", "007", "P")
myCar = .Custom( "XDOTDOT", 1984 )

func yearOfRegistration( carDets: carReg ) -> String{
   var result =  "unknown"
   switch carDets {
   case let .New(_, middle, _):
      if middle >= 50 {
         result = "\(1950+middle)"
      } else {
         result = "\(2000+middle)"
      }
   case let .Old(_, _, end):
```

```
      switch end {
         case "A": result = "1983"
         case "B": result = "1984"
       default: result = "Between 1985 and 1999"
       }
   case let .Custom( _, year):
      result = String(year)
   }
   return result
}

yearOfRegistration(carReg.New("CP", 57, "BHO"))  // returns 2007
yearOfRegistration(carReg.New("HY", 07, "FPB"))  // returns 2007
yearOfRegistration(carReg.Old("HY", 07, "FPB"))    // returns old
yearOfRegistration( myCar ) // returns old
```

Switch statements

Examples with enumerated types have already been shown above.

```
// Here is an example with Int, classifying people by their age
func agesOfMan( age: Int ) -> String {
   switch age{
   case 0...7:
      return( "infant")
   case 8...17:
      return( "whining schoolboy")
   case 18...25:
      return( "lover sighing like a furnace")
   case 26...39:
      return( "seeking the bubble reputation")
   case 40...59:
      return( "the justice in fair round belly")
   case 60...69:
      return( "Shrinking from the world")
   default:
      return( "Second childishness")
   }
}
agesOfMan( 30 )  // returns "seeking the bubble reputation"
agesOfMan( 6 )    // returns "infant"

// Example  - use of multiple switch items and where clauses
func allowedToDrink( age: Int, country: String ) -> String{
   switch (age, country){
   case ( _, "Saudi Arabia"):      // _ matches anything
      return "Not allowed"
   case (let myAge, _ ) where age < 5:    // use of a where clause - let enables access to value
      return "Infants of \(myAge) are never allowed"
   case ( _, "United States" ):
      if age >= 21 {return "Allowed"}
      else {return "Not allowed"}
   default:
      return "Unknown"
   }
}
allowedToDrink( 44, "Saudi Arabia")   // returns "Not allowed"
allowedToDrink( 4, "England")     // returns "Infants of 4 are never allowed"
```

```
allowedToDrink( 20, "United States")  // returns "Not allowed"
allowedToDrink( 14, "France ")   // returns "Unknown"
```

6. Collections - Strings, Arrays (and tuples), Dictionaries

Strings

Simple string expressions and printing out

```
let emptyString = ""
let greeting = "Hello"
let friendlyGreeting = greeting + ", friend"
```

String interpolation

```
let temp = "Boiling point of water is \(100*1.8 + 32) fahrenheit"
```

String tests

```
if emptyString.isEmpty ...
if greeting.hasPrefix( "Hell" )...
if greeting.hasSuffix( "matey" )...
```

equality tests whether made up of same characters in same order

```
if emptyString == ""     //this is true
if greeting == "hello"   // this is false as lower case H
if “aardvark” < “apple”  // lexically less, so true
```

All characters are unicodes. No idea what happens when you compare emojis.

Arrays

```
var myPets: [String]  =  [ ]                  //Declares empty array of strings
var pets = [String]( )                        //Different way of doing same thing
myPets.count                                  // will be zero

myPets.append( "Chaz the Dog" )
myPets.append( "Dave the Goldfish" )
myPets.count                                  // will be two

let yourPets = [ "Idris the Guinea Pig"]      //Declares immmutable array with one string in it
var ourPets = myPets + yourPets               // Combines the two string arrays in one string array
ourPets.count                                 // it has three elements

ourPets[0] = "Chaz the Bad Dog"               // can replace elements in mutable arrays
let deadPet = ourPets.removeAtIndex(2)        // can delete elements
ourPets                                       // now only two pets
```

