Kotlin Quick Reference

This reference summarizes the topics covered in the Kotlin Bootcamp course in the form of code snippets. See the Kotlin Language Documentation for full reference. See the Kotlin Koans for more snippets to practice with. See the Kotlin Bootcamp course if you need anything explained.

Lesson 0

- Install JDK if you don't have it
- Link to downloading IntelliJ
- Starting the interpreter: **Tools > Kotlin > Kotlin REPL**

Lesson 1

Hello Kotlin function	**Hello Kotlin program**

```
fun printHello () {
    println ("Hello Kotlin")
}

printHello()
```
	```
	fun main (args: Array<String>) {
	   println("Hello ${args[0]} ")
	}
	```

Operators	**Type conversion**

```
*, fish.times(6)
/, fish.div(10)
+, fish.plus(3)
-, fish.minus(3)
```

```
1.toLong()
1.toString()
```

Number formatting

```
val oneMillion = 1_000_000
val socialSecurityNumber = 999_99_9999L
```

val (immutable) & var (mutable)

```
val aquarium = "my aquarium"
var fish = 50
var snails : Int = 12
```

Nullability

```
var rocks: Int = null //Error
var marbles: Int? = null
fishFoodTreats?.dec()

var lotsOfFish: List<String?> =
    listOf(null, null)

return fishFoodTreats?.dec() ?:0
goldfish!!.eat
```

Strings / String Templates

```
"hello" + "fish" + "!"
"I have $numberOfFish fish"
"Print ${ numberOfFish + 5 } fish"
"fish" == "fish"
val message = "You are ${ if (length <
5) "fried" else "safe" } fish"
```

if/else

```
if (numberOfFish > numberOfPlants) {
    println("Good ratio!")
} else {
    println("unhealthy ratio")
}

if (fish in 1..100) println(fish)

val isHot =
  if (temperature > 90) true else false
```

When

```
when (numberOfFish) {
  0  -> println("Empty tank")
  in 1..50 -> println("Got fish!")
  else -> println("Perfect!")
}
```

listOf / mutableListOf

```
val myList =
    mutableListOf("tuna",,"shark")

myList.remove("shark") // OK!

val swarm = listOf(fish, plants)
```

arrayOf / mutableArrayOf / intArray…

```
val school =
    arrayOf("tuna","salmon","shark")

val mix = arrayOf("fish", 2)

println(Arrays.toString(intArrayOf(2,
"foo")))
```

```
val bigSwarm = arrayOf(swarm,
arrayOf("dolphin","whale","orka"))

val array = Array (5) { it * 2 }
```

for loop

```
for (element in swarm) {...}
for ((index, element) in
swarm.withIndex()) {
    println("Fish at $index is
$element")
}
for (i in 'b'..'g') print(i)

for (i in 1..5) print(i)

for (i in 5 downTo 1) print(i)

for (i in 3..6 step 2) print(i) //
Prints: 35
```

Lesson 2

Functions

```
fun randomDay(): String {return
"Monday"}

fun fishFood (hour: Int, day: String =
"Tuesday"): String {}

fun isTooHot(temperature: Int): Boolean
= temperature > 30
```

Compact Functions

```
fun isTooHot(temperature: Int) =
temperature > 30

fun shouldChangeWater (day: String,
temperature: Int = 22, dirty: Int =
20): Boolean {
    return when {
       isTooHot(temperature)-> true
       else  -> false
    }
}

fun getDirtySensorReading() = return 20

fun shouldChangeWater (day: String,
temperature: Int = 22, dirty: Int =
getDirtySensorReading()) {...}
```

Filters

Lambdas (anonymous functions)

```
{ println("Hello") }()
```

```
println( decorations.filter {it[0] ==
'p'})
```

```
val waterFilter = { dirty: Int -> dirty
/ 2 }
val waterFilter : (Int) -> Int = {
dirty -> dirty / 2 }
```

Higher order functions (fun with fun arg)

```
fun updateDirty(dirty: Int, operation:
(Int) -> Int): Int {
   return operation(dirty)
}
updateDirty(50, ::increaseDirty)
```

