**//List Example**

var listData = mutableListOf("Abhishek", "Rai", "CK", "PK", "DK")

listData.add("Sandeep Singh")

**//Set Example**

var setData = mutableSetOf("Ck", "PK", "Dk", "CK")

**//Map Example**

var map = mutableMapOf<String, Int>("Name" to 1, "Address" to 100)

map.put("phoneNO", 24243)

**//ForEach loop on list**

listData.forEach{ e ->

println("forEach Iteams is $e")

}

**//For loop on list**

for(items in listData){

println("For loop Items is $items")

}

**//Iterate items from list using Iterator**

var iteratorData: Iterator<String> = listData.iterator()

while(iteratorData.hasNext()){

println("Iterator Next Data items is ${iteratorData.next()}")

}

// **Iterate items from map**

**for(key in map.keys){**

println("Items key= $key and value is ${map[key]}}")

}

**//Deblicate Element find or remove from the list**

val concrete = listOf(1, 3, 5, 7, 2, 8, 2, 1);

var uniqList = mutableListOf<Int>();

for(element in concrete){

if(uniqList.contains(element)){

println("Dublicate element found: $element")

}else{

uniqList.add(element)

}

}

**OR**

val concrete2 = listOf(1, 3, 5, 7, 2, 8, 2, 1);

var uniqList2 = mutableListOf<Int>();

for(elementIndex in 0.. concrete2.size-1){

if(uniqList2.contains(concrete2 [elementIndex])){

println("Dublicate element found: ${ concrete2 [elementIndex]}")

}else{

uniqList2.add(concrete2 [elementIndex])

}

}

**//Print list/all dublicate item from the list in kotlin**

val list = listOf(1, 2, 3, 4, 5, 2, 4, 6, 7, 8, 4, 9, 1, 5, 2, 2)

**//Print only Dublicate items from list => outputl => [1, 2, 4, 5] OR 1, 2, 4, 5**

val duplicates = list.groupBy { it }.filter { it.value.size > 1 }.keys

println("Duplicate items: $duplicates")

println("Duplicate items: ${duplicates.joinToString()}")

**//Print all repeted / dublicate item from list => Expected output => [1, 2, 4, 5, 2, 4, 4, 1, 5, 2, 2] OR 1, 2, 4, 5, 2, 4, 4, 1, 5, 2, 2**

val duplicatesX = list.filter { item -> list.count { it == item } > 1 }

println(duplicatesX)

println(duplicatesX.joinToString())

**//Find the dublicate items with count(item: count) => 1 count: 2 , 2 count: 4 , 4 count: 3 , 5 count: 2**

val duplicatesItemsCount = list.groupBy { it }.filter { it.value.size > 1 }

for (entry in duplicatesItemsCount) {

val item = entry.key

val count = entry.value.size

println("$item count: $count ")

}

**//Reverse The items from list => reverseItemsList: [2, 2, 5, 1, 9, 4, 8, 7, 6, 4, 2, 5, 4, 3, 2, 1]**

val reverseItemsList = list.asReversed()

println("reverseItemsList: $reverseItemsList")

**//Combine the two list**

val lisData = listOf(2,4,3,6,2,8,3,8,1,5)

val combineTwoList = list.union(lisData)

println("combineTwoList: $combineTwoList")

**//Distinct or insersection of two list or comman items in two list**

val distinctInTwoList = list.intersect(lisData)

println("distinctInTwoList: $distinctInTwoList")

**//UniqueItems items in list [1, 2, 3, 4, 5, 6, 7, 8, 9]**

val uniqueItems = list.distinct()

println("UniqueItems items in list $uniqueItems")

val lisData = listOf(2,4,3,6,2,8,3,8,1,5) **//2,3,2,8,3,8**

var uniqLisData = mutableListOf<Int>()

var dublicateLisData = mutableListOf<Int>()

for(item in lisData){

if(uniqLisData.contains(item)){

dublicateLisData.add(item)

println("Dublicate item is $item")

}else{

uniqLisData.add(item)

}

}

for(item in lisData){

if(dublicateLisData.contains(item)){

print("$item")

}

}

//**Given number is odd or Even**

var no1 = 12

if(no1 % 2 == 0){

println("Given number is even")

}else{

println("Given number is odd")

}

//**Given number is Prime or not**

var no2 = 43

var isPrime = true;

for(i in 2..<no2){

if(no2 % i == 0){

isPrime = false

println("Multiplication found $i and $no2")

break;