```
deadPet                                  // removeAtIndex returns the element removed

ourPets.insert( "Bob the Capybara", atIndex: 0)  // can add extra element at specific position
for pet in ourPets {                     // can loop over the elements
   println (pet)
}

// Initialized fixed size with default values
var readings = [Double](count:200, repeatedValue: 0.0)  // Makes array with 200 values of 0.0
readings.removeLast()                    // Can delete last element
readings.count                           // readings now has 199 elements
```

Tuples

We have already seen some tuples.

Tuple as compound result from a function:

```
func ageAndPensioned( birthYear: Int) -> (age: Int, pensioned: Bool) {
   let age = 2015 - birthYear
   return (age, age > 60)
}
tuple = ageAndPensioned(1922)
if tuple.pensioned { println("The pensioner is \(tuple.age) years old" )}
```

Different tuple as details attached with each case of an enumerated type:

```
enum carReg {
   case New( String, Int, String )
   case Old( String, String, String )
   case Custom( String, Int )
}
var myCar = carReg.New("CP", 57, "BHO")
```

Can also use tuples for temporary sets of details, e.g. when reading in data

```
var tup = (age:23, name:"Bill", female:false)
tup.age                  // This will return 23
tup.female               // This will return false

//Could have an array of such details
var (rray = [ (age:23, name:"Bill", female:false),
   (age:72, name:"Jane", female:true),
   (age:45, name:"Evonne", female:true) ]

tupArray.append(age: 42, name: "Jack", female: false))  // Adds 4th tuple

for item in tupArray {
   var gender = "female"
   if !item.female {gender = "male"}
   println( "\(item.name) is \(item.age) and is \(gender)")
}
```

Dictionaries

```
var pets = [
   "Dave": "Goldfish",
   "Chaz": "Dog",
   "Idris": "Guinea Pig"]

pets["Murphy"] = "Guinea Pig"   //Add Murphy to the list of pets
pets.isEmpty  // returns false
pets.count    // returns 4

// Next two lines are an approximate truth - actually return an optional
pets["Murphy"]      // returns "Guinea Pig"
pets["Dog"]         // returns nil
pets["Chaz"] = "Bad Dog"    // updates entry for Chaz
for (name, animal) in pets {
   println( "\(name) is a \(animal)" )
}

pets["Idris"] = nil   // Deletes Idris from the dictionary
for name in pets.keys {
   println( "\(name) is a pet" )
}
for animal in pets.values {
   println( "We have a \(animal)" )
}
let petNames = [String](pets.keys)  // Makes an array from the keys.
```

Optionals

Under dictionaries, we saw that nil is a possible result when looking up a key:

```
var pets = [ "Dave": "Goldfish", "Chaz": "Dog", "Idris": "Guinea Pig"]
 var result = pets["Jim"]      // should return nil as Jim isn't in dictionary
```

Swift handles this by making the result not to be **String**, but **String?**.
This is called an optional. You cannot use it as a String - you'll get a compile-time error.

You need to unwrap it. You can do this by testing it is not nil then using the ! operator.

```
if (animalRef != nil) {println( animalRef!) }
```

There is a much better shorthand for unwrapping optionals. You can test and assign them at the same time using the form

if let *stringvariable* = *optional* { *use stringvariable safely here* }

So the animal example above could be used as:

```
if let animal = animalRef { println(animal) }
else { println( "Animal Unknown") }
```

You will see the less clear form which saves thinking of another name:

```
if let animalRef = animalRef { println(animalRef) }
else { println( "Animal Unknown") }
```

When writing code that uses Cocoa libraries, you are continually unwrapping results. At that point, you'll probably need the short form that can unwrap more than one optional at once.