Lesson 3

Class

```
class Aquarium(var length: Int = 100,
var width: Int = 20, var height: Int =
40) {

constructor(numOfFish: Int): this() {

    init {
     // do stuff
    }

    val volume: Int
       get() {
         return w * h * l / 1000
       }

    init {
       // do stuff with volume
    }
}
```

Visibility

package:
`public` - default. Everywhere
`private` - file
`internal` - module

class:
`sealed` - only subclass in same file

inside class:
`public` - default. Everywhere.
`private` - inside class, not subclasses
`protected` - inside class and subclasses
`internal` - module

Inheritance

```
open class Aquarium ….. {
    open var water = volume * 0.9

    open var volume
}

class TowerTank (): Aquarium() {
```

Abstract classes

```
abstract class AquariumFish {
   abstract val color: String
}

class Shark: AquariumFish() {
    override val color = "gray"
}
```

```
override var volume: Int
   get() = (w * h * l / 1000 *
PI).toInt()
   set(value) {
       h = (value * 1000) / (w * l)}
}
```

```
class Plecostomus: AquariumFish() {
    override val color = "gold"
}
```

Interfaces

```
interface FishAction  {
   fun eat()
}

class Shark: AquariumFish(), FishAction
{
   override val color = "gray"
   override fun eat() {
       println("hunt and eat fish")
   }
}

fun feedFish(fish: FishAction) {
   // make some food then
    fish.eat()
}
```

Data Classes

```
data class Decorations(val rocks:
String, val wood: String, val diver:
String){
}

val d = Decorations("crystal", "wood",
"diver")
val (rock, wood, diver) = d


dataClassInstance1.equals(dataClassInst
ance2)
val
dataClassInstance3.copy(dataClassInstan
ce2)
```

Composition

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


fun delegate() {
   val pleco = Plecostomus()
   println("Fish has has color
${pleco.color}")
   pleco.eat()
}

interface FishAction {
   fun eat()
}

interface FishColor {
   val color: String
}

object GoldColor : FishColor {
   override val color = "gold"
}

class PrintingFishAction(val food:
String) : FishAction {
   override fun eat() {
       println(food)
   }
}

class Plecostomus (fishColor: FishColor
= GoldColor):
       FishAction by
PrintingFishAction("eat a lot of
algae"),
       FishColor by fishColor
```

Singleton / object

```
object Database

object MobyDickWhale {
    val author = "Herman Melville"
}
```

enum

```
enum class Color(val rgb: Int) {
   RED(0xFF0000), GREEN(0x00FF00),
BLUE(0x0000FF);
}
Color.RED
```

Lesson 4

Pairs

```
val equipment = "fishnet" to "catching
fish"
println(equipment.first)
println(equipment.second)

val (tool, use) = fishnet
val fishnetString = fishnet.toString()
println(fishnet.toList())


Nesting with parentheses:
val equip = ("fishnet" to "catching
fish") to ("of big size" to "and
strong")
equipment.first.first
```

Lists

```
val testList =
listOf(11,12,13,14,15,16,17,18,19,20)
listOf<Int>(1,2,3,4,5,6,7,8,9,0).revers
ed()

var symptoms = mutableListOf("white
spots", "red spots", "not eating",
"bloated", "belly up")
symptoms.add("white fungus")
symptoms.remove("white fungus")
symptoms.contains("red")
println(symptoms.subList(4,
symptoms.size))

listOf(1, 5, 3).sum()
listOf("a", "b", "cc").sumBy {
it.length }
```

Mapping

```
val cures = hashMapOf("white spots" to
"Ich", "red sores" to "hole disease")

println(cures["white spots"])

cures.getOrDefault("bloating", "sorry,
I don't know")

cures.getOrElse("bloating") {"No cure
for this"}

val inventory = mutableMapOf("fish net"
to 1)

inventory.put("tank scrubber", 3)
```

Constants

```
const val CONSTANT = "top-level
constant" // compile time

object Constants {
const val CONSTANT2 = "object constant"
}

class MyClass {
    companion object {
        const val CONSTANT3 = "constant
in companion"
    }
}
```