}

}

if(isPrime){

println("Given number $no2 is prime")

}else{

println("Given number $no2 is not prime")

}

//**Fectorial of Given number**

val no3 = 10;

var factorialResult = 1

for(item in 1..no3){

factorialResult = factorialResult \* item

}

println("factorialResult of $no3 is $factorialResult")

//**Reverse the given number**

var no4 = 124

var temp = no4

var revNo = 0

while(temp != 0){

val rem = temp % 10

revNo = revNo \* 10 + rem

temp = temp / 10;

}

println("Actual No is: $no4 and revNo is: $revNo")

//**Reverse the given number**

var no5 = 121

var temp5 = no5

var revNo5 = 0

while(temp5 != 0){

val rem = temp5 % 10

revNo5 = revNo5 \* 10 + rem

temp5 = temp5 / 10;

}

if(no5 == revNo5){

println("Given No is: $no5 palindrome")

}else{

println("Given No is: $no5 is not palindrome")

}

//**Given String is palindrome or not**

val strVal = "RADAR"

var reverseStr = ""

for (i in strVal.length - 1 downTo 0) {

reverseStr += strVal[i]

}

if(strVal == reverseStr){

println("Given String is: $strVal palindrome")

}else{

println("Given String is: $strVal is not palindrome")

}

//Reverse the String

val str1 = "Abhishek"

val revstr = StringBuilder(str1).reverse().toString()

println("Reverse of $str1 String is $revstr")

//Some of Natural number: all o....n(1, 2, 3, 4, ...100) is natural number

val no6 = 100

var sum = 0

for(i in 1..no6){

sum += i;

}

println("Sum of natural number: $no6 is $sum")

//Deblicate Element find or remove from the list

val concrete = listOf(1, 3, 5, 7, 2, 8, 2, 1);

var uniqList = mutableListOf<Int>();

for(element in concrete){

if(uniqList.contains(element)){

println("Dublicate element found: $element")

}else{

uniqList.add(element)

}

}

var uniqList2 = mutableListOf<Int>();

for(elementIndex in 0..concrete.size-1){

if(uniqList2.contains(concrete[elementIndex])){

println("Dublicate element found: ${concrete[elementIndex]}")

}else{

uniqList2.add(concrete[elementIndex])

}

}

val arrr :MutableList<Int> = mutableListOf(3,76,8,5,7,9,2)

val sizex = arrr.size -1

for (i in 0..sizex){

var lowstIndex = i;

for (j in i..sizex){

if(arrr[j] < arrr[lowstIndex]){

lowstIndex = j

}

}

val temp = arrr[i];

arrr[i] = arrr[lowstIndex]

arrr[lowstIndex] = temp

}

println("sortest list ${arrr}")

println("second Lowest ${arrr[1]}")

println("Second highest Items is ${arrr[sizex-1]}")

**open class SuperParrent{**

var Address = "Abhishek" //Can not be access from chil class

open var phoneNo = "9988766" // will be accessable from child class

//will be accessable from child class

open fun addOne(){

println("SuperParrent One")

}

}

**open class Parrent: SuperParrent(){**

override var phoneNo = "29694"

open var name = "Abhishek"

override fun addOne(){

println("Parrent One")

}

open fun addTwo(){

println("Parrent two")

}

}

**class Child: Parrent(){**

override var name = "CK"

override fun addTwo(){

println("Child two")

}

}

**abstract class MyAbtract{**

abstract fun addTwoNo()

fun MyAbtract(){

println("ChildAbtract class addThreeNo")

}

}

**class ChildAbtract: MyAbtract(){**

override fun addTwoNo(){

println("ChildAbtract class addTwoNo")

}

}

**interface MyInterface {**

//Mandatory method to override in child class

fun bar()

//Option method to override in child class

fun foo() {

// optional body

println("MyInterface class bar")

}

}

**class C : MyInterface**{

override fun bar(){

println("child class bar")

}

override fun foo(){

super.foo()

println("child class foo")

}

}

//var emp = Employee("Rudra")

**class Employee (name:String,age: Int){**

init{

println("init block is calling name $name and age is $age")

}

constructor (name: String):this(name, 20){

println("child constroctor is calling: name is $name")

}

}

// CompanionObjectExample.doSomthing()

// CompanionObjectExample.myObj.doSomthing()

**class CompanionObjectExample{**

companion object myObj{

fun doSomthing(){

println("companion object method called")

