UNIT: 1 INTRODUCTION TO KOTLIN PROGRAMMING

CS-31 Mobile Application Development in Android using Kotlin

- **BASICS OF KOTLIN**
- **OPERATIONS**
- **PRIORITIES**
- **DECISION MAKING**
- **LOOP CONTROL**
- **DATA STRUCTURES(COLLECTIONS)**
- **FUNCTIONS**
- **OBJECT ORIENTED PROGRAMMING**
- **INHERITANCE**
- **ABSTRACT**
- **INTERFACE**
- **SUPER AND THIS**
- **VISIBILITY MODIFIERS**

-:ASSIGNMENT:-

1. **JVM STANDS FOR**
2. **__IS THE FILE EXTENSION FOR SOUI'CEFILE IN KOTLIN**
3. **ENTRY PEINT TO A KOTLIN PROGRAM IS THE FUNCTION __**
4. **MUTABLE VARIABLES ARE DECLARED USING __ KEYWORD**
5. **EXPLAIN EXPRESSION & STATEMENT IN KOTLIN**
6. **EXPLAIN ARRAY IN KOTLIN**
7. **EXPLAIN TYPES OF USER DEFINED FUNCTIONS WITH EXAMPLE**
8. **WRITE A NOTE ON ABSTRACT CLASSES AND INTERFACES IN KOTLIN**
9. **WRITE A NOTE ON COLLECTIONS.**
10. **EXPLAIN TYPES OF LOOP WITH EXAMPLE**
11. **EXPLAIN TYPES INHIRENTENCE WITH SUITABLE**
12. **EXPLAIN VISIBLITY MODIFIERS IN KOTLIN**

PREPARED BY: PROF. N.K.PANDYA (PHD*, MCA, BCA, BPP)
KAMANI SCIENCE COLLEGE, AMRELI(BCA/BSC)

Basics/History of Kotlin

Kotlin is a programming language introduced by JetBrains in 2011, the official designer of the most intelligent Java IDE, named Intellij IDEA. This is a strongly statically typed general-purpose programming language that runs on JVM. In 2017, Google announced Kotlin is an official language for android development. Kotlin is an open source programming language that combines object-oriented programming and functional features into a unique platform.

Kotlin is a new open source programming language like Java, JavaScript, Python etc. It is a high level strongly statically typed language that combines functional and technical part in a same place. Currently, Kotlin mainly targets the Java Virtual Machine (JVM), but also compiles to JavaScript.

Kotlin is influenced by other popular programming languages such as Java, C#, JavaScript, Scala and Groovy. The syntax of Kotlin may not be exactly similar to Java Programming Language, however, internally Kotlin is reliant on the existing Java Class library to produce wonderful results for the programmers. Kotlin provides interoperability, code safety, and clarity to the developers around the world.

Kotlin was developed and released by JetBrains in 2016. Kotlin is free, has been free and will remain free. It is developed under the Apache 2.0 license and the source code is available on GitHub.

Why Kotlin?

- Cross-platform Mobile applications.
- Android Application Development.
- Web Application Development
- Server Side Applications
- Desktop Application Development
- Data science based applications
- Kotlin works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.) and it's 100% compatible with Java.
- Kotlin is used by many large companies like Google, Netflix, Slack, Uber etc to develop their Android based applications.
- The most importantly, there are many companies actively looking for Kotlin developers, especially in the Android development space.

Kotlin Advantages

- **Maximize the productivity**
  Kotlin is a programming language that is based on Java. Kotlin can easily get rid of aggravations as well as obsolescence of Java. Kotlin is a clear compact and dynamic language. Kotlin can maximize the productivity of the developer's team as it takes very little time to write and also you can deploy it pretty fast.
- **Works with existing Java Code**
  Kotlin is interoperable with Java codes. Kotlin is persistent with Java and so many other related frameworks and tools. So switching to Kotlin is much easy. If the product you are

building can't only be written in Kotlin, you can use the other one and both can be used together easily.

- **Can be easily maintained**
  Kotlin is supported by many IDEs which also include Android Studio and so many other SDK tools. By this, it also can maximize the productivity of the developer as it can continuously deal with toolkits they are already using.
- **Less Buggy**
  As we know Kotlin is a clear and compact codebase so it does not leave any space for making any mistakes, it also helps to provide more stable codes in production. The compiler will identify every possible mistake at a compile-time span sans any tumult.
- **Reliable**
  So many programming languages are available out there but Kotlin is more sophisticated language among them. Kotlin came on the big screen in the year of 2011. Till the time it has been introduced Kotlin has undergone many Beta and Alfa stages prior to releasing its final version.
- **Easy to learn**
  Kotlin aims to enhance the features of Java not just rewrite them. Mobile developers who have worked on Java they also can use Kotlin easily and their skill on Java will also stay relevant and helpful on Kotlin.
- **Merges functional and procedural programming**
  Nowadays so many programming languages are available in the market and everyone has their fair share of advantages and disadvantages. In order to get the best results Kotlin has combined the best of the functional as well as procedural programming.
- **Easy Language** − Kotlin supports object-oriented and functional constructs and very easy to learn. The syntax is pretty much similar to Java, hence for any Java programmer it is very easy to remember any Kotlin Syntax.
- **Very Concise** − Kotlin is based on Java Virtual Machine (JVM) and it is a functional language. Thus, it reduce lots of boiler plate code used in other programming languages.
- **Runtime and Performance** − Kotlin gives a better performance and small runtime for any application.
- **Interoperability** − Kotlin is mature enough to build an interoperable application in a less complex manner.
- **Brand New** − Kotlin is a brand new language that gives developers a fresh start. It is not a replacement of Java, though it is developed over JVM. Kotlin has been accepted as the first official language of Android Application Development. Kotlin can also be defined as - Kotlin = Java + Extra updated new features.

