Kotlin Cheat Sheet and Quick Reference

Declaring Variables

```
var mutable: Int = 1
mutable = 2 // OK: You can reassign a var.
val immutable: Double = 2.0
// immutable = 3.0 // Error: You can't reassign a val!
var greeting = "Hello, world!" // Inferred as String
var catchphrase: String? = null // Nullable type
catchphrase = "Hey, what's up, everybody?"
```

Nullable Types

```
var mutable: Int = 1
mutable = 2 // OK: You can reassign a var.
val immutable: Double = 2.0
// immutable = 3.0  // Error: You can't reassign a val!
var greeting = "Hello, world!" // Inferred as String
var catchphrase: String? = null // Nullable type
catchphrase = "Hey, what's up, everybody?"
var name: String? = null // Can hold a String or null

// Safe cast operator ?.
// length1 contains name's length if name isn't null;
null otherwise
val length1: Int? = name?.length

// Elvis operator ?:
// length1 contains name's length if name isn't null; 0
otherwise
val length2: Int = name?.length ?: 0
// The Elvis operator can also execute statements in
the case of null values.
val length3 = name?.length ?: return

// Non-null assertion operator !!
name = "Francis"
val length4: Int = name!!.length // Works if name isn't
null; crashes otherwise

// Smart casts and checking for null
var nonNullableAuthor: String
var nullableAuthor: String?
if (name != null) { // Checking for null
  nonNullableAuthor = name // Smart cast to String
} else {
  nullableAuthor = name // Smart cast to String?
}
```

Control Flow: if expression

```
// Using if to choose different paths
var condition = true
if (condition) {
  // If condition is true, this gets executed
} else {
  // If condition is false, this gets executed
}
// Using if to set a value
val x = 100
val y = 1
val more = if (x > y) x else y // more == 100
val less = if (x < y) {
  println("x is smaller.")
  x // The last expression is the block's value
} else {
  println("y is smaller.")
  y
}
```

Control Flow: when expression

```
// Using when to choose different paths
val year = 2010
when (year) {
  2010 -> print("Froyo")
  2011 -> print("Ice Cream Sandwich")
  2008, 2009 -> print("The early days")
  in 2012..2015 -> {
      println("Jellybean through Marshmallow,")
      println("when things got interesting.")
  }
  else -> println("Some other era")
}
// Using when to set a value
val androidEra = when (year) {
  2010 -> "Froyo"
  2011 -> "Ice Cream Sandwich"
  2008, 2009 -> "The early days"
  in 2012..2015 -> {
      print("Jellybean through Marshmallow")
      // The last expression is the block's value
      "When things got interesting"
  }
  else -> "Some other era"
}
// Using when with conditionals to set a value
val catsOwned = 2
val dogsOwned = 1
val judgement = when {
  catsOwned == 0 -> "No cats"
  catsOwned < 0 -> {
      print("Call the cat police!")
      // The last expression is the block's value
      "Owes someone some cats"
  }
  catsOwned == 1 && dogsOwned == 1 ->
    "Seeking balance"
  catsOwned > 0 && catsOwned < 3 -> "Yay cats!"
  else -> "Cat Nirvana"
}
```

Collections: List

```
val immutableList = listOf("Alice", "Bob")
val valMutableList = mutableListOf("Carol", "Dave")
var varMutableList = mutableListOf("Eve", "Frank")
// One way to test membership
val isBobThere1 = "Bob" in immutableList
// Another way to test membership
val isBobThere2 = immutableList.contains("Bob")
val name: String = immutableList[0] // Access by index
valMutableList[1] = "Bart" // Update item in list
// immutableList[1] = "Bart" // Error: Can't change
valMutableList.add(2, "Ellen") // Add item at index
// Delete by index
val removedPerson = valMutableList.removeAt(1)
// Delete by value
val wasRemoved = valMutableList.remove("Bart")
// You can change the contents of a val mutable
collection, but you CAN'T reassign it:
// You can change the contents of a var mutable
collection, and you CAN reassign it:
varMutableList[0] = "Ellen"
varMutableList = mutableListOf("Gemma", "Harry")
```

Collections: Map

```
val immutableMap = mapOf("name" to "Kirk", "rank" to
"captain")
val mutableMap = mutableMapOf("name" to "Picard",
"rank" to "captain")
// Is this key in the map?
val hasRankKey = immutableMap.containsKey("rank")
// Is this value in the map?
val hasKirkValue = immutableMap.containsValue("Kirk")
// Access by key, returns nullable
val name: String? = immutableMap["name"]
// Update value for key
mutableMap["name"] = "Janeway"
// Add new key and value
mutableMap["ship"] = "Voyager"
mutableMap.remove("rank") // Delete by key
// Delete by key and value
mutableMap.remove("ship", "Voyager")
// Won't work, value doesn't match
mutableMap.remove("name", "Spock")
```

Collections: Set

```
// Sets ignore duplicate items, so immutableSet has 2
items: "chocolate" and "vanilla"
val immutableSet = setOf<String>("chocolate",
"vanilla", "chocolate")
val mutableSet = mutableSetOf("butterscotch",
"strawberry")
// One way to test membership
val hasChocolate1 = "chocolate" in immutableSet
// Another way to test membership
val hasChocolate2 = immutableSet.contains("chocolate")
mutableSet.add("green tea") // Add item
// Delete by value
val flavorWasRemoved = mutableSet.remove("strawberry")
```

