Introduction to Kotlin

- Kotlin is a programming language that can run on JVM.
- Google has announced Kotlin as one of its officially supported programming languages in Android Studio; and the Android community is migrating at a pace from Java to Kotlin.

Simplest Version

```
fun main() {
   println("Hello World")
}
```

Object Oriented Hello

```
class Greeter(val name: String) {
   fun greet() {
      println("Hello, $name")
   }
}

fun main(args: Array<String>) {
   Greeter("Atmiya").greet()
}
```

Output: Hello, Atmiya

1. Kotlin code is usually defined in packages. Package specification is optional: If you don't specify a package in a source file, its content goes to the default package.
2. An entry point to a Kotlin application is the main function. In Kotlin 1.3, you can declare main without any parameters. The return type is not specified, which means that the function returns nothing.
3. println writes a line to the standard output. It is imported implicitly. Also note that semicolons are optional.

In Kotlin versions earlier than 1.3, the main function must have a parameter of type Array<String>

```
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

Variables

- Kotlin has powerful type inference.
- While you can explicitly declare the type of a variable, you'll usually let the compiler do the work by inferring it.
- Kotlin does not enforce immutability, though it is recommended. In essence use *val* over *var*.

```
1   var a: String = "initial"  // 1
2   val b: Int = 1             // 2
3   val c = 3                  // 3
4
5   fun main(){
6       println(a)
7   }
```

1. Declares a mutable variable and initializing it.
2. Declares an immutable variable and initializing it.
3. Declares an immutable variable and initializing it without specifying the type. The compiler infers the type Int.

```
var e: Int  // 1
println(e)  // 2
```

1. Declares a variable without initialization.
2. An attempt to use the variable causes a compiler error: Variable 'e' must be initialized.

You're free to choose when you initialize a variable, however, it must be initialized before the first read.

Functions

Default Parameter Values and Named Arguments

```
fun printMessage(message: String): Unit {                          // 1
    println(message)
}

fun printMessageWithPrefix(message: String, prefix: String = "Info") {  // 2
    println("[$prefix] $message")
}

fun sum(x: Int, y: Int): Int {                                     // 3
    return x + y
}

fun multiply(x: Int, y: Int) = x * y                               // 4

fun main() {
    printMessage("Hello")                                          // 5
    printMessageWithPrefix( message: "Hello",  prefix: "Log")      // 6
    printMessageWithPrefix( message: "Hello")                      // 7
    printMessageWithPrefix(prefix = "Log", message = "Hello")      // 8
    println(sum( x: 1,  y: 2))                                     // 9
}
```

```
Output:
Hello
[Log] Hello
[Info] Hello
[Log] Hello
3
```

1. A simple function that takes a parameter of type String and returns Unit (i.e., no return value).
2. A function that takes a second optional parameter with default value Info. The return type is omitted, meaning that it's actually Unit.
3. A function that returns an integer.
4. A single-expression function that returns an integer (inferred).
5. Calls the first function with the argument Hello.

6. Calls the function with two parameters, passing values for both of them.
7. Calls the same function omitting the second one. The default value Info is used.
8. Calls the same function using named arguments and changing the order of the arguments.
9. Prints the result of a function call.

Functions with vararg Parameters

Varargs allow you to pass any number of arguments by separating them with commas.

```
fun printAll(vararg messages: String) {
    for (m in messages) println(m)
}

fun printAllWithPrefix(vararg messages: String, prefix: String) {
    for (m in messages) println(prefix + m)
}

fun log(vararg entries: String) {
    printAll(*entries)
}

fun main(){
    printAll( ...messages: "Atmiya", "University", "MCA", "Android", "Kotlin")
    printAllWithPrefix( ...messages:
        "Atmiya", "University", "MCA", "Android", "Kotlin",
        prefix = "Greeting: "
    )
}
```

Output:
Atmiya
University
MCA
Android
Kotlin
Greeting: Atmiya
Greeting: University
Greeting: MCA
Greeting: Android
Greeting: Kotlin

Null Safety

In an effort to rid the world of NullPointerException, variable types in Kotlin don't allow the assignment of null. If you need a variable that can be null, declare it nullable by adding ? at the end of its type.

