Lesson 2
Kotlin Fundamendals

Agenda

- What is Kotlin?
- Kotlin Features
- main() function
- Mutable and Immutable
- Kotlin Strings
- Looping
- Null safety
- Class and Objects
- Inheritance, Interface and Data class

Develop Android apps with Kotlin

- The Android team announced during Google I/O 2017 that Kotlin is an official language to develop Android Apps.
- It's expressive, safe, and powerful.
- Best of all, it's interoperable with our existing Android languages and runtime. Kotlin is concise while being expressive.
- It contains safety features for nullability and immutability, to make your Android apps healthy and performant by default.

Develop Android apps with Kotlin

- Write better Android apps faster with Kotlin. Kotlin is a modern statically typed programming language that will boost your productivity and increase your developer happiness.
- Reference resources :
  - https://developer.android.com/kotlin/
  - https://developer.android.com/kotlin/faq

What is Kotlin?

- Kotlin is a JVM based language developed by JetBrains5 , a company known for creating IntelliJ IDEA, a powerful IDE for Java development.
- Android Studio, the official Android IDE, is based on IntelliJ.
- Kotlin was created with Java developers in mind, and with IntelliJ as its main development IDE.
  - Learn more about from https://developer.android.com/kotlin/index.html
  - Try online : https://try.kotlinlang.org/
  - Find Answers about Kotlin from https://developer.android.com/kotlin/faq.html

Kotlin Features

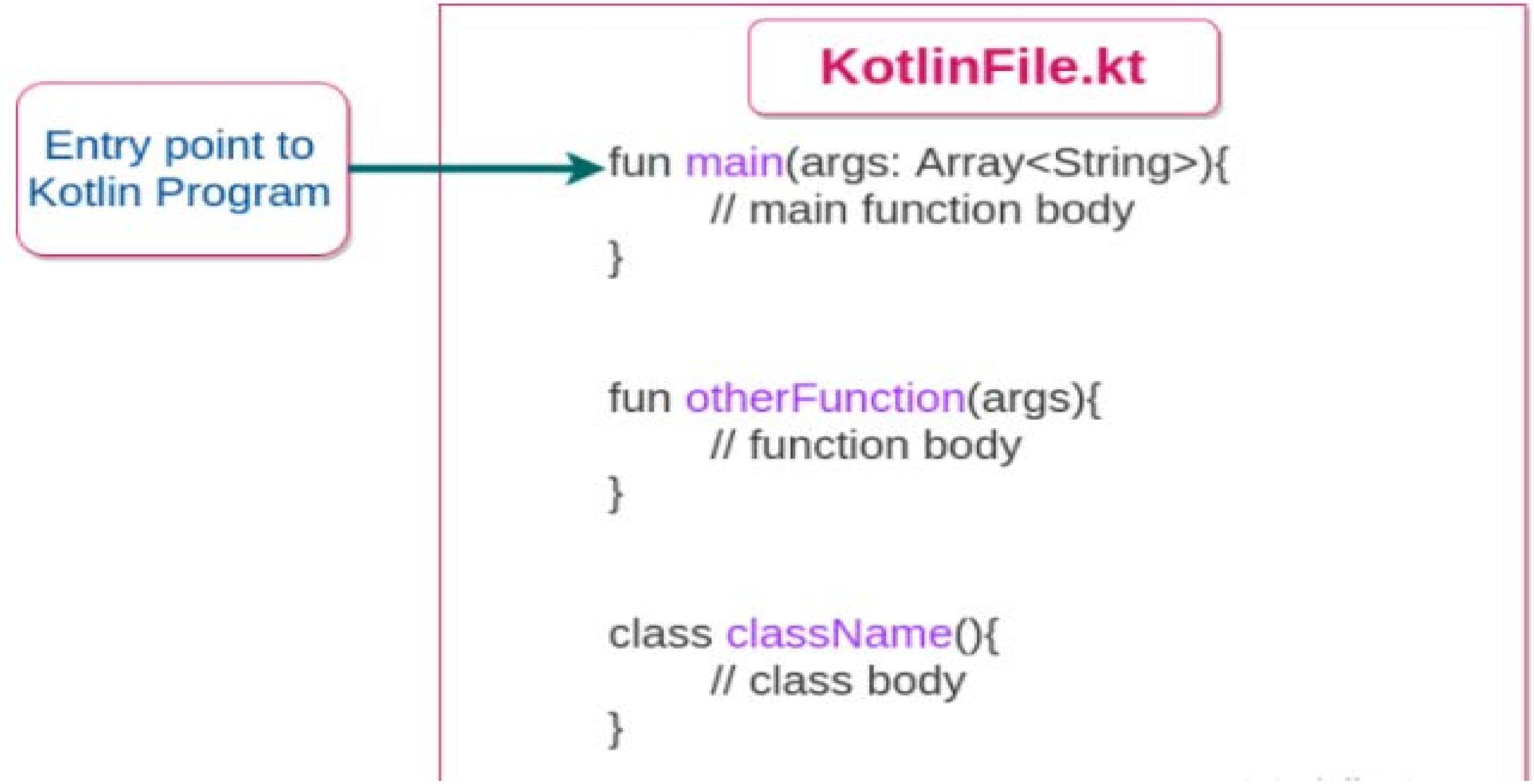
- **Modern and expressive:** You can write more with much less code.
- **It’s safer :** Kotlin is null safe, which means that we deal with possible null situations in compile time, to prevent execution time exceptions.
- **Functional and object-oriented:** Kotlin is basically an object oriented language, also gains the benefits of functional programming.
- **Statically typed:** This means the type of every expression in a program is known at compile time, and the compiler can validate that the methods and fields you’re trying to access exist on the objects you’re using.
- **Free and open source:** The Kotlin language, including the compiler, libraries, and all related tooling, is entirely open source and free to use for any purpose.
- **It’s highly interoperable:** You can continue using most libraries and code written in Java, because the interoperability between both languages is excellent. It’s even possible to create mixed projects, with both Kotlin and Java files coexisting. Easily convert the existing Java code into Kotlin by selecting Code >Convert Java File to Kotlin File