Kotlin Drawbacks

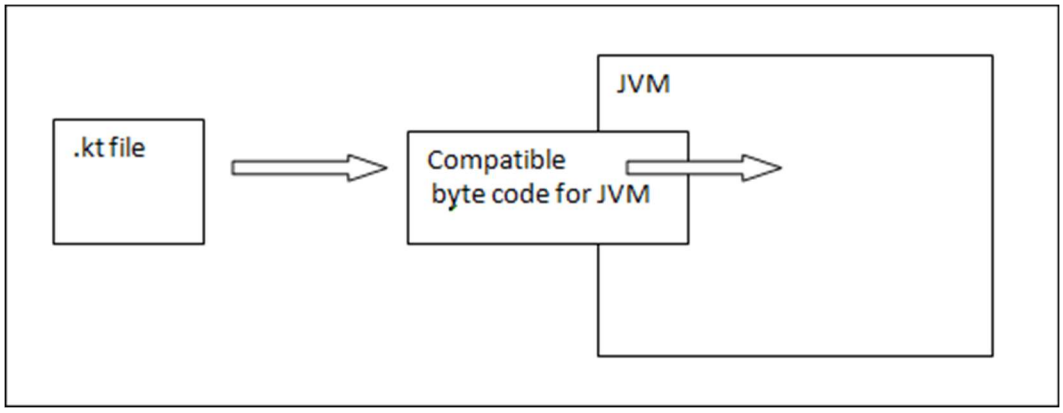
- **Namespace declaration** − Kotlin allows developers to declare the functions at the top level. However, whenever the same function is declared in many places of your application, then it is hard to understand which function is being called.
- **No Static Declaration** − Kotlin does not have usual static handling modifier like Java, which can cause some problem to the conventional Java developer.
- **Different from Java**
  Though Kotlin and Java have too many resemblances but still, there are some differences too. App developers just can't make the switch if they have inadequate knowledge about Kotlin.

- **Compilation Speed**
  In some cases, Kotlin works even faster than Java while performing incremental builds. But it should be remembered that Java will still stay incomparable when we talk about clean building.
- **Less Kotlin professionals**
  Kotlin has the sky-high popularity in developers community still only few programmers are available in this field. The fact about Kotlin is developers need to have in-depth knowledge about this language but nowadays it is really hard to find experienced experts in the domain of Kotlin.
- **Limited sources to learn**
  The number of developers who are switching to Kotlin is increasing but there are only limited developers community available to learn this language from or solve quarries in the process of development.

Kotlin Architecture

Kotlin is a programming language and has its own architecture to allocate memory and produce a quality output to the end user.
Following are the different scenarios where Kotlin compiler will work differently.

- Compile Kotlin into bytecode which can run on JVM. This bytecode is exactly equal to the byte code generated by the Java .class file.
- Whenever Kotlin targets JavaScript, the Kotlin compiler converts the .kt file into ES5.1 and generates a compatible code for JavaScript.
- Kotlin compiler is capable of creating platform basis compatible codes via LLVM.
- Kotlin Multiplatform Mobile (KMM) is used to create multiplatform mobile applications with code shared between Android and iOS.



Basic Syntax (Program Entry Point):

An entry point of a Kotlin application is the main function.

Type 1:

```
fun main() {
  println("Kamani Science College")
}
```

- The **fun** keyword is used to declare a function. A function is a block of code designed to perform a particular task.
- In the example above, it declares the **main()** function.

- The **main()** function is something you will see in every Kotlin program.
- This function is used to execute code.
- Any code inside the **main()** function's curly brackets **{}** will be executed.
- The **println()** function is inside the main() function, meaning that this will be executed.
- The **println()** function is used to output/print text, and in our example it will output "Kamani Science College".
- **Note**: **In Kotlin, code statements do not have to end with a semicolon (;)** (which is often required for other programming languages, such as Java, C++, C#, etc.).

Type 2:

Before Kotlin version 1.3, it was required to use the main() function with parameters, like: fun main(args : Array<String>).

The example above had to be written like this to work:

```
fun main(args : Array<String>) {
  println("Kamani Science College ")
}
```

- **Note:** This is no longer required, and the program will run fine without it. However, it will not do any harm if you have been using it in the past, and will continue to use it.

Comments

- A comment is a programmer-readable explanation or annotation in the Kotlin source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Kotlin compiler.
- Just like most modern languages, Kotlin supports single-line (or end-of-line) and multi-line (block) comments. Kotlin comments are very much similar to the comments available in Java, C and C++ programming languages.

Kotlin Single-line Comments

- Single line comments in Kotlin starts with two forward slashes // and end with end of the line. So any text written in between // and the end of the line is ignored by Kotlin compiler.