}

}

}

**val userList = mutableListOf<UserInfo>()**

userList.add(UserInfo("Abhishek", "abh01@gmail.com"))

userList.add(UserInfo("Abhishek 2", "abh02@gmail.com"))

userList.add(UserInfo("Abhishek 3", "abh03@gmail.com"))

userList.add(UserInfo("Abhishek 4", "abh04@gmail.com"))

**println("size of user list ${userList.size}")**

for(user in userList){

if(user.userName.equals("Abhishek")){

println("user found")

}

}

**//Search / find the user from list using find method**

val personWithName = userList.find { it.userName == "Abhishek" }

println("user found $personWithName")

**//get the index from object / elemet**

val listpp = listOf("A", "b", "c", "d")

val objectIndex = listpp.indexOf("b")

println("user found 2nd way $objectIndex")

**//Lamda funtion**

val sumLamda = { x: Int, y: Int -> x + y }

OR

val sumLamda: (Int, Int) -> Int = { x, y -> x + y }

val sumCC = sumLamda(1, 9)

println("sum of Lamda is $sumCC ")

fun addOfTwoNo(x: Int, y: Int): Int {

return x + y

}

fun MulOfNo(x: Int, y: Int): Int {

return x \* y

}

**fun higherOrderFunction(n: Int, dynamicMethodWithTwoIntParm: (Int, Int) -> Int): Int{**

return dynamicMethodWithTwoIntParm(n + 3, 8)

}

**fun higherOrderFunction2(n1: Int, n2:Int, add: (Int, Int) -> Int, multi: (Int, Int) -> Int): Int{**

val add = add(n1, n2)

val mul = multi(n1, n2)

return add + mul

}

**Note: Mark double :: when you are passing function but not required with lamda function. Like**

**val sumVal: Int = higherOrderFunction(10, ::addOfTwoNo)**

println("Higher order funtion call Add $sumVal ")

**val sumVal: Int = higherOrderFunction(10, sumLamda)**

println("Higher order funtion call Add $sumVal ")

val mulVal: Int = higherOrderFunction2(10, 20, ::addOfTwoNo, ::MulOfNo)

println("Higher order funtion call Mul $mulVal ")

//Data class example

**data class UserInfo(val userName: String, val userEmail: String)**

//Extention funtion Example

**fun String.PrintName(strSirName: String): String {**

println("full name is $this $strSirName ")

return "$this + $strSirName";

}

**//calling Extention funtion**

val printNameExt = "Abhishek".PrintName("Rai")

println("Extention funtion Concatinate two string is $printNameExt")

**infix fun Int.AddTwoNo(n2: Int): Int{**

println("sum of two number is ${this + n2} ")

return this + n2

}

**//Calling companion object**

A.AddTowNo(2)

**class A{**

**companion object getObject{**

val PI = 3.14

fun AddTowNo(no2: Int){

println("sum of two nu is ${PI + no2} ")

}

}

}

**//calling infix funtion for add two number**

val addSum = 5 AddTwoNo 10

println("Infix funtion Sum of two no is $addSum")

**object MyObjectClass {**

fun getInstance(): MyObjectClass{

return this

}

}

**//Calling Object class**

val v1 = MyObjectClass.getInstance()

val v2 = MyObjectClass.getInstance()

val v3 = MyObjectClass.getInstance()

println("Printing object class $v1 $v2 $v3")

**val myRandomList = mutableListOf(1, 2, 0, 4, 3, 0, 5, 0) // 1, 2, 4, 3, 5, 0, 0, 0**

var tmpArr = mutableListOf<Int>() //

var countZero = 0

for(index in 0..myRandomList.size-1){

val items = myRandomList.get(index)

if(items != 0){

tmpArr.add(items); // 1, 2,

}else{

countZero = countZero + 1

//tmpArr.add(myRandomList.size - (index +1) , items); //10 2 = 8

}

}

for(items in 0..countZero){

//if(items == 0){

tmpArr.add(0)

//}

}

println(" $tmpArr ")

**OPPS**

**Constroctor**

A class in Kotlin can have at most one primary constructor, and one or more secondary constructors. The primary constructor initializes the class, while the secondary constructor is used to initialize the class and introduce some extra logic.

