Kotlin Basic Syntax Cheatsheet

Example Kotlin File

```
package my.demo

import java.util.*

fun main(args: Array<String>) {
   ...
}

fun sum(a: Int, b: Int) : Int {
  return a + b
}

class MyFirstClass(){
  fun generateId(): String {
    return
  }
  ...
}
```

Note:

- Multiple classes can be defined in a single file
- Name of file does **not** need to match class name
- Functions can be declared outside of classes.

Data Types

Everything is an object, no primitive types !!

- Byte, Short, Int, Long
- Float, Double
- Char, String
- Boolean

Root class is Any (similar to Java's Object). Null values are instances of Nothing.

NB - By default, data types will not allow nulls. Suffix datatype by '?' to indicate nullable values.

```
var a:Int = null    <-- Generates compile error
var b:Int? = null   <-- Nullable
val c = "Welcome to Kotlin"  <-- Immutable value with inferred data type of String

\\ String Interpolation injects variables into Strings
println("The current value is $a")
```

Handling Nulls

Null Pointer Exceptions are one of the most common causes of errors in Java applications. Kotlin has been designed to prevent these errors.

- Need to explicitly state that variable/parameter allows nulls
- Unsafe calls on potentially null values are prevented by the compiler

```
    var a: String = "abc"
    a = null // compilation error as variable does not allow null

    var b: String? = "abc"
    b = null // ok

    b = getName()  // Call a function that returns a Nullable String
    b.length // Compilation Error, invoking method on potentially Null Object
    if (b != null){
        b.length // Can now execute method on the variable
    }

    // Safe Call Operator - Returns length or Null as an Int?
    var i = b?.length

    // Elvis Operator - Returns either the length of the String or -1 if b is null
    b?.length ?: -1

    // Null Chaining
    val city = order?.customer?.address?.city

    val city = order?.customer?.address?.city
                  ?: throw IllegalArgumentException("Invalid Order")
```

Tuples

Kotlin doesn't have inbuilt support for tuples so each function can only return a single value.

However there are a couple of inbuilt wrapper classes for tuples with 2 or 3 components called

Pair and Triple.

```
class Book(val id: Int, val title:String, val author:String, val price:Float){
    fun getIdAndTitle(): Pair<Int, String>{
        return Pair(id, title)
    }
}

val idAndTitle = book.getIdAndTitle()
val id = idAndTitle.first
val title = idAndTitle.second
```

Control of Flow

Conditional Statements

```
if (response.isSuccess()){
    processResponse(response)
} else {
    processError(response)
}

val max = if (a > b) a else b
```

When conditions

```
val ordinal = when (x){
   1 -> "1st"
   2 -> "2nd"
   3 -> "3rd"
   else -> "$(x)th"
}

fun getWarmth(color: Color) = when(color) {
    Color.RED, Color.ORANGE, Color.YELLOW -> "warm"
    Color.GREEN -> "neutral"
    Color.BLUE, Color.INDIGO, Color.VIOLET -> "cold"
}

// Type checking
when (e) {
    is Num -> e.value
    is Sum -> eval(e.right) + eval(e.left)
    else -> throw IllegalArgumentException("Unknown expression")
}
```

Loops

```
while (condition){
  /* .... */
}

do {
    /*...*/
} while (condition)
```

Iteration through Collections, Sequences, & Ranges

```
for (item in collection) print(item)
for (i in 1..100) { ... }

val myMap = mapOf(....)
for ((key, value) in myMap) {
    println("$key = $value")
}
```

Function Definitions

```kotlin
fun sum(a: Int, b:Int) : Int {
return a + b
}

// Using inferred return types
fun sum(a: Int, b: Int) = a + b

// Default parameter values
fun sendHttpRequest(url: String,
method: String = "GET",
headers: Map? = null) {
…
}
// Call method with named parameters
sendHttpRequest("http://test.mydomain.com/customer/1",
headers = mapOf("Accept","application/json"))
```

Extension Functions

Functions can be added to existing classes without needing to extend or subclass. These 'Extension Functions' are then available on all instances of that type.

```
fun String.toCamelCase() : String {
   return this.split(' ')
            .map { it -> it.toLowerCase()
            .capitalize() }
            .joinToString(separator = "")
}

"This is an example".toCamelCase() // returns ThisIsAnExample
```

Collections

```
val names = listOf("Andrew","Jane", "Dennis", "Charlotte")
val names = mutableSetOf("Andrew","Jane", "Dennis", "Charlotte")
val animals = mapOf(1 to "Dog", 2 to "Cat", 3 to "Squirrel")
val animals = mutableMapOf(1 to "Dog", 2 to "Cat", 3 to "Squirrel")

names.map { name -> name.first() }
names.map { it.length }
names.filter{ it.length > 4 }.map{ name -> name.first() }
val longestName = names.maxBy { it.length }
```

Classes and Inheritance

```
class Customer(name: String) {
    init {
        logger.info("Customer initialized with value ${name}")
    }
}

val customer = Customer("Guy Edwards")

// Define class with properties which can be accessed directly.
class Book(val title:String, val author:String, val price:Float)
fun createBook(){
    val book = Book("Fly Fishing", "J R Hartley", 20.99f)
    println(book.title)
}

interface Clickable {
    fun click()
```

```
}
// ':' is used as equivalent to Java's 'extends' & 'implements'
class Button : Clickable {
    override fun click() = println("I was clicked")
}
```

Data classes provide a simple way to create domain objects with standard Java methods such as equals,
toString, hashCode, …

```
data class Person(val id: String,
    val forename: String,
    val surname: String,
    val dateOfBirth: Date)
```

Functional Programming

Functions can be declared as variables, passed as parameters and returned from functions.

```
// Define a function which takes a function as a parameter & applies the function t
fun twoAndThree(operation: (Int, Int) -> Int) {
    val result = operation(2, 3)
        println("The result is $result")
}
// Define a variable which is a function to multiply 2 integers
val multiply = { x: Int, y : Int -> x * y}

// Invoke the function
twoAndThreee(multiply)

// Should see the output
The result is 6
```

Lambda Functions

Lambda functions provide an alternative to Java's anonymous classes and are a convenient way to pass a functionality into functions.

```
button.setOnClickListener { alert("You clicked the button") }
```

More

We have just scratched the surface of this language & there is plenty more to learn.

- Coroutines - Lightweight threading which simplifies asynchronous programming
- Android - One area where Kotlin is really making an impact
- Native - Compilation to Native code to run on a variety of platforms
- Compilation to JavaScript

Further Information

- Kotlin Language Official Documentation - Online documentation
- Kotlin Koans - Short exercises to get you started with Kotlin
- Kotlin Slack Channel - Official Slack channel for Kotlin. Very friendly & helpful, frequented by JetBrains developers
- Kotlin Coroutines - Introduction to Kotlin Coroutines
- Kotlin Yorkshire Meetup Group - Local Kotlin User Group

Books

- Kotlin in Action - Dmitry Jemerov and Svetlana Isakova
- Kotlin for Android Developers - Antonio Leiva