Kotlin Data Types

- Integer Data Types
  - Byte, Short, Int and Long
- Floating Point Data Types
  - Float, Double
- Boolean
  - Accepts true or false
- Character Data Type
  - Char
- String
- All of the above data types are actually objects, each of which provides a range of functions and properties that may be used to perform a variety of different type specific tasks. These functions and properties are accessed using so-called dot notation.

main function in Kotlin

This is the main function, which is mandatory in every Kotlin application. The Kotlin compiler starts executing the code from the main function.



```
fun main(args: Array<String>){
        // main function body
}


fun otherFunction(args){
        // function body
}


class className(){
        // class body
}
```

Mutable and Immutable (Variables and Constants)

- Kotlin is categorized as a statically typed programming language. Uses a technique referred to as *type inference* to identify the type of the variable.
- Mutable/Variable : type is not required.

  var answer = 42 is similar to → **var** answer: Int = 42
- Immutable/ Constants :

  val answer = 42 is similar to → **val** answer: Int = 42
- Refer : DataTypes.kt, GettingInput.kt

Kotlin Strings

- Strings in kotlin represent an array of characters. Strings are immutable. It means operations on string produces new strings.

Eg: val str = “ Kotlin Strings”

 println(str)

- we can create a multi line string using """

Eg: val x: String = """Kotlin

supports

Multiline

Strings"""

- The above code produce the intent spacing from the second line. To avoid spacing you can trim the space by giving

Eg: val x: String = """Kotlin

supports

Multiline

Strings""".trimMargin()

- **String templates**: String templates is a powerful feature in Kotlin when a string can contain expression and it will get evaluated.

```
val x = “David"
val y = "My name is $x"
println(y)
Println("My name is $x with the length
${x.length}")
```

```
Output : My name is David
         My name is David with the length 5
```

- **String operations**

Eg:

```
var s: String = "Hello"
println("Length of string $s is ${s.length}")
 println("Init cap of string is ${s.capitalize}")
println("Lower case is ${s.toLowerCase}")
println("Upper case is ${s.toUpperCase}")
```

Refer : Strings.kt

Refer for more info :
https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html

Kotlin Operators

Type	Operators
Arithmetic Operators	+ - * / %
Assignment Operators	+ = - = *= /= %=
Increment and decrement operators	++ --
Comparison and Equality Operators	> < >= <= == !=
Logical Operators	&& ||
Sign operators	+ and -. They are used to indicate or change the sign of a value.
Range Operator	.. (Eg: x..y → the range of numbers starting at x and ending at y.

