Kotlin Basic Cheatsheet [Keyword, Note]

val [immutable]
var [mutable]
DataType [9]
number.toDouble() [Type Cast/Change Type]
num.plus() / num.minus() [build in function]
string[0] [String char position]
in 10..20 [Range/ elvis operator]
fun add(name:String = "Anwar") [Default value]
add(name="Anwar") [Parameter specific value pass]
add(name: String) = "Anwar" [Single expression function or Inline return]

Array

val array = arrayOf(1, 2, 3)
val array:Array<Int> = arrayOf(1, 2, 3)
print(Arrays.toString(array))
array[0] [index wise print]
array.first() [build in function]
array.last() [build in function]
array.set(position, value) [build in function]
print(array.indexOf(value)) [return index]
val array = intArrayOf(1, 2, 3) [type is fixed which is int]
array.sum() [build in function]
array.max() [build in function]
array.min() [build in function]
array.count() [build in function]
array.sortedArray() [build in function for sort by default ASC]
array.find{it:DataType } [build in function iterate item one by one]

List/MutableList

val list = listOf(1, 2, 3)
list.first() [build in function]
list.last() [build in function]
list.find{it: } [build in function iterate item one by one]
list.filter{it: } [build in function filter item one by one]
list.map{it: } [build in function iterate item and change item one by one]
list.filterNot{it: } [build in function not filter item one by one]

list.slice(0..3) [print slice of list by index range]
list.subList(1,3) [print range of list not include last position]
list.take(3) [another list first 3 element]
list.elementAtOrNull(8) [prevent index outOfBound exception]
list.sort() [another list first 3 element]
val eList = emptyList<Int>() [empty list]

Map/MutableMap

val map = mapOf(key to value) | mapOf("one" to 1)
val mutableMap = mutableMapOf(key to value) | mutableMapOf("one" to 1)
map[key] [get value by key]
map.count() [build in function]
map.size() [build in function]
map.put(key, value) [put value in map]
map.remove(key) [build in function]
map.containsKey(key) [check key is exists]
map.containsValue(value) [check value is exists]
map.filter{} [filter by key or value]
map.filterKeys{} [filter by key]
map.filterValues{} [filter by value]
map.toSortedMap() [build in function]
map.getOrDefault(key, default) [build in function]
map.keys [all keys]
map.values [all values]
map.filterNot{it: } [build in function not filter item one by one]

Set/MutableSet

val sets = setOf(1, 2, 3, 4) [set only collection of unique value]
val mutableSets =mutableSetOf(1, 2, 3) [set only collection of unique value]
sets.add(value) [build in function]
sets.remove(value) [build in function]
sets.contains(value) [build in function]
sets.count() [build in function]
sets1 union sets2 [two sets union in one sets]
sets1 intersect sets2 [two sets union in one sets]

Loops: for/while/forEach

```
for(name in list){}
for(num in 0..100){} [range left & right include]
for(num in 0 until 100){} [range not right include]
for(num in 0 until 100 step 2){} [range not right include with step]
for(index in list.indices){} [print only index]
for((index, value) in list.withIndex){} [print only index & value]

var num = 0;
while(num < 100){
        print(num)
        num++
}

list.forEach{name-> print(name)}
list.forEachInIndex{index, name-> }
```

Class

```
val person = Person()
class Person{
        fun add(){
        }
}
```

```
[this is primary constructor]
class Person(name: String){

[init block called after primary constructor called and then called secondary constructor]
        init{

        }

[this is secondary constructor also we have to call primary constructor if primary constructor has parameter]
        constructor(age: Int):this(name){

        }
        fun add(){
        }
}
```

Companion Object [java static method or static variable]

```
class Person{
        companion object{
                fun add(){ [now add() is static method]

                }
        }
}
```

Singleton [java static method or static variable]

```
class object Singleton{
        fun add(){
        }
}
```

[This is singleton we can use inside all method like java static method]
Singleton.add()

Class/Data Class

[both object same location in heap just because of data class]

```
data class User(name: String)
val userOne = User("Anwar")
val userTwo = User("Anwar")
if(userOne = userTwo) [true]

data class User1(name: String, age: Int)
val user1 = User1("Anwar", 30)
print(user1.name)
print(user1.component1)
print(user1.component2)

val user2 = user1.copy(age=30) [change only age]
val (name: String, age: Int) = user2
```

Inheritance

[By default all class and method final in kotlin only inheritable when use open keyword]

[inheritable class & method]

```
open class BaseClass{
        open fun add(){
        }
}
```

Abstract Class

[Abstract class we can't create object we can only extends or inheritance]

```
abstract class Person{
        abstract fun add()
}
```

Interface

```
interface Computer{
        fun operatingSystem(): String
        fun buildYear(): String
        fun buildBy(): String = "This is default method"
}
```