```
var neverNull: String = "This can't be null"                    // 1
neverNull = null                                                // 2
var nullable: String? = "You can keep a null here"              // 3
nullable = null                                                 // 4
var inferredNonNull = "The compiler assumes non-null"           // 5
inferredNonNull = null                                          // 6
fun strLength(notNull: String): Int {                           // 7
    return notNull.length
}
fun main(){
    strLength(neverNull)                                        // 8
    strLength(nullable)                                         // 9
}
```

1. Declares a non-null String variable.
2. When trying to assign null to non-nullable variable, a compilation error is produced.
3. Declares a nullable String variable.
4. Sets the null value to the nullable variable. This is OK.
5. When inferring types, the compiler assumes non-null for variables that are initialized with a value.
6. When trying to assign null to a variable with inferred type, a compilation error is produced.
7. Declares a function with a non-null string parameter.
8. Calls the function with a String (non-nullable) argument. This is OK.
9. When calling the function with a String? (nullable) argument, a compilation error is produced.

Working with Nulls

Sometimes Kotlin programs need to work with null values, such as when interacting with external Java code or representing a truly absent state. Kotlin provides null tracking to elegantly deal with such situations.

```
1   fun describeString(maybeString: String?): String {          // 1
2       if (maybeString != null && maybeString.length > 0) {    // 2
3           return "String of length ${maybeString.length}"
4       } else {
5           return "Empty or null string"                       // 3
6       }
7   }
8   fun main(){
9       println(describeString( maybeString: null))
10      println(describeString( maybeString: "Atmiya"))
11  }
```

Kotlin Control Flow

If Expression

- if executes a certain section of code if the testExpression is evaluated to true.
- It can have optional else clause.
- Codes inside else clause are executed if the testExpression is false.

```
fun myfun(){
        print("Enter Any Per : ")
        var per = readLine()!!.toDouble()

        if(per>=90 && per<=100)
                print("Excellent")
        else if(per>=75 && per<90)
                print("Very Good")
        else if(per>=60 && per<75)
                print("Good")
        else
                print("Fine")
}
```

//Same Code using Range

```
fun main(){
        print("Enter Any Per : ")
        var per = readLine()!!.toDouble()

        if(per in 90..100)
                print("Excellent")
        else if(per in 75..89)
                print("Very Good")
        else if(per in 60..74)
                print("Good")
        else
                print("Fine")
}
```

When Expression

- The when construct in Kotlin can be thought of as a replacement for Java switch Statement.
- It evaluates a section of code among many alternatives.

Example: Simple when Expression

```
fun main(args: Array<String>) {
    val a = 12
    val b = 5

    println("Enter operator ")
    val operator = readLine()

    val result = when (operator) {
        "+" -> a + b
        "-" -> a - b
        "*" -> a * b
        "/" -> a / b
        else -> "$operator is invalid"
    }

    println("result = $result")
}
```

When you run the program, the output will be something like:

```
Enter operator
*
result = 60
```

Check value in the range. For example,

```
var i = 67
when(i){
   0 -> print("Hello")
   in (1..50) -> print("Hi")
   in(51..100) -> print("Kotlin Rocks")
   else -> print("Testing")
}
```

Check if a value is of a particular type.

- To check whether a value is of a particular type in runtime, we can use is and !is operator. For example,

```
when (x) {

    is Int -> print(x + 1)

    is String -> print(x.length + 1)

    is IntArray -> print(x.sum())

}
```

Combine two or more branch conditions with a comma. For example,

```
fun main(args: Array<String>) {
    val n = -1
    when (n) {
 1, 2, 3 -> println("n is a positive < 4.")
       0 -> println("n is zero")
  -1, -2 -> println("n is a negative > 3.")
    }
}
```

Loops

- Loop is used in programming to repeat a specific block of code until certain condition is met (test expression is false).

Kotlin while Loop

- The syntax of while loop is:

```
while (testExpression) {

    // codes inside body of while loop

}
```

Examples of Loops

```
for (i in 1..10)
   print("$ ")
for (i in 1..10)
   print("$i ")
1 2 3 4 5 6 7 8 9 10
================================
for (i in 1..10 step 2)
   print("$i ")
1 3 5 7 9
================================
for(i in 10 downTo 1)
   print("$i ")
10 9 8 7 6 5 4 3 2 1
================================
for (i in 1..10)
   print("$i ")
1 2 3 4 5 6 7 8 9 10
================================
var i = 1
while(i<=10){
   print("$i ")
   i++
}
1 2 3 4 5 6 7 8 9 10
================================
var i = 1
while(i<=10){
   print("$i ")
   i+=2
}
1 3 5 7 9
================================
print("Enter No1 : ")
var no1 = readLine()!!.toInt()
print("Enter No2 : ")
var no2 = readLine()!!.toInt()
var ch = -1
do{
        print("1.Addition\n")
        print("2.Subtraction\n");
        print("Enter Your Choice")
        ch = readLine()!!.toInt()
        when(ch){
 1 -> print("\nSum = ${no1+no2}\n")
 2 -> print("\nSubtraction = ${no1-no2}\n")
 0 -> print("\nByye...\n")
          }
}while(ch!=0)
```

Arrays

- Array is one of the most fundamental data structure in practically all programming languages.
- The idea behind an array is to store multiple items of the same data-type, such as an integer or string under a single variable name.
- Arrays are used to organize data in programming so that a related set of values can be easily sorted or searched.

Examples

Following code will accept n no from user and then produce the average

```
print("Enter how many nos you want : ")
var size = readLine()!!.toInt()
var arr:Array<Int> = Array(size)

for(i in 0 until size){
   println("Enter No ${i+1}")
   arr[i] = readLine()!!.toInt()
}
var sum = 0
for(i in 0 until size){
   sum += arr[i]
}
print("\nAverage = ${sum/size}")
```

Following code will accept 5 strings and display the same with length count

```
var names:Array<String> = Array(5){""}
for(i in 0 until 5){
   println("Enter Name : ")
   names[i] = readLine()!!.toString()
}
for (i in 0 until 5){
   println("Name = ${names[i]}, Length = ${names[i].length}")
}
```

Following code will accept 5 string from user and display only those string which name ends with the .com.

```
var email:Array<String> = Array(5){""}
for(i in 0 until 5){
   println("Enter Email : ")
   email[i] = readLine()!!.toString()
}
for (i in 0 until 5){
   if(email[i]!!.endsWith(".com")) {
        println("Email = ${email[i]}, Length = ${email[i].length}\n")
     }
}
```

Classes

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor etc.) and the class body, surrounded by curly braces. Both the header and the body are optional; if the class has no body, curly braces can be omitted.

```
1   class Customer                                          //1
2
3   class Contact(val id : Int, var name : String)          //2
4
5   fun main(){
6
7       val cust1 = Customer()                              //3
8       var contact = Contact( id: 10, name: "Atmiya")      //4
9
10      println(contact.id)                                 //5
11      println(contact.name)                               //5
12      contact.name = "Android"                            //6
13      println(contact.name)
14  }
```

1. Declares a class named Customer without any properties or user-defined constructors. A non-parameterized default constructor is created by Kotlin automatically.
2. Declares a class with two properties: immutable id and mutable email, and a constructor with two parameters id and email.
3. Creates an instance of the class Customer via the default constructor. Note that there is no new keyword in Kotlin.
4. Creates an instance of the class Contact using the constructor with two arguments.
5. Accesses the property id.
6. Updates the value of the property email.

Inheritance

Kotlin fully supports the traditional object-oriented inheritance mechanism.

```
open class Dog {                    // 1
    open fun sayHello() {           // 2
        println("wow wow!")
    }
}

class Yorkshire : Dog() {           // 3
    override fun sayHello() {       // 4
        println("wif wif!")
    }
}

fun main() {
    val dog: Dog = Yorkshire()
    dog.sayHello()
}
```

1. Kotlin classes are *final* by default. If you want to allow the class inheritance, mark the class with the open modifier.
2. Kotlin methods are also *final* by default. As with the classes, the open modifier allows overriding them.
3. A class inherits a superclass when you specify the : SuperclassName() after its name. The empty parentheses () indicate an invocation of the superclass default constructor.
4. Overriding methods or attributes requires the override modifier.

Inheritance with Parameterized Constructor

```
open class Tiger(val origin: String) {
    fun sayHello() {
        println("A tiger from $origin says: grrhhh!")
    }
}

class SiberianTiger : Tiger("Siberia")                    // 1

fun main() {
    val tiger: Tiger = SiberianTiger()
    tiger.sayHello()
}
```

1. If you want to use a parameterized constructor of the superclass when creating a subclass, provide the constructor arguments in the subclass declaration.

Passing Constructor Arguments to Superclass