Kotlin loops for, forEach, repeat, while, do-while

- **for :** for loop iterates through anything that provides an iterator.

Syntax : for (item **in** collection) { body}

Example :

```
var daysOfWeek =
listOf("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday")
   for(day in daysOfWeek){
      println(day)
   }
```

- **for with index :**

```
for ((index, value) in
daysOfWeek.withIndex()) {
   println("the element at $index is
$value")
}
```

- forEach can be used to repeat a set of statement for each element in an iterable.

```
daysOfWeek.forEach{
      println(it)
   }
```

- while and do..while work as usual in Java
- **repeat :** repeat statement is used when a set of statements has to be executed N-number of times.

  Eg:
  ```
  repeat(4) {
     println("Hello World!")
  }
  ```

Refer : forloop.kt

Kotlin - Ranges, when expression

Ranges

- You can create a range in kotlin via **..** operator.
- Example

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

Result: 1 2 3 4 5

- downTo : Using downTo function we can go reverse in a range.
- for(i in 5 downTo 1) { print(i) }

Result: 5 4 3 2 1

- step : Using step function we can increase the step

for (i in 1..10 step 2) { print(i) }

Result: 1 3 5 7 9

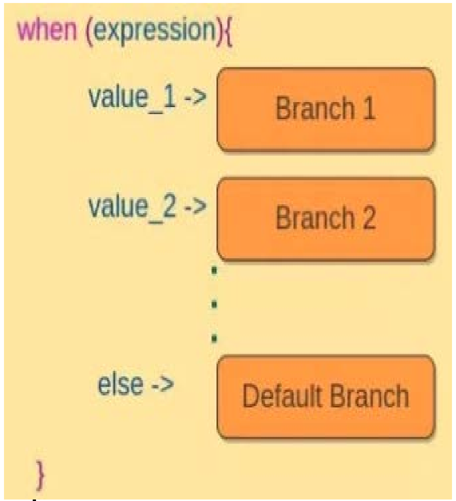
if we want to exclude the last value in a range use until

for (i in 1 until 5) { print(i) }

Result: 1 2 3 4

When expression

- Kotlin when expression is kind of switch case in Java, but concise in syntax, extended in functionality and more fun. Using “Any” object type with **when** expression makes is really broad in the usage.



- **Example**

when (x) {
 1 -> print("x == 1")
2 -> print("x == 2")
 else -> { // Note the block
  print("x is neither 1 nor 2")
  }
 }

Refer : WhenExample.kt

Single Expression Function

- When a function contains a single expression, it is not necessary to include the braces around the expression. All that is required is an equals sign (=) after the function declaration followed by the expression.
- Way 1

```
fun multiply(x: Int, y: Int): Int {    return x * y }
```

- Way 2

```
fun multiply(x: Int, y: Int): Int = x * y
```

- Way 3 – no need to specify return type

```
fun multiply(x: Int, y: Int) = x * y
```

Declaring Default Function Parameters

- Kotlin provides the ability to designate a default parameter value to be used in the event that the value is not provided as an argument when the function is called. This simply involves assigning the default value to the parameter when the function is declared.
- Example

```
fun buildMessageFor(name: String = "Customer", count: Int = 0) : String {
   return("$name, you are customer number $count")
}
```

- Calling the function

```
val message = buildMessageFor("John", 10)
```

- To improve code readability, the parameter names may also be specified when making the function call:

```
val message = buildMessageFor(name = "John", count = 10)
val message = buildMessageFor(count = 10)
val message = buildMessageFor("John")
```

Refer : functions.kt

Kotlin - Null Safety

- Null Safety in Kotlin is to eliminate the risk of occurrence of NullPointerException in real time.
- These are the scenarios, you will get NullPointerException in Kotlin
  1. If you make a explicit call to **throw a NullPointerException**
  2. use**!!** operator. (see in the example section)
  3. You may be aware that you can run external Java code in your application. (From outside Kotlin)
  4. Data inconsistency with regard to initialization
- Way to handle Null Safety in Kotlin
  1. Differentiate between nullable references and non-nullablereferences.
  2. User explicitly checks for a null if conditions
  3. Using a Safe Call Operator (?.)
  4. Elvis Operator (?:)
  - Refer : NullabilityCheck.kt