**/\***

**Problam Statement:** suppose we have two class A & B abd both having bothe type constructor primary & secondary & init block. Bothe class Primary cunstroctor having one parameter and secondary having two parameter. Class act as a Parrent class & class is act as Child class and class C inhirites class B. Pls tell me which constroctor and init block call 1st and what will be oredr or calling

**Parrent init**

**parrent secondary cons call**

**Child init**

**child secondary cons call Dear 10**

**\*/**

**//val ob1 = C("Dear", 10);**

**open class P(name: String){**

public var name = "Abhishek"

init{

println("Parrent init")

}

constructor(name: String, age: Int): this(name){

println("parrent secondary cons call")

}

}

**class C(nax: String): P(nax, 1){**

init{

println("Child init")

}

constructor(name: String, xx: Int): this(name){

println("child secondary cons call $name $xx")

}

fun pritName(){

println("Name is $name")

}

}

**Open**

The open keyword allows classes, functions, and properties to be extended, while public is a visibility modifier that doesn't have any explicit usage since all classes, functions, and properties are publicly visible by default

**//Open keyword will help to inheritance from child class.**

**//If not make it open then will not inheritance from child class**

**open class AB1{**

//Will be accessable from child class But not modify

var name = "Parrent"

//Will be accessable from child modify

open var name1 = "Parrent"

//Will access from child class but not override

fun M1(){

}

//Will access from child class but not override

open fun M2(){

}

}

**class Child(): AB1(){**

override var name1 = "Child"

override fun M2(){

println("child class M2 method")

}

}

**Abstract**

Abstract classes are **partially defined classes, methods and properties which are no implementation but must be implemented into derived class**

In Kotlin, an abstract class is a class that cannot be instantiated and is meant to be subclassed. An abstract class may contain both abstract methods (methods without a body) and concrete methods (methods with a body). An abstract class is used to provide a common interface and implementation for its subclasses.

//**A class declear by using abstract keywork is partial implemanted class and need to implement all abstract method in the child class**

//Open keyword will help to inheritance from child class. If not make it open then will not inheritance from child class

**abstract class AB1{**

//Will be accessable from child class But not modify But Not Mandatory

var name = "Parrent"

//Will override access & Modify from child class But Not Mandatory

open var name1 = "Parrent"

//Will be accessable & modify from child But Mandatory

abstract var name2: String

//Will access from child class but not override But Not Mandatory

fun M1(){

}

//Will override, access & Modify from child class But Not Mandatory

open fun M2(){

}

//Will override, access & Modify from child class But Mandatory

abstract fun M3()

}

**class Child: AB1(){**

// Not Mendatory to override

override var name1 = "Abhishek"

// Mendatory to override

override var name2 = "Abhishek"

// Mendatory to override

override fun M3(){

println("child class M2 method")

}

}

**Interface**

In Kotlin, an interface is a collection of abstract methods and properties that define a common contract for classes that implement the interface. An interface is similar to an abstract class, but it can be implemented by multiple classes, and it cannot have state

//By default all variable & method declear in interface class is public, open, abstract.

//we need to implement it in child class

**interface IF1(){**

//variable inisialization is not allowed & default it public, open, abstarct

val name: String

//default it public, open, abstarct

fun M1()

//abstract declearaction is optional and default it public, open, abstarct

abstract fun M2()

fun M3(){

println("optional method")

}

}

**class Child: IF1{**

//We need to override this variable in chaild class

//bocz all default all variable in chaild class is abstarct, open, public

override var name = ""

//All method which only declear and not define then it's abstarct and

// we need to implement it in child class

override fun M1(){

}

//Method dec;ear using abstarct keyword is required to implement in chail

override fun M2(){

}

//Thsi is optional method and it's totally depand on your choice for implemantion

override fun M3(){

}

}

**Object**

In Kotlin, object is a special class that only has one instance. If you create a class with the object keyword instead of class , the Kotlin compiler makes the constructor private, creates a static reference for the object, and initializes the reference in a static block.

It’s create only single object for that and same object will return

A Kotlin object is like a class that can't be instantiated so it must be called by name. ( a static class per se) The android converter saw that your class contained only a static method, so it converted it to a Kotlin object.

