Kotlin

Notes

Rashik Ansar

April 4, 2020

Contents

1 Overview

> Kotlin is a Cross-platform, compiled, statically typed, general-purpose programming language with type inference.

- Kotlin is an official language for Android.

```kotlin
// Hello world program in Kotlin
package com.rashik

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

- The Kotlin files are saved with an extension `.kt`
- **Main function** is the **entry point** of a Kotlin program or application.
- Kotlin can be compiled for several different platforms

1.1 Comments

- In Kotlin, single line comments are defined using // and multiline comments using /**/.

```kotlin
/*
  It is a
  multiline
  comment
*/
fun sum(a: Int, b: Int): Int {
  // It is a single line comment
  return a + b
}
```

2 Variables

Variables are used to store information to be referenced and manipulated in a computer program. They also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory.

2.1 Read-only variables

- These variables are declared using keyword `val`.
- These variable can be assigned a value only once.
- If we re-assign these variables it'll throw the compile-time error.

2.1.1 Constants

- If the value is truly constant and its type is a string or primitive type then we can declare a constant variable using keyword **`const`**.
- We can only declare a constant at the top level of a file or inside an object declaration (but not inside a class declaration).

2.2 Mutables

- These variables are declared using keyword `var`.
- These variable can be re-assign hence they are mutable.

> ℹ Variable name should be in `lowerCamelCase`.
> A variable only exists inside the scope (curly-brace-enclosed block of code; more on that later) in which it has been declared - so a variable that's declared inside a loop only exists in that loop; you can't check its final value after the loop.

```kotlin
// Read-only variable
val myName = "Rashik Ansar"

// Mutalble
var year =  2020

// Constant
const val pi = 3.14

println("Hello $myName. This is $year A.D")
println("Pi value is $pi")
```

> Kotlin has a feature called String Interpolation. This feature allows us to directly insert a template expression inside a String. Template expressions are tiny pieces of code that are evaluated and their results are concatenated with the original String.
> A template expression is prefixed with `$` symbol.

3 Data types

- In Kotlin we need not to specify types of a variable explicitly unless we're not initializing the variable. (Check code of variables we didn't specify any types.)
- It has type inference feature which automatically assigns type to the variable based on the context or value provided.

> 1 Byte = 8 bits.
> value range of n bits is 2^n

Table 3.1: Storage size of different Data types

Data type	Storage size
Byte	1 Byte (8 bits)
Short	2 Bytes (16 bits)
Int	4 Bytes (32 bits)
Long	8 Bytes (64 bits)
Float	4 Bytes (32 bits)
Double	8 Bytes (64 bits)
Boolean	1 Byte (8 bits)
Char (UTF-16)	2 Bytes (16 bits)

```
fun main (args: Array<String>) {
    // Integer Types
    val myByte: Byte = 12
    val myShort: Short = 10804
    val myInt: Int = 123456789
    val myLong: Long = 12_345_678_900

    // Floating Types
    val myFloat: Float = 15.67F
    val myDouble: Double = 3.14

    // Boolean Type
    val isSunny: Boolean = true
    val isRainy: Boolean = false

    // Characters
    val myChar: Char = 'A'

    // String
    val myString: String = "This is my string..!"
    // Any character in a string can be accessed using square bracket notaion.
    // Index starts at 0 instead of 1
    var firstCharInMyString = myString[0]
    var lastCharInMyString = myString[myString.length - 1]
}
```

`String` is not a primitive data type. String literal (which you can only do with double quotes), is an immutable sequence of UTF-16 code units. `ByteArray` is a fixed-size (but otherwise mutable) byte array (and `String` can specifically not be used as a byte array).

Normal strings are defined within the double quotes (`" "`) and it can contain escape characters.
Where as Raw Strings are declared within triple double quotes (`""""""`) and it should not contain escape characters.

4 Operators

- Operators are special symbols (characters) that carry out operations on operands (variables and values).

4.1 Arithmetic Operators

Table 4.1: Arithmetic Operators

Operator	Description	Example
$+$	Addition	$15 + 5$
$-$	Subtraction	$15 - 5$
$*$	Multiplication	$15 * 5$
$/$	Division	$15/5$
$\%$	Modulus	$15\%5$