Way to handle Null Safety in Kotlin

- Way – 1 - Differentiate between nullable references and non-nullablereferences.
  - Kotlin’s type system can differentiate between nullable references and non-nullable references. **?** operator is used during variable declaration for the differentiation.

```
//Non- nullable – You cannot assign null to
//the non-null variable
var a: String = “Hello"
a = null // compilation error
```

```
// If you want to assign null value use ? operator
var a: String? = “Hello"
a = null // OK
```

- Way – 2 – Nullable Check
  - The Kotlin system will tell you an error if you want to call a method from a nullable variable. Always check before accessing whether it is null or not.

```
var a: String? = “Hello"
val l = if (a != null) a.length else -1
```

Way to handle Null Safety in Kotlin

- Way – 3 - Safe Calls (?.)
  - The safe call operator returns the variables property only if the variable is not null, else it returns null. So the variable holding the return value should be declared nullable.

```
fun main(args: Array){
    var b: String? = "Hi !"     // variable is declared as nullable
    var len: Int?
    len = b?.length
    println("b is : $b")
    println("length is : $len")

    b = null
    len = b?.length
    println("b is : $b")
    println("length is : $len")
}
```

Result for the Code

```
b is : Hi !
length is : 4
b is : null
length is : null
```

Way to handle Null Safety in Kotlin

- Way – 4 – Elvis Operator (?:)
  - If reference to a variable is not null, use the value, else use some default value that is not null.

    This might sound same as explicitly checking for a null value.

```
fun main(args: Array){
    var b: String? = " David" // variable is declared as nullable
    val len = b?.length ?: -1
    println("length is : $len")
    val noname = b?: “No one knows me"
    println(“Name is : $noname")
}

Result for the Code

length is : 5
Name is : No one knows me
```

The !! Operator (not-null assertion operator)

- Despite the safety measures Kotlin provides to handle NPE, if you need NPE so badly to include in the code use !! operator.
- You can use this operator, if you are 100% sure that variable holds a non null value, or else you will get NullPointerException.

```
fun main(args: Array){
    var b: String? = "Hello"     // variable is declared as nullable
    var blen = b!!.length
    println("b is : $b")
    println("b length is : $blen")

    b = null
    println("b is : $b")
    blen = b!!.length // Throws NullPointerException
    println("b length is : $blen")
}
```

Result for the code

```
b is : Hello
b length is : 5
b is : null

Exception in thread "main" kotlin.KotlinNullPointerException
    at ArrayaddremoveKt.main(Arrayaddremove.kt:9)
```