```
inventory.remove("fish net")
```

Extension functions

```
fun String.hasSpaces(): Boolean {
   val found = this.find { it == ' ' }
   return found != null
}

fun extensionExample() {
   “Does it have spaces?”.hasSpaces()
}

⇒ fun String.hasSpaces() = find { it
== ' ' } != null

fun AquariumPlant.isRed() = color ==
"red"

fun AquariumPlant?.pull() {
   this?.apply {
       println("removing $this")
   }
}
```

Property extensions

```
val AquariumPlant.isGreen: Boolean
   get() = color == "green"

fun propertyExample() {
   val plant = GreenLeafyPlant(30)
   plant.isGreen // true
}
```

Generic classes

```
class MyList<T> {
   fun get(pos: Int): T {
   TODO("implement")
}
fun addItem(item: T) {}
}

fun workWithMyList() {
   val intList: MyList<String>
   val fishList: MyList<Fish>
}
```

Generics: Full example

```
open class WaterSupply(var
needsProcessed: Boolean)

class TapWater : WaterSupply(true) {
   fun addChemicalCleaners() {
       needsProcessed = false
   }
}

class FishStoreWater :
WaterSupply(false)

class LakeWater : WaterSupply(true) {
   fun filter() {
       needsProcessed = false
   }
}

class Aquarium<T>(val waterSupply: T)

fun genericsExample() {
   val aquarium = Aquarium(TapWater())
aquarium.waterSupply.addChemicalCleanes
()
}
```

Generic constraint

Non-nullable:
```
class Aquarium<T: Any>(val waterSupply:
T)

class Aquarium<T: WaterSupply>(val
waterSupply: T)
```

In and Out Types

```
class Aquarium<out T: WaterSupply>(val
waterSupply: T) { …}

interface Cleaner<in T: WaterSupply> {
   fun clean(waterSupply: T)
}
```

Generic functions and methods

```
fun <T: WaterSupply>
isWaterClean(aquarium: Aquarium<T>) {
   println("aquarium water is clean:
${aquarium.waterSupply.needsProcessed}"
)
}

fun genericsFunExample() {
   val aquarium = Aquarium(TapWater())

   isWaterClean(aquarium)
}

fun <R: WaterSupply>
hasWaterSupplyOfType() = waterSupply is
R
```

Inline / reified

```
inline fun <reified R: WaterSupply>
hasWaterSupplyOfType() = waterSupply is
R

inline fun <reified T: WaterSupply>
WaterSupply.isOfType() = this is T

inline fun <reified R: WaterSupply>
Aquarium<*>.hasWaterSupplyOfType() =
waterSupply is R
```

Annotations

```
@file:JvmName(“InteropFish”)
@JvmStatic fun interop()

annotation class ImAPlant
@ImAPlant class Plant{...}

val plantObject = Plant::class
for (a in plantObject.annotations) {

println(a.annotationClass.simpleName)
}
```

Reflection

```
val classobj=Plant::class
for(m in
classobj.declaredMemberFunctions){
  println(m.name)
}
```

Annotations for getters and setters

```
@Target(PROPERTY_GETTER)
annotation class OnGet
@Target(PROPERTY_SETTER)
Annotation class OnSet

@ImAPlant class Plant {
   @get:OnGet
   val isGrowing: Boolean = true

   @set:OnSet
   var needsFood: boolean = false
}
```

Labeled breaks

```
fun labels() {
   loop@ for (i in 1..100) {
       for (j in 1..100) {
           if (i > 10) break@loop
       }
   }
}
```

Lesson 5

Lambda recap

```
myFish.filter {
it.name.contains("i")}.joinToString ("
") { it.name }
```

Higher order function

```
fun myWith(name: String, block:
String.() -> Unit) {
    name.block()
}
```

Standard Library: apply & run

```
fish.run {
   name
}

val fish2 = Fish().apply {
     name = “sharky”
}
```

Standard Library: with & repeat

```
with(fish.name) {
   println(name)
}

repeat(3) { rep ->
    println(" current repetition:
$rep")}
```

Inline

```
Inline fun myWith(name: String,
operation: String.() -> Unit) {
    name.operation()
}
```

Lambda instead ofSAM

```
fun example() {
   runNow {
      println(“Passing a lambda as a
Runnable”)
}
```