```
fun main(args: Array<String>) {
    val number1 = 15
    val number2 = 5
    var result: Int

    result = number1 + number2
    println("number1 + number2 = $result")

    result = number1 - number2
    println("number1 - number2 = $result")

    result = number1 * number2
    println("number1 * number2 = $result")

    result = number1 / number2
    println("number1 / number2 = $result")

    result = number1 % number2
    println("number1 % number2 = $result")
}
```

The + operator is also used for concatenation of strings.

4.2 Assignment Operators

Table 4.2: Assignment Operators

Operator	Equivalent	Example
=	`var a = 20`	`var a = 20`
+=	`a = a + 5`	`a += 5`
−=	`a = a - 5`	`a += 5`
∗=	`a = a * 5`	`a *= 5`
/=	`a = a / 5`	`a /= 5`
%=	`a = a % 5`	`a %= 5`

```
fun main(args: Array<String>) {
    var number = 20

    number += 5
    println("number  = $number")

    number -= 5
    println("number  = $number")

    number *= 5
    println("number  = $number")

    number /= 5
    println("number  = $number")

    number %= 5
    println("number  = $number")
}
```

4.3 Comparision Operators

Table 4.3: Comparision Operators

Operator	Description	Example
<	Less than	`a < b`
>	Greater than	`a > b`
<=	Less than or equal to	`a <= b`
>=	Greater than or equal to	`a >= b`
==	is equal to	`a == b`
!=	not equal to	`a != b`

```kotlin
fun main(args: Array<String>) {

    val a = -4
    val b = 12

    val isEqual = a == b       // it returns false
    println("isEqual is $isEqual")

    println("isNotEqual is ${a != b}")


    val max = if (a > b) {
        println("a is larger than b.")
        a
    } else {
        println("b is larger than a.")
        b
    }

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

4.4 Unary Operators

Table 4.4: Unary Operators

Operator	Description	Example
++	Increment	a++ or ++a
--	Decrement	a-- or --a
+	Unary plus	+a
-	Unary Minus (inverts sign)	-a
!	Not (inverts value)	!a

```kotlin
fun main(args: Array<String>) {
    val a = 1
    val b = true
    var c = 24

    var result: Int

    result = -a
    println("-a = $result")

    println("!b = ${!b}")

    println("--c = ${--c}")
    println("c-- = ${c--}")
    println("c = $c")
    println("++c = ${++c}")
    println("c++ = ${c++}")
    println("c = $c")
}
```

4.5 Logical Operators

Table 4.5: Logical Operators

Operator	Description	Example
||	or	(a>b)|| (a>c)
&&	and	(a>b)&& (a>c)

```kotlin
fun main(args: Array<String>) {

    val a = 54
    val b = 28
    val c = -12
    val result: Boolean

    result = (a>b) && (a>c)    // result = (a>b) and (a>c) = true
    println(result)
}
```

4.6 `in` Operator

- In operator is used to check whether an object belongs to a collection.

Operator	Equivalent	example
`in`	`b.contains(a)`	`a in b`
`!in`	`!b.contains(a)`	`a !in b`

> ⚠ There are no bitwise and bitshift operators in Kotlin.
> To perform these task, various functions (supporting infix notation) are used: `shl`, `shr`, `xor` etc.

5 Control Flow

5.1 `if` Expression

- In Kotlin, if is an expression, i.e. it returns a value.
- It is used to control the flow of program structure.

```kotlin
fun main() {
    val a = 10
    val b = 15

    // Traditional usage
    var max = a
    if (a < b) max = b

    println("Max is $max")

    // With else
    var max1: Int
    if (a > b) {
        max1 = a
    } else {
        max1 = b
    }

    println("Max1 is $max1")

    // As expression
    val max2 = if (a > b) a else b

    println("Max2 is $max2")
}
```

Listing 5.1: Compare height of two persons

```kotlin
fun main() {
  var person1Height = 170
  var person2Height = 189

  if (person1Height > person2Height){
    println("Person 1 is taller than Person 2")
  }else if (person1Height == person2Height) {
    println("Person 1 and Person 2 are of same height")
  }else{
    println("Person 2 is taller than Person 1")
  }
}
```

Listing 5.2: Greet if his name is Rashik

```
fun main() {
  val name = "Rashik"

  if(name == "Rashik") {
    println("Welcome home $name")
  } else {
    println("Who are you?")
  }
}
```

> ⚠ If you're using `if` as an expression rather than a statement (for example, returning its value or assigning it to a variable), the expression is required to have an else branch.