Arrays

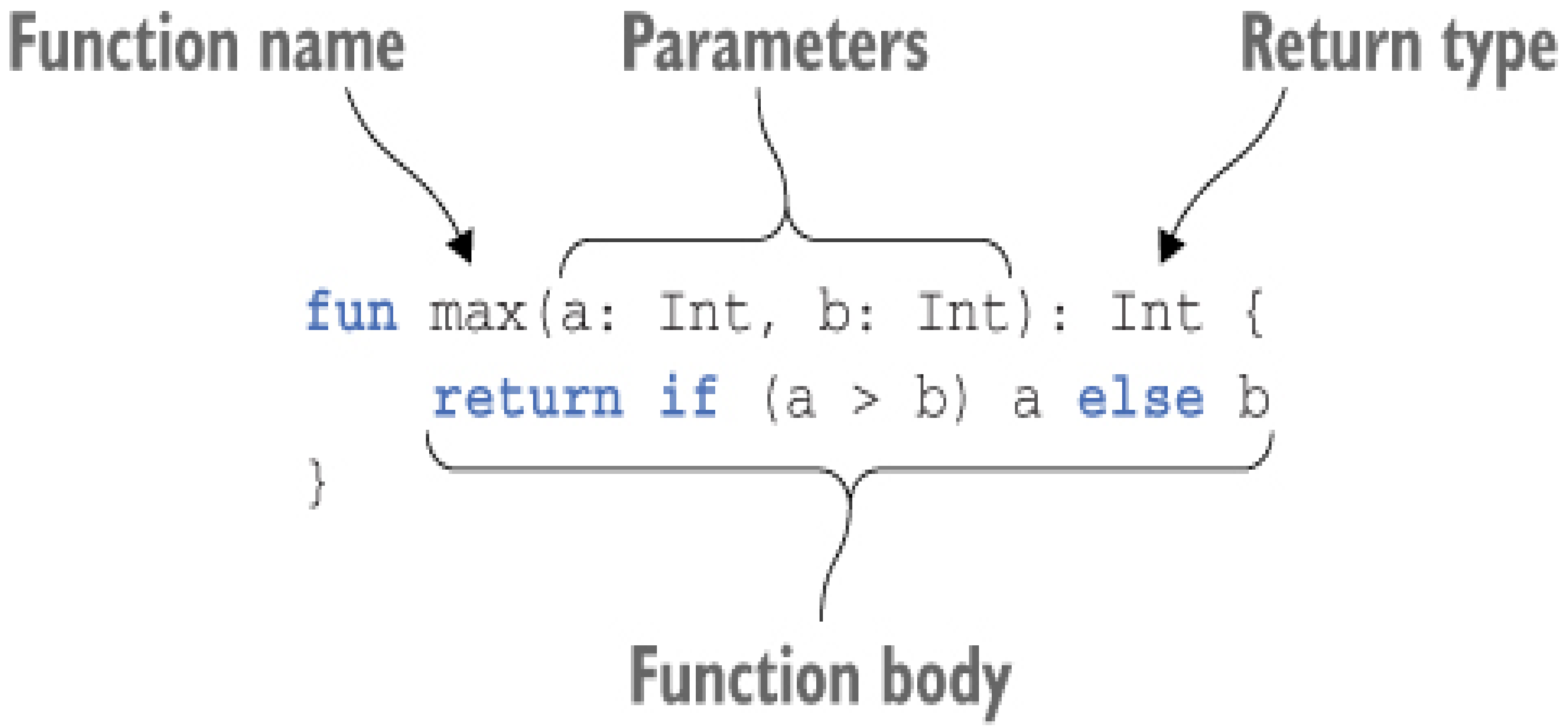
```
var myArray = arrayOf(1, 2, 3)
var arraySize = "myArray has ${myArray.size} items"
var firstItem = "The first item is ${myArray[0]} "
var myArray2 = arrayOf<Int>(1,10,4,6,15)
var myArray5: IntArray = intArrayOf(5,10,20,12,15)

  for (index in 0..4){
println(myArray5[index])
    }

val name = arrayOf<String>("Ajay","Prakesh","Michel","John","Sumit")

for(element in name){
println(element)
}
```

The provided classes are ByteArray, CharArray, ShortArray, IntArray, LongArray, BooleanArray, FloatArray, and DoubleArray.
For String : Array<String> or arrayOf("String1","String2");

ArrayList

```
val arrayList = ArrayList<String>()
    arrayList.add("Ajay")//Adding object in arraylist
    arrayList.add("Vijay")
    arrayList.add("Prakash")
    arrayList.add("Rohan")
    arrayList.add("Vijay")
    println(".......print ArrayList.......")
    for (i in arrayList) {
        println(i)
    }
```

Declaring Functions



```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}
```

Single Expression Functions

```
fun multiply(x: Int, y: Int): Int {
return x * y
}
```

Below is the same function expressed as a single line expression:

```
fun multiply(x: Int, y: Int): Int = x * y
```

When using single line expressions, the return type may be omitted in situations where the compiler is able to infer the type returned by the expression making for even more compact code:

```
fun multiply(x: Int, y: Int) = x * y
```

Example 1 - functions

```
fun main(args: Array<String>) {
    val name = "John"
    val count = 5
      fun displayString() {
            for (index in 0..count) {
            println(name)
       }
    }
    displayString()
}
```

Example 2- Variable Number of Function Parameters

Kotlin handles this possibility through the use of the *vararg* keyword to indicate that the function accepts an arbitrary number of parameters of a specified data type.

```
fun displayStrings(vararg strings: String){
        for (string in strings) {
                println(string)
        }
}
displayStrings("one", "two", "three", "four")
```

Example 3-Declaring Default Function Parameters

Kotlin provides the ability to designate a default parameter value to be used in the event that the value is not provided as an argument when the function is called.

```
fun buildMessageFor(name: String = "Customer", count: Int = 0): String {
        return("$name, you are customer number $count")
}

// Calling Functions

val message = buildMessageFor("John“,10) //Valid
val message = buildMessageFor("John") // Valid
```