```
if let animal = animalRef, let animal2 = animalRef2 {
   println(animal, animal2)
}
else { println( "Animals Unknown") }
```

7. Structs and Classes

Structs

Structs are like simple classes that are copied rather than referenced, and do not have inheritance. If you want to make complex data structures where objects point to each other, then you probably want classes; if all you want is collections of objects, you probably just want Structs.
Structs are intended for data items that won't change - so where you do have a function that changes the data within a Struct, you need to give it the **mutating** keyword or it will cause a compiler error.

In some ways, enums are a third choice in the same dimension - they can have associated data and methods.

The example below uses enums, structs and classes to define a pack of cards.

```
/* Deck of Cards
A playground to explore enums, structs and random numbers
Lets start with by making ranks for the cards*/

import UIKit   // Needed for random function

enum Rank: Int {
   case Ace = 1, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King
   func name() -> String {
      switch self {
      case .Jack:
         return "Jack"
      case .Queen:
         return "Queen"
      case .King:
         return "King"
      case .Ace:
         return "Ace"
      default:
         return String(self.rawValue)
      }
   }

   func shortName() -> String {
      switch self {
      case .Jack:
         return "J"
      case .Queen:
         return "Q"
      case .King:
         return "K"
      case .Ace:
         return "A"
      default:
         return String(self.rawValue)
      }
   }
}
```

```
// Now we declare the four suits in Bridge value order, least first

enum Suit: Int { // Suits in Bridge order
   case Clubs=1, Diamonds, Hearts, Spades
   func name() -> String {
      switch self {
      case .Clubs:
         return "Clubs"
      case .Diamonds:
         return "Diamonds"
      case .Hearts:
         return  "Hearts"
      case .Spades:
         return "Spades"
      }
   }
   func emoji() -> String {
      switch self {
      case .Clubs:
         return "♣"
      case .Diamonds:
         return "♦"
      case .Hearts:
         return  "♥"
      case .Spades:
         return "♠"
      }

   }
}

/*:
Now a `struct` can be used for each card.

Create a struct called "Card" that has a rank of type `Rank` and a suit of type `Suit`

Make Card follow printable protocol so we can provide a description of each card to
routines like print
*/

struct Card: CustomStringConvertible {
   var rank: Rank
   var suit: Suit
   var longDesc: String {
      return "The \(rank.name()) of \(suit.name())"
   }
   var description: String {
      return "\(rank.shortName())\(suit.emoji())"
   }
```

```
}

// Now we can have a pack of cards. We will make this via a Pack class.

class Pack {
  var cardPack: [Card] = []

  init() {
    for suit in [Suit.Clubs, .Diamonds, .Hearts, .Spades] {
      for cardValue in (1...13) {
        let cardRank = Rank(rawValue: cardValue)
        var newCard = Card(rank: cardRank!, suit: suit)
        self.cardPack.append(newCard)
      }
    }
  }

  func drawAnyCard() -> Card {
    // To get a random number `arc4random()` is used.
    var tmpNum = Int(arc4random_uniform(UInt32(cardPack.count)))
    return cardPack.removeAtIndex(tmpNum)
  }

  func dealCard() -> Card {
    // take the first card
    return cardPack.removeAtIndex(0)
  }

  func returnHand( cards: [Card] ){
    // returns to bottom of pack
    for card in cards {
      cardPack.append( card )
    }
  }

  func shuffleDeck() {
    // Swap two random cards 100 times
    var tmpNum1:Int
    var tmpNum2:Int
    var tmpCard:Card
    for i in (1...100) {
      tmpNum1 = Int(arc4random_uniform(UInt32(cardPack.count)))
      tmpNum2 = Int(arc4random_uniform(UInt32(cardPack.count)))
      tmpCard = cardPack[tmpNum2]
      cardPack[tmpNum2] = cardPack[tmpNum1]
      cardPack[tmpNum1] = tmpCard
    }
  }
}
```

```
// Here is some code that uses the pack
var p = Pack()
p.shuffleDeck()

for card in p.cardPack {
   println(card)
}

for i in (1...5){
println( "Now deal: \(p.dealCard())" )
}

println("================")
for card in p.cardPack {
   println(card)
}
```