Kotlin Multi-line Comments

- A multi-line comment in Kotlin starts with /* and end with */. So any text written in between /* and */ will be treated as a comment and will be ignored by Kotlin compiler.
- Multi-line comments also called Block comments in Kotlin.

Kotlin Nested Comments

- Block comments in Kotlin can be nested, which means a single-line comment or multi-line comments can sit inside a multi-line comment

Variables/Priorities

- Variables are an important part of any programming. They are the names you give to computer memory locations which are used to store values in a computer program and later you use those names to retrieve the stored values and use them in your program.
- Kotlin variables are created using either **var or val** keywords and then an equal sign = is used to assign a value to those created variables.

Kotlin Mutable Variables

- Mutable means that the variable can be reassigned to a different value after initial assignment. To declare a mutable variable, we use the **var** keyword

Kotlin Read-only Variables

A read-only variable can be declared using **val** (instead of var) and once a value is assigned, **it can't be re-assigned**.

Read-only vs Mutable

- The Mutable variables will be used to define variables, which will keep charging their values based on different conditions during program execution.
- You will use Read-only variable to define different constant values i.e. the variables which will retain their value throughout of the program.

Kotlin Variable Naming Rules

- There are certain rules to be followed while naming the Kotlin variables.
- Kotlin variable names can contain letters, digits, underscores, and dollar signs.
- Kotlin variable names should start with a letter, $ or underscores
- Kotlin variables are case sensitive which means Zara and ZARA are two different variables.
- Kotlin variable can't have any white space or other control characters.
- Kotlin variable can't have names like var, val, String, Int because they are reserved keywords in Kotlin.

Kotlin - Data Types

- Kotlin data type is a classification of data which tells the compiler how the programmer intends to use the data. For example, Kotlin data could be numeric, string, boolean etc.
- Kotlin treats everything as an object which means that we can call member functions and properties on any variable.
- Kotlin is a statically typed language, which means that the data type of every expression should be known at compile time.

Kotlin built in data type can be categorized as follows:

- Number
- Character
- String
- Boolean
- Array

Kotlin Number Data Types

- Kotlin number data types are used to define variables which hold numeric values and they are divided into two groups:
  - Integer types store whole numbers, positive or negative.
  - Floating point types represent numbers with a fractional part, containing one or more decimals.
- **Example:** var a = 10;

Kotlin Character Data Type

- Kotlin character data type is used to store a single character and they are represented by the type Char keyword. A Char value must be surrounded by single quotes, like 'A' or '1'.
- **Example:** var letter = 'N'
- Kotlin supports a number of escape sequences of characters.
- When a character is preceded by a backslash (\), it is called an escape sequence and it has a special meaning to the compiler.
- For example, **\n** in the following statement is a valid character and it is called a new line character
- The following escape sequences are supported in Kotlin: \t, \b, \n, \r, \', \", \\ and \$.

Kotlin String Data Type

- The String data type is used to store a sequence of characters. String values must be surrounded by double quotes **(" ")** or triple quote **(""" """)**.
- We have two kinds of string available in Kotlin - one is called Escaped String and another is called Raw String.
- **Escaped string** is declared within double quote **(" ")** and may contain escape characters like '\n', '\t', '\b' etc.
- **Raw string** is declared within triple quote **(""" """)** and may contain multiple lines of text without any escape characters.
- **Example:**
  - var escapedString = "I am escaped String!\n"
  - var rawString = """This is going to be a
  - multi-line string and will
  - not have any escape sequence""";

Kotlin Boolean Data Type

- Boolean is very simple like other programming languages. We have only two values for Boolean data type - either true or false.
- Example:
  - val A = true // defining a variable with true value
  - val B = false // defining a variable with false value

Kotlin Array Data Type

- Kotlin arrays are a collection of homogeneous data.
- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- Example: var fruits = arrayOf("Apple", "Mango", "Banana", "Orange")

Kotlin Data Type Conversion

- Type conversion is a process in which the value of one data type is converted into another type. Kotlin does not support direct conversion of one numeric data type to another,
- **For example**, it is not possible to convert an Int type to a Long type.
- To convert a numeric data type to another type, Kotlin provides a set of functions:

toByte()	toShort()	toInt()	toLong()
toFloat()	toDouble()	toChar()	

Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- Kotlin is rich in built-in operators and provide the following types of operators:
  - Arithmetic Operators
  - Relational Operators
  - Assignment Operators
  - Unary Operators
  - Logical Operators
  - Bitwise Operations

Kotlin Arithmetic Operators

- Kotlin arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication and division etc.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y

-	Subtraction	Subtracts one value from another	x – y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y

Kotlin Relational Operators

- Kotlin relational (comparison) operators are used to compare two values, and returns a Boolean value: either true or false.