If parameter names are not used within the function call, however, only the trailing arguments may be omitted:

```
val message = buildMessageFor(10) // Invalid
val message = buildMessageFor(count = 10) // Valid
```

Refer : Functions.kt

Main Point 1

- Java is a powerful object oriented and functional programming language features that is easier to use, faster development and more secured. ***Science of Consciousness:*** *Transcendental Meditation is an easy and effortless technique to make a mind clear and more powerful so that we can make life easier.*
-

Kotlin Modifiers

- Classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers. There are four visibility modifiers in Kotlin: private, protected, internal and public.
- **public: default**
  - The public modifiers means that the declarations are visible everywhere.
  - In Kotlin the default visibility modifier is public
- **internal**
  - internal means that the declarations are visible inside a module. A module in kotlin is a set of Kotlin files compiled together.
- **private**
  - With private declarations are only visible in the class
- **protected**
  - Declarations are only visible in its class and in its sub classes

Declaring a Kotlin Class

- Kotlin provides extensive support for developing object-oriented applications.
- The basic syntax for a new class is as follows:

```
class NewClassName {
   // Properties
   // Methods
}
```

Class and Object [Example – 1]

- **Simple Java class Person**

```
public class Person {
 private String name;
  public Person(String name) {
        this.name = name;
 }
public String getName() {
return name;
}
public void setName() {
this.name = name;
}

}
```

- **Simple Kotlin class Person**

```
//If you have only one constructor

class Person(val name: String)

//If you have many constructor

class Person {
  var name:String
  var age:Int = 0
  constructor(name:String) {
    this.name = name
  }
  constructor(name:String, age:Int) {
    this.name = name
    this.age = age
  }
}
```

Note : No need of semicolon in the end of the line

Constructors [Example – 2]

- There could be only one **Primary constructor** for a class in Kotlin.
- The primary constructor comes right after the class name in the header part of the class.
- Constructors that are written inside the Body of Class are called **Secondary constructors**.
- Secondary Constructor should call primary constructor or other secondary constructors using **this** keyword.

Refer : Item.kt & TestItem.kt

```
fun main(args: Array<String>){
    var person_1 = Person(“David",25, "Teaching")
    person_1.printPersonDetails()
}

// Kotlin Primary Constructors
class Person constructor(var name: String, var age: Int){
    var profession: String = "Not Mentioned"

// Kotlin Secondary Constructors

    constructor (name: String, age: Int, profession: String):
this(name,age){
        this.profession = profession
    }

    fun printPersonDetails(){
        println("$name whose profession is $profession, is
$age years old.")
    }
}
```

Output : David whose profession is Teaching, is 25 years old.

Class and Object [Example – 3]

```
class BankAccount {
   var accountBalance: Double = 0.0
   var accountNumber: Int = 0
   var lastName: String = ""
constructor(number: Int, balance: Double) {
accountNumber = number
accountBalance = balance
}
constructor(number: Int, balance: Double,name: String ) {
accountNumber = number
accountBalance = balance
lastName = name
}
fun displayBalance() {
println("Number $accountNumber")
println("Current balance is $accountBalance")
}
}
```

Object Creation

```
val account1: BankAccount = BankAccount(456456234, 342.98, "Smith")
```

Custom Accessors

- Accessors that are provided automatically by Kotlin.
- In addition to these default accessors it is also possible to implement *custom accessors* that allow calculations or other logic to be performed before the property is returned or set.
- Example

```
class BankAccount (val accountNumber: Int, var accountBalance: Double) {
val fees: Double = 25.00
var balanceLessFees: Double
get() {
return accountBalance - fees
}
set(value) {
accountBalance = value - fees
}
..
}
```

The following code gets the current balance less the fees value before setting the property to a new value:

```
val balance1 = account1.balanceLessFees
account1.balanceLessFees = 12123.12
```

Refer : Account.kt and TestAccount.kt

Kotlin Inheritance and Subclassing

Subclassing Syntax

- As a safety measure designed to make Kotlin code less prone to error, before a subclass can be derived from a parent class, the parent class must be declared as open. This is achieved by placing the *open* keyword within the class header:

```
open class MyParentClass {
var myProperty: Int = 0
}
```