5.2 When Expression

- `when` replaces the switch operator of 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 `when` is used as an expression, the else branch is mandatory.

```
fun main() {
  var season = 3

  when(season) {
    1 -> println("Spring")
    2 -> println("Summer")
    3 -> {
      println("Fall")
      println("Autumn")
    }
    4 -> println("Winter")
    else -> println("Invalid season")
  }
}
```

```
fun main() {
  var month = 3
  when (month) {
    in 3..5 -> println("Spring")
    in 6..8 -> println("Summer")
    in 9..11 -> println("Fall")
    12, 1, 2 -> println("Winter")
    else -> println("Invalid month")
  }
}
```

```
fun main() {
  var x: Any = 18.97

  when (x) {
    is Int -> println("$x is Int")
    is Double -> println("$x is Double")
    is String -> println("$x is String")
    else -> println("$x is not Int, Double, or String.")
  }
}
```

5.3 Loops

> Loops are used to iterate a part of program several time.

- There are three loops in kotlin. They are

    1. While Loop
    2. Do - While Loop
    3. For Loop

5.3.1 While Loop

- While loop executes a block of code repeatedly as long as the given condition is true.

```
fun main() {
  var x = 10
  while(x > 0) {
    print("$x\t")
    x--
  }
}
```

```
fun main() {
  var x = 1
  while(x <= 10) {
    println("4 * $x\t= ${4*x}")
    x++
  }
}
```

```
// Change room temp from cold to comfortable.
fun main() {
    var feltTemp = "cold"
    var roomTemp = 10

    while (feltTemp == "cold") {
        roomTemp++
        if (roomTemp >= 20) {
            feltTemp = "comfortable"
            println("It's comfy now.")
        }
    }
}
```

5.3.2 Do-While Loop

- It is similar to `while` loop but the only difference is even though the condition is false, loop will execute atleast once.

```
// Even though the condition is false it executed once.
fun main() {
  var x = 15
  do {
    println("$x")
    x++
  } while (x <= 10)
}
```

- When do we need do-while loop?
  - ex: Menu programs.

5.3.3 For Loop

- `for` loop iterates through anything that provides an iterator.

```
fun main() {
    for(num in 1..10){
        print("$num\t")
    }

    println()

    for(i in 1 until 10){
        print("$i\t")
    }

    println()

    for(i in 10 downTo 1 step 2){
        print("$i\t")
    }
}
```

5.4 Returns and Jumps

- Kotlin has three structural jump expressions
  - **`return`**: By default returns from the nearest enclosing function or anonymous function.
  - **`break`**: Terminates the nearest enclosing loop
  - **`continue`**: Proceeds to the next step of the nearest enclosing loop

5.4.1 Break and Continue Labels

Any expression in Kotlin may be marked with a label. Labels have the form of an identifier followed by the @ sign, for example: abc@, fooBar@ are valid labels (see the grammar). To label an expression, we just put a label in front of it

```
fun main(args: Array<String>) {
  loop@ for (i in 1..3) {
    for (j in 1..3) {
      println("i = $i and j = $j")
      if (i == 2)
          break@loop
    }
  }
}
```

```
fun main(args: Array<String>) {
  labelname@ for (i in 1..3) {
    for (j in 1..3) {
      println("i = $i and j = $j")
      if (i == 2) {
        continue@labelname
      }
      println("this is below if")
    }
  }
}
```

6 Functions

Function is a group of inter related block of code which performs a specific task. Function is used to break a program into different sub module. It makes reusability of code and makes program more manageable.

- **Standard library function:** Kotlin Standard library function is built-in library functions which are implicitly present in library and available for use.
- **User defined function :** It is a function which is created by user. User defined function takes the parameter(s), perform an action and return the result of that action as a value
- Kotlin Functions are declared using `fun` keyword.

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

- We have to call the function to run the code inside the function by using function name followed by `()`.

```
functionName()
```

Check out this link for Parameter vs Argument

```
fun main() {
    var result = sum(5,9)
    myFunction()
    println("Sum of 9 and 5 is $result")
    println("Average of 5.3 and 13.37 is ${avg(5.3, 13.37)}")
}

fun myFunction() {
    println("From myFunction")
}

fun sum(a:Int, b:Int ):Int {
    return a+b
}

fun avg(a: Double, b: Double): Double {
    return (a+b)/2
}
```

```
// Example of recursice function
fun main(args: Array<String>) {
    val number = 5
    val result = factorial(number)
    println("Factorial of $number = $result")
}

fun factorial(n: Int): Long {
    return if(n == 1){
        n.toLong()
    } else {
        n*factorial(n-1)
    }
}
```

```
fun main(args: Array<String>) {
    run()
    run(9, 'a')
    run(8)
    run(letter='h')
}
fun run(num:Int= 5, letter: Char ='x'){
    println("parameter in function definition $num and $letter")
}
```

7 Kotlin Null Safety

Kotlin null safety is a procedure to eliminate the risk of null reference from the code. Kotlin compiler throws `NullPointerException` immediately if it found any **`null`** argument is passed without executing any other statements.