//Const 'val' are only allowed on **top level(Object class & Companion Object)**, in named objects, or in companion objects

//object Child() => Constructors are not allowed for objects

object Child {

const val v1 = "V1.2"

final const val v2 = " V3.0"

fun getInstance(){

println("Object class: $v1 and $v2")

}

}

**Companion objects**

If we need to provide the Singleton behavior, then we are better off with Objects , else if we just want to add some static essence to our classes, we can use Companion objects

companion object is Kotlin's equivalent of static members in Java. Everything that applies to static, applies to companion object as well

**Note: We can declare the companion object with Interface, Abstract class, Object class and Class**

//Const 'val' are only allowed on top level, in named objects, or in companion objects

//object Child() => Constructors are not allowed for objects

//Const 'val' are only allowed on top level, in named objects, or in companion objects

//object Child() => Constructors are not allowed for objects

val obj = Child.v1

val obj1 = Child.getInstance()

println("Abstract Class ${AbstarctClass.vXX} ${AbstarctClass.AddTwoNumber()}")

println("Abstract Class ${InterfaceClass.vXX} ${InterfaceClass.AddTwoNumber()}")

**class Child : AbstarctClass(), InterfaceClass{**

companion object {

const val v1 = "V1.2"

final const val v2 = " V3.0"

fun getInstance(){

println("Object class: $v1 and $v2")

}

}

}

**abstract class AbstarctClass{**

//abstract fun P()

companion object{

val vXX = "Abstarct class companion object ";

fun AddTwoNumber(){

println("sum of two number")

}

}

}

**interface InterfaceClass{**

//abstract fun X()

companion object{

val vXX = "Interface class companion object";

fun AddTwoNumber(){

println("sum of two number")

}

}

}

**Sealed Class**

Sealed classes and interfaces represent restricted class hierarchies that provide more control over inheritance. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear outside the module and package within which the sealed class is defined.

A sealed class **defines a set of subclasses within it**. It is used when it is known in advance that a type will conform to one of the subclass

Sealed classes are used for representing restricted class hierarchies wherein the object or the value can have value only among one of the types, thus fixing your type hierarchies. Sealed classes are commonly used in cases, where you know what a given value to be only among a given set of options.

Diff b/w Sealed & Enum class

In some sense, sealed classes are similar to [enum](https://kotlinlang.org/docs/enum-classes.html) classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances, each with its own state. Means ENUM can be hold only single type of value

enum classes can't extend a sealed class (as well as any other class), but they can implement sealed interfaces.

Sealed classes allow us to create instances with different types, unlike Enums which restrict us to use the same type for all enum constants

In a sealed class, we can simply add multiple custom constructors depending on what we need. Furthermore, we can define multiple functions with different names, parameters, and return types. In an enum class, however, we can't define different functions in each enum constant.

Note:

**//Sealed class child will be Simple class, Data class, Abstract class,**

**//But not Interface & Object class**

**// A sealed class is**[**abstract**](https://kotlinlang.org/docs/classes.html#abstract-classes)**by itself, it cannot be instantiated directly and can have abstract members.**

var x = MyResponse.P(1, "Helow")

printResponseData(x)

**sealed class MyResponse(){**

class X(var n1: Int, var n2: Int) : MyResponse()

class P(var n1: Int, var n2: String) : MyResponse()

sealed class IOError(code: Int): MyResponse()

//object RuntimeError(str: String) : MyResponse() //Erro will be: Constructors are not allowed for objects

}

**fun printResponseData(resp: MyResponse){**

when(**resp**){

is MyResponse.P -> println("P")

is MyResponse.X -> println("X")

}

}

**Extension Function**

Kotlin gives the programmer the ability to add more functionality to the existing classes, without inheriting them. This is achieved through a feature known as extensions. When a function is added to an existing class it is known as Extension Function

Kotlin extension function **provides a facility to "add" methods to class without inheriting a class or using any type of design pattern**

**"Abhishek".myExtentionFunction("Rai" , 31)**

fun String.myExtentionFunction(str2: String, age: Int){

println("your complete string is $this $str2 and age is $age")

}

Inline Function:

Inline function **instruct compiler to insert the complete body of the function wherever that function gets used in the code**

The use of inline function enhances the performance of higher order function. The inline function tells the compiler to copy parameters and functions to the call site

In Kotlin, the higher order functions or lambda expressions, all stored as an object which takes some memory and might increase the memory overhead. Sometimes, the function does a very basic functionality that passing the control is not even worth but as it (the function) is being used at multiple places, we need to create a separate function. To reduce the memory overhead of such functions we can use the inline keyword which ultimately requests the CPU to not allocate any memory for the function and simply copy the body of that function at the calling place.

Note: One one parameters accepted by inline function

//'infix' modifier is inapplicable on this function: must have a single value parameter

"Abhishek".myInlineFunction("Rai")

val x = "Ck" myInlineFunction "Singh"

infix fun String.myInlineFunction(str2: String): Int{

println("your complete string is $this $str2")

return 121;

}