With a simple class of this type, the subclass can be created as follows:

```
class MySubClass : MyParentClass() {
}
```

Deal with Parent class primary constructors in subclass

- For classes containing primary or secondary constructors, the rules for creating a subclass are slightly more complicated.
- Consider the following parent class which contains a primary constructor:

```
open class MyParentClass(var myProperty: Int)  {}
```

- In order to create a subclass of this class, the subclass declaration references any base class parameters while also initializing the parent class using the following syntax:

class MySubClass(**myProperty: Int**) : MyParentClass(**myProperty**) {
}

Example – BankAccount and SavingAccount

- Refer same BankAccount class and use open keyword to achieve inheritance and if you want to override a method, declare the method as open.

```
open class BankAccount {
    var accountNumber = 0
    var accountBalance = 0.0
    constructor(number: Int, balance: Double) {
        accountNumber = number
        accountBalance = balance
    }
     open fun displayBalance(){
        println("Number $accountNumber")
        println("Current balance is $accountBalance")
    }
}
```

Example – BankAccount and SavingAccount

```
class SavingsAccount : BankAccount {
    var interestRate: Double = 0.0
    constructor(accountNumber: Int, accountBalance: Double) :
            super(accountNumber, accountBalance)

    constructor(accountNumber: Int, accountBalance: Double, rate: Double) :
            super(accountNumber, accountBalance) {
        interestRate = rate
    }

    fun calculateInterest(): Double
    {
        return interestRate * accountBalance
    }
    override fun displayBalance()
    {
        println("Number $accountNumber")
        println("Current balance is $accountBalance")
        println("Prevailing interest rate is $interestRate")
    }
}
```

main function

```
fun main(args: Array<String>) {
    val savings1 = SavingsAccount(12311, 600.00, 0.07)
    println(savings1.calculateInterest())
    savings1.displayBalance()
}
```

Console output

```
42.01
Number 12311
Current balance is 600.0
Prevailing interest rate is 0.07
```

Refer : inheritancedemo Package

Interface

- An interface is essentially a contract that a class may choose to sign; if it does, the class is obliged to provide implementations of the properties and functions of the interface.
- However, an interface may provide a default implementation of some or all of its properties and functions.
- If a property or function has a default implementation, the class may choose to override it, but it doesn't have to.

Interface

Example

```
interface MyInterface {
    val test: Int
    // Abstract method
    fun print() : String

    // default implementation
    fun hello(name: String) {
        println("Hello there, $name!")
    }
}

class InterfaceImp : MyInterface {
    override val test: Int = 25
    override fun print() = "Kotlin"
}
```

```
fun main(args: Array<String>) {
    val obj = InterfaceImp()

    println("test = ${obj.test}")
    print("Calling hello(): ")

    obj.hello("Tim")
    print("Calling and printing
print(): ")
    println(obj.print())
}
```

Data Class

- While building any application, we often need to create classes whose primary purpose is to hold data/state.
- These classes generally contain the same old boilerplate code in the form of getters, setters, equals(), hashcode() and toString() methods.
- Will look into Kotlin data classes and how concise they are compared to POJOs (Plain Old Java Objects) and how much boilerplate we can get rid of by moving to Kotlin data classes during the Android app development.
- In Kotlin, this is called a data class and is marked as data

```
data class Person (val fname:String, val lname:String, var age:Int){
}
```

Rules for Creating Data Classes

- The Kotlin documentation on data classes notes that there are some basic restrictions in order to maintain consistency/behavior of generated code:
  - The primary constructor needs to have at least one parameter;
  - All primary constructor parameters need to be marked as val or var;
  - Data classes cannot be abstract, open, sealed(abstract in java)or inner;
  - Data classes may not extend other classes (but may implement interfaces).
  - Refer : Person.kt and TestPerson.kt

Main Point 2

- In the OO paradigm of programming, execution of a program involves objects interacting with objects. Each object has a type, which is embodied in a Kotlin *class*. The Kotlin language specifies syntax rules for the coding of classes, and also for how objects are to be created based on their type (class). *By using more and more of the intelligence of Nature, we are able to successfully manage all complexity in life, and live a life of success, harmony and fulfillment.*