Pair/Triple

[only two data type we can use in pair]

```
val pair = Pair<String, Int>("Anwar", 30)
print(pair.first)
print(pair.second)
val (name:String, age:String) = Pair("Anwar", 30)
val list = pair.toList() [convert pair to list]
```

[any number of data type we can use in triple]

```
val triple = Triple<String, Int, String…..>("Anwar", 30, "Hossain"....)
```

lateinit

```
class Declaration{
        lateinit var name: String [mutable and we can assign value later]
        fun setValue(name: String){
                name = name
        }

        fun getValue() = if(::name.isInitialized) name else "Not Initialized"

}
```

Lazy [lazy variable immutable]

```
val user: UserDetails by lazy {
        UserDetails("Anwar", 30)
}

class UserDetails(name:String, age: Int){
        init{
                println("This name is $name")
                println("This age is $age")
        }
}
```

Enum

```
enum class StudentType{
        MALE,
        FEMALE
}
class Student(val name: String, val type: StudentType){
        init{
                print("$name is $type")
        }
}
val studentOne = Student("Anwar", StudentType.MALE)

enum class StudentType(val type: String){
        MALE(type),
        FEMALE(type)
}
for(type: StudentType in StudentType.values()){
        println(type.name) [name means enum name]
        println(type.ordinal) [ordinal means index]
}
```

Exception

```
val age = 30
if(age < 30){
        throw Exception("Age is less than 30")
}
```

Scope Function [let, run, also, apply]

	it	this
return result	let	run
return same object	also	apply

```
class Person(val name: String, val age: Int){
        var address = "Unknown"

        fun getInfo() = "Name = $name & age = $age"
}
val info = Person("Anwar", 30).let{it: ObjectType
        return it.age.plus(10)
}
val info = Person("Anwar", 30).run{this: ObjectType
        this.address = "Dhaka, Bangladesh" [this keyword use optional]
        return getInfo()
}
val person = Person("Anwar", 30).apply{this: ObjectType
        this.address = "Dhaka, Bangladesh" [this keyword use optional]
        this.age = 30
        [By default return same object that means return this]
}
val numbers = mutableListOf("One", "Two", "Three", "Four")
val finalResult = numbers
                        .map{ it.length }
                        .also{ print(it) } [after change in map then check value]
                        .filter{ it > 3}
```

Generics

```
class Event<T>(value: T? = null){
        init{
                println(value.toString().length)
        }
}
Event<String>("Anwar")
Event("Anwar")
Event() [used default value]
```

Lambda Functions

```
fun square(number: Int) = number * number
print(square(5))

[now convert lambda]
val square: (Int)->Int = number * number
[Unit means don't return anything]
```

Higher Order Functions

```
[function pass as a parameter & return function]
val addition = doAddition()
fun doAddition(): (Int, Int)->Int{
        return ::addNumber
}
fun addNumber(numOne: Int, numTwo: Int)-> = numOne.plus(numTwo)
printName{ [This is perimeter function body]
        println("Hello World")
}
fun printName(print: ()-> Unit){
        print()
}
```

Extension Functions

```
val number = 0
number.arithmetic(20)
fun Int.arithmetic(number: Int){
        println("Addition is ${this + number}")
        println("Subtraction is ${this - number}")
        println("Multiplication is ${this * number}")
}
"Anwar".midValue()
fun String.midValue(){
        if(this.length % 2 == 0){
                println("It has no mid value")
        }else{
                println("this[this.length / 2]")
        }
}
```

Sealed Class

[*flexibility to handle different state of subclass]
[*restricted number of hierarchies]
[*preferred over enum and abstract classes]

```
enum class Result(val message: String){
        SUCCESS("Success")
        ERROR("Error", exception [it's not possible])
}
sealed class Employee
data class Manager(val name: String, val age: Intt, val team: List<String>):
Employee()
class SeniorDev(val name: String, val age: Intt, val projects: Int):
Employee()
object class JuniorDev: Employee()
```

[sealed class is given the flexibility of having different type of subclass]
[others class can't extends sealed class]
[we can't create the object of a sealed class]

```
val employee: Employee = SeniorDev("Anwar", 30, 10)
val message = when(employee){
        is Manager-> TODO()
        is SeniorDev-> TODO()
        is JuniorDev-> TODO()
}
print(message)
```

@JVMStatic,@JVMOverloads, @JVMField

[What is the benefit of Kotlin?]
[Expressive and Concise - That's mean less boilerplate code]
[Safe code - That's mean nullable]
[Interoperable - That's mean we can create Kolin code inside Java code]
[Structure Concurrency - That's mean Coroutines]
@JVMStatic [object class fun/method use in java static manner]
@JVMOverloads [Kotlin function default value feature enabled in Java]
@JVMField [Kotlin data class field use in Java without getValue() manner]

[Please check below link details]
https://blog.mindorks.com/jvmstatic-jvmoverloads-and-jvmfield-in-kotlin