Operator	Name	Example
>	greater than	x > y
<	less than	x < y
>=	greater than or equal to	x >= y
<=	less than or equal to	x <= y
==	is equal to	x == y
!=	not equal to	x != y

Kotlin Assignment Operators

- Kotlin assignment operators are used to assign values to variables.
- Following is an example where we used assignment operator = to assign a values into two variables:

Operator	Example	Expanded Form
=	x = 10	x = 10
+=	x += 10	x = x - 10
-=	x -= 10	x = x - 10
*=	x *= 10	x = x * 10
/=	x /= 10	x = x / 10
%=	x %= 10	x = x % 10

Kotlin Unary Operators

- The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

Operator	Name	Example
+	unary plus	+x
-	unary minus	-x
++	increment by 1	++x
--	decrement by 1	--x
!	inverts the value of a boolean	!x

Kotlin Logical Operators

- Kotlin logical operators are used to determine the logic between two variables or values.

Operator	Name	Description	Example
&&	Logical and	Returns true if both operands are true	x && y
||	Logical or	Returns true if either of the operands is true	x || y
!	Logical not	Reverse the result, returns false if the operand is true	!x

Decision Making/Conditions

- Decision Making in programming is similar to decision making in real life. In programming too, a certain block of code needs to be executed when some condition is fulfilled. A programming language uses control statements to control the flow of execution of program based on certain conditions .If condition is true then it enters into the conditional block and executes the instructions.

There are different types of if-else expressions in Kotlin:

- if expression
- if-else expression
- if-else-if ladder expression
- nested if expression

if statement:

- It is used to specify a block of statements to be executed or not i.e if a certain condition is true then the statement or block of statements to be executed otherwise fails to execute.
- **Syntax:**

```
if(condition) {
    // code to run if condition is true
}
```

if-else statement:

- if-else statement contains two blocks of statements. 'if' statement is used to execute the block of code when the condition becomes true and 'else' statement is used to execute a block of code when the condition becomes false.
- **Syntax:**

```
if(condition) {
// code to run if condition is true
}
else {
// code to run if condition is false
}
```

Kotlin if-else expression as ternary operator:

- In Kotlin, if-else can be used as an expression because it returns a value. Unlike java, **there is no ternary operator in Kotlin because** if-else return the value according to the condition and works exactly similar to ternary.

if-else-if ladder expression:

- Here, a user can put multiple conditions. All the 'if' statements are executed from the top to bottom.
- One by one all the conditions are checked and if any of the condition found to be true then the code associated with the if statement will be executed and all other statements bypassed to the end of the block.
- If none of the conditions is true, then by default the final else statement will be executed.
- **Syntax:**

```
if(Firstcondition) {
// code to run if condition is true
}
else if(Secondcondition) {
// code to run if condition is true
}
```

```
else{
}
```

nested if expression:

- Nested if statements means an if statement inside another if statement.If first condition is true then code the associated block to be executed, and again check for the if condition nested in the first block and if it is also true then execute the code associated with it. It will go on until the last condition is true.
- **Syntax:**

```
if(condition1){
        // code 1`
        if(condition2){
        // code2
        }
}
```

when expression:

- when defines a conditional expression with multiple branches. It is similar to the switch statement in C-like languages.
- when matches its argument against all branches sequentially until some branch condition is satisfied.
- when can be used either as an expression or as a statement. If it is used as an expression, the value of the first matching branch becomes the value of the overall expression.
- If it is used as a statement, the values of individual branches are ignored. Just like with if, each branch can be a block, and its value is the value of the last expression in the block.
- The else branch is evaluated if none of the other branch conditions are satisfied.
- If when is used as an expression, the else branch is mandatory.
- **Syntax**:

```
when (x) {
   1 -> print("x == 1")
   2 -> print("x == 2")
   else -> {
      print("x is neither 1 nor 2")
   }
}
```

- **Example**:

```
fun main() {
    val a = 12
    val b = 5
    println("Enter operator either +, -, * or /")
    val operator = readLine()
    val result = when (operator) {
        "+" -> a + b
        "-" -> a - b
        "*" -> a * b
        "/" -> a / b
        else -> "$operator operator is invalid operator."
    }
    println("result = $result")
}
```

Loop Control

- Like any programming language, in kotlin also, the loops concept is used. Loops are mainly used to repeat the specific task. For example, if we want to print the number from 1 to 10, instead of writing a print statement 10 times, we can implement it through loops. Loops reduce the number of lines in the program and improve reliability.
- Loops are control flow structures that controls the flow of the program within loops. there are three types of loops in kotlin.
- **for loop**
- **while loop**
- **do-while loop**

for loop:

- Kotlin for loop is used to iterate the program several times. It iterates through ranges, arrays, collections, or anything that provides for iterate.
- **Syntax:**

```
for(Item in collection)
{
        // code
}
```

- **for** the keyword is used to implement for loop in kotlin.
- **Item** is the name of the variable which is used to store the data or information.
- **in** is a keyword that is used to check the items in the collection.
- **collection** can be anything **numbers, arrays, list**, etc.
- for loops first check the condition, if the condition matches, it will execute the loop else will stop the loop and execute the other statements.
- **Example:**

```
fun main() {
  val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
  for (x in cars) {
    println(x)
  }
}
```

while loop:

- Kotlin while loop is used to iterate the program serval times. It executes the block of the code until the result of the condition is true.
- **Syntax:**

```
while(condition)
{
        //code
}
```