```
open class Lion(val name: String, val origin: String) {
    fun sayHello() {
        println("$name, the lion from $origin says: graoh!")
    }
}

class Asiatic(name: String) : Lion(name = name, origin = "India") // 1

fun main() {
    val lion: Lion = Asiatic("Rufo")                              // 2
    lion.sayHello()
}
```

1. name in the Asiatic declaration is neither a var nor val: it's a constructor argument, whose value is passed to the name property of the superclass Lion.
2. Creates an Asiatic instance with the name Rufo. The call invokes the Lion constructor with arguments Rufo and India.

Visibility Modifiers Inside Package

- A package organizes a set of related functions, properties and classes, objects, and interfaces.

Modifier	Description
public	declarations are visible everywhere
private	visible inside the file containing the declaration
internal	visible inside the same module (a set of Kotlin files compiled together)
protected	not available for packages (used for subclasses)

Note: If visibility modifier is not specified, it is public by default.

Visibility Modifiers Inside Classes and Interfaces

- Here's how visibility modifiers work for members (functions, properties) declared inside a class:

Modifier	Description
public	visible to any client who can see the declaring class
private	visible inside the class only
protected	visible inside the class and its subclasses
internal	visible to any client inside the module that can see the declaring class

By default, the visibility of a constructor is public. However, you can change it. For that, you need to explicitly add constructor keyword.

Note: In Kotlin, local functions, variables and classes cannot have visibility modifiers.

abstract class

1. Like Java, abstract keyword is used to declare abstract classes in Kotlin.
2. An abstract class cannot be instantiated (you cannot create objects of an abstract class). However, you can inherit subclasses from them.
3. The members (properties and methods) of an abstract class are non-abstract unless you explicitly use abstract keyword to make them abstract. Let's take an example:

```
abstract class Person {

    var age: Int = 40

    fun displaySSN(ssn: Int) {

        println("My SSN is $ssn.")

    }

    abstract fun displayJob(description: String)

}
```

1. an abstract class Person is created. You cannot create objects of the class.
2. 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.
3. 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: Kotlin Abstract Class and Method

```
abstract class Person(name: String) {
    init {
        println("My name is $name.")
    }
    fun displaySSN(ssn: Int) {
        println("My SSN is $ssn.")
    }
    abstract fun displayJob(description: String)
}
class Teacher(name: String): Person(name) {
    override fun displayJob(description: String) {
        println(description)
    }
}
```

```
fun main(args: Array<String>) {
    val avt = Teacher("Mr. Anand Tank")
    avt.displayJob("I'm a computer science teacher.")
    avt.displaySSN(92744)
}
```

When you run the program, the output will be:

```
My name is Mr. Anand Tank.
I'm a computer science teacher.
My SSN is 92744.
```

- Here, a class Teacher is derived from an abstract class Person.
- An object avt of Teacher class is instantiated. We have passed "Mr. Anand Tank" as a parameter to the primary constructor while creating it. This executes the initializer block of the Person class.
- Then, displayJob() method is called using avt object. Note, that the displayJob() method is declared abstract in the base class, and overridden in the derived class.
- Finally, displaySSN() method is called using avt object. The method is non-abstract and declared in Person class (and not declared in Teacher class).

Kotlin Interface

1. Kotlin interfaces are similar to interfaces in Java 8. They can contain definitions of abstract methods as well as implementations of non-abstract methods.
2. Meaning, interface may have property but it needs to be abstract or has to provide accessor implementations.

Keyword `interface` is used to define interfaces in Kotlin. For example,

```
interface MyInterface {

    var test: String    // abstract property

    fun foo()           // abstract method

    fun hello() = "Hello there" // method with default implementation

}
```

1. an interface MyInterface is created.
2. the interface has an abstract property test and an abstract method foo().
3. the interface also has a non-abstract method hello().

How to implement interface?

Here's how a class or object can implement the interface:

```
interface MyInterface {

    val test: Int   // abstract property

    fun foo() : String   // abstract method (returns String)

    fun hello() {   // method with default implementation

        // body (optional)

    }

}

class InterfaceImp : MyInterface {

    override val test: Int = 25

    override fun foo() = "Atmiya University"

    // other code

}
```

Here, a class `InterfaceImp` implements the `MyInterface` interface.
The class overrides abstract members (`test` property and `foo()` method) of the interface.

Example: How interface works?

```
interface MyInterface {
    val test: Int
    fun foo() : String
    fun hello() {
        println("Hello there, AU!")
    }
}

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

```
fun main(args: Array<String>) {
    val obj = InterfaceImp()
    println("test = ${obj.test}")
    print("Calling hello(): ")
    obj.hello()

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

When you run the program, the output will be:

```
test = 25
Calling hello(): Hello there, AU!
Calling and printing foo(): MCA
```