Kotlin Cheat Sheet and Quick Reference

Control Flow: loops

```
// Iterate over list or set
for (item in listOrSet) {
  println(item)
}
// Iterate over map
for ((key, value) in myMap) {
  println("$key -> $value")
}
// Iterating over ranges
for (i in 0..10) {} // 0 to 10
for (i in 0 until 10) {} // 0 to 9
for (i in 1..10 step 2) {} // 1, 3, 5, 7, 9
for (i in 10 downTo 1) {} // 10 to 1
// while and do while
var x = 0
while (x < 10) {
  x++
  println(x)
}
do {
  x--
  println(x)
} while (x > 0)
```

Functions

```
fun sayHi() { // A Unit function
  println("Hello")
}
// Function with parameters
fun sayHello(name: String) {
  println("Hello, $name!")
}
// Function with default arguments
fun sayFriendlyHello(name: String = "Friend") {
  print("Hello, $name!")
}
// Function with mix of regular and default arguments
fun createCat(name: String = "Kitty", age: Int,
isSpayed: Boolean = false) {
  print("$name / $age / $isSpayed")
}
createCat(age = 1) // Using just the non-default
argument
createCat("Fluffy", 2, true) // One way to call a
function
// Calling a function with named arguments
createCat(age = 2, isSpayed = true, name = "Fluffy")
// Function with parameters and return value
fun total(x: Int, y: Int): Int {
  return x + y
}
// A function as a single expression
fun product(x: Int, y: Int) = x * y
// A function that accepts another function
fun doMath(mathOperation: (Int, Int) -> Int, a: Int, b:
Int): Int {
  return mathOperation(a, b)
}
// Calling a function that accepts another function
val add = doMath(::total, 2, 3)
val multiply = doMath(::product, 2, 3)
```

Lambdas

```
// Lambda
val adder: (Int, Int) -> Int = { x, y -> x + y}
// Lambda with single parameter: it keyword
val square: (Int) -> Int = { it * it}
// Passing a lambda to a function
val addWithLambda = doMath(adder, 2, 3)
```

Extensions

```
// Add the "fizzbuzz()" function to the Int class
fun Int.fizzBuzz(): String {
  return when {
      this % 3 == 0 -> "fizz"
      this % 5 == 0 -> "buzz"
      this % 15 == 0 -> "fizzbuzz"
      else -> this.toString()
  }
}
println(6.fizzBuzz()) // Prints "fizz"
println(8.fizzBuzz()) // Prints "8"
// Add the "absValue" property to the Int class
val Int.absValue: Int
  get() = abs(this)
println((-3).absValue) // Prints "3"
```

Objects

```
// Only a single instance exists
// Takes the place of static utility classes
object Constants {
  const val baseUrl = "http://api.raywenderlich.com"
}
```

Classes

```
// Class basics
class Spaceship(var name: String, val size: Int) {
  var speed: Int = 0
  fun fly() {
    speed = 100
  }
  fun isFlying(): Boolean {
    return speed > 0
  }
  // Companion object replaces static members
  companion object {
    fun newSpaceship(): Spaceship {
      return Spaceship("Falcon", 25)
    }
  }
}
val myShip = Spaceship("Enterprise", 150)
myShip.fly()
val flying = myShip.isFlying()
```

```
class Sailor(var rank: String, var lastName: String) {
  // Class properties with accessors
  var fullName: String
    get() = "$rank $lastName"
    set(value) {
      val (firstWord, remainder) = value.split(" ",
limit = 2)
      rank = firstWord
      lastName = remainder
    }
}
// Subclassing: only open classes can be subclassed
open class Crewmember(val name: String) {
  // Only open methods can be overridden
  open fun sayHello() = "Hello, I'm crewmember $name."
}
// Subclassing
class Captain(name: String): Crewmember(name) {
  override fun sayHello() = "Greetings! I am Captain
$name."
}
```

Data Classes

```
// A data class is a structured data container
// with pre-defined toString() and other overrides
data class Student(val name: String, var year: Int)
// name is a read-only property, year is mutable
val newStudent = Student("Siddartha", 1)
// Data class with properties outside the constructor
data class Professor(val name: String) {
  var isTenured: Boolean = false
}
val newProfessor = Professor("Snape")
newProfessor.isTenured = true
```

Enum Classes

```
enum class Taste {
  SWEET, SOUR, SALTY, BITTER, UMAMI
}

val vinegarTaste: Taste = Taste.UMAMI
// Iterating through an enum class
for (flavor in Taste.values()) {
  print("Taste: ${flavor.ordinal}: ${flavor.name}")
}
```

Sealed Classes

```
// Like enum classes, but can make multiple instances
sealed class Shape {
  class Circle(val radius: Int): Shape()
  class Square(val sideLength: Int): Shape()
}
val circle1 = Shape.Circle(3)
val circle2 = Shape.Circle(42)
val square = Shape.Square(5)
```