- **while** keyword is used to implement the while loop in kotlin.
- **condition** is used to validate the requirement of the program, based on the outcome, the condition is mentioned. `
- **while** loop first checks the condition, if the condition is true, it will execute the block of the while. It repeat the same untill the condition is true.
- Once the condition becomes false, it will transfer the flow of the program outside the loop.
- **Example:**

```
fun main() {
  var i = 0
  while (i < 5) {
```

```
    println(i)
    i++
  }
}
```

do while loop:

- Kotlin do-while loop is the same as while loop, only one difference is there. while loop first checks the condition and then executes the block of the code, whereas do-while loop, first execute the block of the code then checks the condition.
- **Syntax:**

```
do{
        //code
}while(condition);
```

- **do** keyword is used to implement the do-while loop in kotlin.
- **do** first execute the block of the code and then check the condition. If the condition is true, it will repeat the execution of the loop else stop the execution of the loop.
- **while** keyword is used to specify the condition which is ended by using **; semicolon**.
- **Example:**

```
fun main() {
  var i = 0;
  do {
    println(i)
    i++
  }
  while (i < 5)
}
```

Kotlin repeat():

- Kotlin repeat statement is used when a set of statements has to be executed N-number of times.
- **Syntax:**

```
Repeat(N){
}
```

- **Example:**

```
fun main(args: Array) {
   repeat(4) {
      println("Hello World!")
   }
}
```

Kotlin Ranges:

- Kotlin range is defined by its two endpoint values which are both included in the range.
- Kotlin ranges are created with **rangeTo()** function, or simply using **downTo** or **(. .)** operators.
- The main operation on ranges is contains, which is usually used in the form of **in** and **!in** operators.
- **Creating Ranges using rangeTo()**:To create a Kotlin range we call rangeTo() function on the range start value and provide the end value as an argument.
  - for (num in 1.rangeTo(4))
- **Creating Ranges using .. Operator:** The rangeTo() is often called in its operator form ..,
  - for (num in 1..4)
- **Creating Ranges using downTo() Operator:** If we want to define a backward range we can use the downTo operator.
  - for (num in 4 downTo 1)

- **Kotlin step() Function:** We can use step() function to define the distance between the values of the range.
    - for (num in 1..10 step 2)
- **Kotlin reversed function():** It is used to reverse the given range type. Instead of downTo() we can use reverse() function to print the range in descending order.
- **until() Function:** When we want to create a range that excludes the end element we can use until().
- **The last, first, step Elements:** If we need to find the first, the step or the last value of the range, there are functions that will return them to us.
    - print((1..9).first) // Print 1
    - print((1..9 step 2).step) // Print 2
    - print((3..9).reversed().last) // Print 3
- **Some predefined function in range:** There are some predefined function in Kotlin Range: min(), max(), sum(), average().
    - val r = 1..20
    - print(r.min()) // Print 1
    - print(r.max()) // Print 20
    - print(r.sum()) // Print 210
    - print(r.average()) // Print 10.5
    - print(r.count()) // Print 20
- **Filtering Ranges:** The filter() function will return a list of elements matching a given predicate.
    - val r = 1..10
    - val f = r.filter { it -> it % 2 == 0 } // Print [2, 4, 6, 8, 10]

Functions

- Kotlin is a statically typed language, hence, functions play a great role in it. We are pretty familiar with function, as we are using function throughout our examples in our last chapters. A function is a block of code which is written to perform a particular task. Functions are supported by all the modern programming languages and they are also known as methods or subroutines.
- At a broad level, a function takes some input which is called parameters, perform certain actions on these inputs and finally returns a value.
- **Kotlin Built-in Functions:** Kotlin provides a number of built-in functions, we have used a number of buil-in functions in our examples.
- For example **print()** and **println()** are the most commonly used built-in function which we use to print an output to the screen.
- **User-Defined Functions:** Kotlin allows us to create our own function using the keyword fun. A user defined function takes one or more parameters, perform an action and return the result of that action as a value.
- **Syntax:**

```
fun functionName(){
  // body of function
}
```

- **Function Parameters:** A user-defined function can take zero or more parameters. Parameters are options and can be used based on requirement. For example, our above defined function did not make use of any paramenter.
- **Syntax:**

```
fun printSum(a:Int, b:Int){
  println(a + b)
}
```

- **Return Values:** A kotlin function return a value based on requirement. Again it is very much optional to return a value.

- To return a value, use the return keyword, and specify the return type after the function's parentheses.

Data Structures(Collections)

- Collections are a common concept for most programming languages. A collection usually contains a number of objects of the same type and Objects in a collection are called elements or items.
- The Kotlin Standard Library provides a comprehensive set of tools for managing collections. The following collection types are relevant for Kotlin:
- **Kotlin List** - List is an ordered collection with access to elements by indices. Elements can occur more than once in a list.
- **Kotlin Set** - Set is a collection of unique elements which means a group of objects without repetitions.
- **Kotlin Map** - Map (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value.
- Kotlin Collection Types
- Kotlin provides the following types of collection:
    - **Collection or Immutable Collection (value can't be change)**
    - **Mutable Collection (value can be change)**

Kotlin List:

- Kotlin list is an ordered collection of items. A Kotlin list can be either mutable (mutableListOf) or read-only (listOf). The elements of list can be accessed using indices. Kotlin mutable or immutable lists can have duplicate elements.
- **Creating Kotlin Lists:** For list creation, use the standard library functions **listOf()** for read-only lists and **mutableListOf()** for mutable lists.
- **Note**: To prevent unwanted modifications, obtain read-only views of mutable lists by casting them to List.
- **Example**:

```
fun main() {
   val theList = listOf("one", "two", "three", "four")
   println(theList)
   // theList.add() // will get error
   val theMutableList = mutableListOf("one", "two", "three", "four")
   println(theMutableList)
   theMutableList.add(0, "zero") // will add new element at 0th index
   theMutableList.add("five") // will add new element at last index

}
```

Kotlin Set:

- Kotlin set is an unordered collection of items. A Kotlin set can be either mutable (mutableSetOf) or read-only (setOf). Kotlin mutable or immutable sets do not allow to have duplicate elements.
- **Creating Kotlin Sets:** For set creation, use the standard library functions **setOf()** for read-only sets and **mutableSetOf()** for mutable sets.
- **Note:** A read-only view of a mutable set can be obtained by casting it to Set.
- **Example:**

```
fun main() {
   val theSet = setOf("one", "two", "three", "four")
   println(theSet)
   //theSet.add // will get error
   val theMutableSet = mutableSetOf("one", "two", "three", "four")
   println(theMutableSet)
   theMutableSet.add("five") // will add new element at last
```

```
}
```

Kotlin – Maps

- Kotlin map is a collection of key/value pairs, where each key is unique, and it can only be associated with one value. The same value can be associated with multiple keys though. We can declare the keys and values to be any type; there are no restrictions.
- A Kotlin map can be either mutable (mutableMapOf) or read-only (mapOf).
- Maps are also known as **dictionaries or associative arrays** in other programming languages.
- **Creating Kotlin Maps:** For map creation, use the standard library functions **mapOf()** for read-only maps and **mutableMapOf()** for mutable maps.
- **Note:**A read-only view of a mutable map can be obtained by casting it to Map.
- **Example:**

```
fun main() {
   val theMap = mapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
   println(theMap)
   // theMap.add() // will get error
   val theMutableMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
   println(theMutableMap)
   theMutableMap.put("five", 5) // will add new element
}
```

Object Oriented Programming

- By definition of OOP, a class is a blueprint of a runtime entity and object is its state, which includes both its behavior and state. In Kotlin, class declaration consists of a class header and a class body surrounded by curly braces, similar to Java.

Syntax:

```
Class myClass { // class Header
  // class Body
}
```

- Like Java, Kotlin also allows to create several objects of a class and you are free to include its class members and functions. We can control the visibility of the class members variables using different keywords In the following example, we will create one class and its object through which we will access different data members of that class.

Example:

```
class myClass {
  // property (data member)
  private var name: String = "Kamani Science College"
  // member function
  fun printMe() {
    print("You are at the best College Named-"+name)
  }
}
fun main() {
  val obj = myClass() // create obj object of myClass class
  obj.printMe()
}
```

Nested Class

- By definition, when a class has been created inside another class, then it is called as a nested class. In Kotlin, nested class is by default **static**, hence, it can be accessed without creating any object of that class. In the following example, we will see how Kotlin interprets our nested class.

Example:

```
fun main() {
  val demo = Outer.Nested().foo() // calling nested class method
  print(demo)
}
class Outer {
  class Nested {
    fun foo() = "Welcome to The Kamani Science College"
  }
}
```

Inner Class

- When a nested class is marked as a **"inner"**, then it will be called as an Inner class. An inner class can be accessed by the data member of the outer class. In the following example, we will be accessing the data member of the outer class.

Example:

```
fun main() {
  val demo = Outer().Nested().foo() // calling nested class method
  print(demo)
}
class Outer {
  private val welcomeMessage: String = "Welcome to The Kamani Science College"
  inner class Nested {
    fun foo() = welcomeMessage
  }
}
```

Anonymous Inner Class/object Expression

- Anonymous inner class is a pretty good concept that makes the life of a programmer very easy. Whenever we are implementing an interface, the concept of anonymous inner block comes into picture. The concept of creating an object of interface using runtime object reference is known as anonymous class. In the following example, we will create an interface and we will create an object of that interface using Anonymous Inner class mechanism.

Example:

```
fun main() {
   var programmer  = object:Human  { // creating an instance of the interface
   override fun think() { // overriding the think method
      print("I am an example of Anonymous Inner Class ")
   }
}
programmer.think()
}
interface Human {
   fun think()
}
```

Inheritance

- By definition, we all know that inheritance means accruing some properties of the mother class into the child class. In Kotlin, the base class is named as "Any", which is the **super class** of the 'any' default class declared in Kotlin. Like all other OOPS, Kotlin also provides this functionality using one keyword known as ":".

- Everything in Kotlin is by default final, hence, we need to use the keyword "**open**" in front of the class declaration to make it allowable to inherit. Take a look at the following example of inheritance.
- he concept of inheritance is allowed when two or more classes have same properties. It allows code reusability. A derived class has only one base class but may have multiple interfaces whereas a base class may have one or more derived classes.
- In Kotlin, the derived class inherits a base class using ":" operator in the class header (**after the derive class name or constructor**)
- As Kotlin classes are **final by default**, they cannot be **inherited simply**. We use the **open** keyword before the class to inherit a class and make it to non-final.

Example:

```
open class ParentClass {
   fun think () {
      print("Parent Class is thinking")
   }
}
class ChildClass: ParentClass(){
}
fun main() {
   var  a = ChildClass ()
   a.think()
}
```

- Now, what if we want to **override** the think() method in the child class. Then, we need to consider the following example where we are creating two classes and override one of its function into the child class. **Like Java, Kotlin too doesn't allow multiple inheritance.**

Example:

```
open class ParentClass {
   open fun think () {
      print("Parent Class is thinking")
   }
}
class ChildClass: ParentClass(){
   override fun think () {
      print("Child Class is thinking")
   }
}
fun main() {
   val  a = ChildClass ()
   a.think()
}
```

Abstract

- Like Java, abstract keyword is used to declare abstract classes in Kotlin. An abstract class cannot be instantiated (**you cannot create objects of an abstract class**). However, you can inherit subclasses from can them.
- Abstract classes are **partially defined classes**, methods and properties which have no implementation but **must be implemented into derived class**. If the derived class does not implement the properties of base class **then is also meant to be an abstract class**.
- Abstract class or abstract function does not need to annotate with **open** keyword as they are open by default. Abstract member function does not contain its body.
- **Points to remember:**

- We can't create an object for abstract class.
- If we declare a member function as abstract then we does not need to annotate with **open** keyword because these are open by default.
- **An abstract member function doesn't have a body, and it must be implemented in the derived class.**

Example:

```
abstract class Person {
   var age: Int = 40
   fun displaySSN(ssn: Int) {
      println("My SSN is $ssn.")
   }
   abstract fun displayJob(description: String)
}
```

- an abstract class **Person** is created. You cannot create objects of the class.
- the class has a non-abstract property age and a non-abstract method **displaySSN()**.
- If you need to override these members in the subclass, they should be marked with open keyword.
- **The class has an abstract method displayJob(). It doesn't have any implementation and must be overridden in its subclasses.**

Note: **Abstract classes are always open. You do not need to explicitly use open keyword to inherit subclasses from them.**

Example:

```
abstract class Person() {
   fun displaySSN(ssn: Int) {
      println("My SSN is $ssn.")
   }
   abstract fun displayJob(description: String)
}
class Teacher: Person() {
   override fun displayJob(description: String) {
      println(description)
   }
   fun displayName(name: String) {
      println("My name is $name.")
   }
}
fun main() {
   val nkp = Teacher()
   nkp.displayName("NKP")
   nkp.displayJob("I work at KSC.")
   nkp.displaySSN(123)
}
```

Kotlin – Constructors

- Kotlin has two types of constructor - one is the **primary constructor** and the other is the **secondary constructor**. One Kotlin class can have one primary constructor, and one or more secondary constructor.
- A constructor is a special member function that is invoked when an object of the class is created primarily to initialize variables or properties. A class needs to have a constructor and if we do not declare a constructor, **then the compiler generates a default constructor**.
- The primary constructor can be declared **at class header level** as shown in the following example.

- The primary constructor cannot contain any code, the initialization code can be placed in a separate initializer block prefixed with the **init** keyword.

Syntax:

```
class Person(val firstName: String, var age: Int) {
  // class body
    init {
      //some functionality
    }
    constructor(list of parameters):this(parameters of primary constructor) {
      //some functionality
    }
}
```

- In the above example, we have declared the primary constructor inside the parenthesis. Among the two fields, first name is read-only as it is declared as "val", while the field age can be edited. In the following example, we will use the primary constructor.

Example:

```
fun main() {
   val person1 = Person("KSC", 15)
   println("First Name = "+ person1.firstName)
   println("Age ="+ person1.age)
}
class Person(val firstName: String, var age: Int) {
}
```

- As mentioned earlier, Kotlin allows to create one or more secondary constructors for your class.
- This secondary constructor is created using the "**constructor**" keyword.
- It is required whenever you want to create more than one constructor in Kotlin or whenever you want to include more logic in the primary constructor and you cannot do that because the primary constructor may be called by some other class.
- Take a look at the following example, where we have created a secondary constructor and are using the above example to implement the same.

Example:

```
fun main() {
   val Human = Human("KSC", 25)
   println(Human.message+Human.firstName+
         " Welcome to the example of Secondary  constructor, Your Age is-"+Human.age)
}
class Human(val firstName: String, var age: Int) {
   var message:String  = "Hey!!!"
   constructor(name : String , age :Int ,message :String):this(name,age) {
      this.message = message
   }
   init {
      println("I am being called when object is created.")
   }
}
```

Note - Any number of secondary constructors can be created, however, all of those constructors should call the primary constructor directly or indirectly.

Interface

- In Kotlin, the interface works exactly similar to Java, which means they can contain method implementation as well as abstract methods declaration. An interface can be implemented by a class in order to use its defined functionality.

Example:

```
interface ExampleInterface {
  var myVar: String     // abstract property
  fun absMethod()       // abstract method
  fun sayHello() = "Hello there" // method with default implementation
}
```

- In the above example, we have created one interface named as “ExampleInterface” and inside that we have a couple of abstract properties and methods all together. Look at the function named “sayHello()”, which is an implemented method.

Example:

```
interface ExampleInterface  {
    var myVar: Int            // abstract property
    fun absMethod():String    // abstract method
    fun hello() {
        println("Hello there, Welcome to KSC!")
    }
}
class InterfaceImp : ExampleInterface {
    override var myVar: Int = 25
    override fun absMethod() = "Happy Learning "
}
fun main() {
    val obj = InterfaceImp()
    println("My Variable Value is = "+ obj.myVar)
    print("Calling hello(): ")
    obj.hello()
    print("Message from the KSC-- ")
    println(obj.absMethod())
}
```

- As mentioned earlier, **Kotlin doesn’t support multiple inheritances**, however, the same thing can be achieved by implementing more than two interfaces at a time.
- In the following example, we will create two interfaces and later we will implement both the interfaces into a class.

Example:

```
interface A {
   fun printMe()= "method of interface A"
}
interface B  {
   fun printMeToo() = "I am Method from interface B"
}
// implements two interfaces A and B
class MultipleInterfaceExample: A, B

fun main() {
   val obj = MultipleInterfaceExample()
```

```
    println(obj.printMe())
    println(obj.printMeToo())
}
```

Super And This

- We can also call the base class member functions or properties from the derived class using the super keyword. In the below program we have called the base class property color and function displayCompany() in the derived class using the super keyword.

Example:

```
// base class
open class Phone() {
    var color = "Rose Gold"
    fun displayCompany(name:String) {
            println("Company is: $name")
    }
}
// derived class
class iphone: Phone() {
    fun displayColor(){
            // calling the base class property color
            println("Color is: "+ super.color)
            // calling the base class member function
            super.displayCompany("Apple")
    }
}
fun main() {
    val p = iphone()
    p.displayColor()
}
```

- In Kotlin, the "**this**" keyword allows us to refer to the instance of a class whose function we happen to be running. Additionally, there are other ways in which "this" expressions come in handy.
- Using **this** as a prefix, we can refer to the properties of the class. We can use this to resolve ambiguity with similarly named local variables. Likewise, we can also call upon member functions using this.

Example:

```
class Counter {
   var count = 0
   fun incrementCount() {
      this.count += 2
   }
}

fun main() {
   var c: Counter = Counter()
    var count = 50
   c.incrementCount()
   println("count in main " + count)
   println("count in counter " + c.count)
}
```

Visibility Modifiers/Visibility Control

Modifier	Description
public	visible everywhere
private	visible inside the same class only
internal	visible inside the same module
protected	visible inside the same class and its subclasses

Access modifier is used to restrict the usage of the variables, methods and class used in the application. Like other OOP programming language, this modifier is applicable at multiple places such as in the class header or method declaration. There are four access modifiers available in Kotlin.

Public

- Public modifier is accessible from anywhere in the project workspace. If no access modifier is specified, then by **default it will be in the public scope**. In all our previous examples, we have not mentioned any modifier, hence, all of them are in the public scope. Following is an example to understand more on how to declare a public variable or method.

```
class publicExample {
  val i = 1
  fun doSomething() {
  }
}
```

- In the above example, we have not mentioned any modifier, thus all these methods and variables are by default public.

Private

- The classes, methods, and packages can be declared with a private modifier. Once anything is declared as private, then it will be accessible within its **immediate scope**. For instance, a private package can be accessible within that specific file. A private class or interface can be accessible only by its data members, etc.

```
private class privateExample {
  private val i = 1
  private val doSomething() {
  }
}
```

- In the above example, the class “privateExample” and the variable i both can be accessible only in the same Kotlin class, where its mentioned as they all are declared as private in the declaration block.

Example:

```
private class A {
   private val int = 10
   fun display()
   {
      println(int)  // we can access int in the same class
      println("Accessing int successful")
   }
}
fun main(){
   var a = A()
   a.display()// can not access 'int': it is private in class A
   println(a.int)
}
```

Protected

Protected is another access modifier for Kotlin, which is currently not available for top level declaration like any package cannot be protected. A protected class or interface is visible to its subclass only.

```
class A() {
  protected val i = 1
}
class B : A() {
  fun getValue() : Int {
    return i
  }
}
```

In the above example, the variable "i" is declared as protected, hence, it is only visible to its subclass.

Example:

```
open class A {
    // protected variable
   protected val int = 10
}
class B: A() { // derived class
   fun getvalue(): Int {
      return int // accessed from the subclass
   }
}
fun main() {
   var a = B()
   println("The value of integer is: "+a.getvalue())
}
```

Internal

- Internal is a newly added modifier introduced in Kotlin. If anything is marked as internal, then that specific field will be in the internal field. An Internal package is visible only inside the module under which it is implemented. An internal class interface is visible only by other class present inside the same package or the module. In the following example, we will see how to implement an internal method.
- In Kotlin, the internal modifier is a newly added modifier that is not supported by Java. Marked as internal means that it will be available in the same module, if we try to access the declaration from another module it will give an error. A module means a group of files that are compiled together.
- **Note**: **Internal modifier benefits in writing APIs and implementations.**

```
internal class A {
}
public class B {
   internal val int = 10
   internal fun display() {
   }
}
```

Here, Class A is only accessible from inside the same module. The variable int and function display() are only accessible from inside the same module, even though class B can be accessed from anywhere.