Kotlin's type system is aimed to eliminate `NullPointerException` form the code. `NullPointerException` can only possible on following causes:

1. An forcefully call to throw `NullPointerException()`
2. An uninitialized of this operator which is available in a constructor passed and used somewhere.
3. Use of external Java code as Kotlin is Java interoperability.

7.1 Kotlin Nullables

Kotlin types system differentiates between references which can hold null (nullable reference) and which cannot hold null (non null reference). Normally,types of String are not nullable. To make string which holds null value, we have to explicitly define them by putting a ? behind the String as: `String?`

```
fun main() {
    var name: String = "Rashik"
    // var nullableName: String? = null
    var nullableName: String? = "Rashik"

    var nameLen = name.length
    // if nullable name is null then assign value null else assign length of nullabl name
    var nullableNameLen = nullableName?.length

    println("$name has $nameLen characters")
    println("$nullableName has $nullableNameLen characters")
}
```

7.1.1 Elvis Operator

When we have a nullable reference `nullableName`, we can say “if `nullableName` is not null, use it, otherwise use some non-null value (”Guest“)”.

```
fun main() {
    var nullableName: String? = null

    val name: String = nullableName ?: "Guest"

    println("$name")
    println("$nullableName ")
}
```

7.1.2 The `!!` Operator

The third option is for `NullPointerException`-lovers: the **not-null assertion operator** (`!!`) converts any value to a non-null type and throws an exception if the value is null. We can write `nullableName!!`, and this will return a non-null value of `nullableName` or throw an `NullPointerException` if `nullableName` is **`null`**

```
fun main() {
    var nullableName: String? = "Rashik"

    val name: String = nullableName ?: "Guest"

    println("$name")
    println("$nullableName ")

    // if the value of nullableName is null then the
    // following line will throw NullPointerException
    nullableName!!.toLowerCase()
}
```

8 Object Oriented Programming

> Kotlin supports both object oriented programming (OOP) as well as functional programming. Object oriented programming is based on real time objects and classes. Kotlin also support pillars of OOP language such as **encapsulation**, **inheritance** and **polymorphism**.

8.1 Class

A class is a blueprint for the objects which have common properties. Kotlin classes are declared using keyword `class`. Kotlin class has a class header which specifies its type parameters, constructor etc. and the class body which is surrounded by curly braces.

Class body contains the data members(properties) and member functions(methods or behaviour).

- The followin two lines are same.

```
class Person constructor(firstName: String, lastName: String) {}
```

```
class Person(firstName: String, lastName: String) {}
```

8.2 Object

Object is real time entity or may be a logical entity which has state and behavior. It has the characteristics:

- state: it represents value of an object.
- behavior: it represent the functionality of an object.

Object is used to access the properties and member function of a class. Kotlin allows to create multiple object of a class.

Properties and member function of class are accessed by . operator using object.

```kotlin
fun main() {
    var rashik = Person("Rashik Ansar", "Shaik")
    println(rashik.hobby)
}
```

```kotlin
fun main() {
    var unknown = Person()

    var rashik = Person("Rashik Ansar", "Shaik")
    rashik.stateHobby()
    rashik.hobby = "Play video games"
    rashik.stateHobby()

    var kevin = Person("Kevin", "Hart", 40)
    kevin.hobby = "Cracking jokes"
    kevin.stateHobby()
}


class Person(firstName: String = "John", lastName: String = "Doe") {
    // Data members
    var firstName: String ?= null
    var lastName: String ?= null
    var age: Int ?= null
    var hobby: String = "watch Netflix"

    // Initializer block
    init {
        this.firstName = firstName
        this.lastName = lastName
        println("First Name: $firstName, Last Name: $lastName")
    }

    // Member secondary constructor
    constructor(firstName: String, lastName: String, age: Int): this(firstName, lastName) {
        this.age = age
        println("First Name: $firstName, Last Name: $lastName, age: $age")
    }


    // Member functions
    fun stateHobby() {
        println("$firstName\'s hobby is $hobby")
    }
}
```

> Why kotlin allows to declare variable with the same name as parameter inside the method?

```kotlin
fun main() {
    var myCar = Car()
    println("Brand is: ${myCar.myBrand}")
    myCar.maxSpeed = 242
    println("Max Speed is ${myCar.maxSpeed}")
    println("Model is ${myCar.myModel}")
}

class Car() {
    lateinit var owner: String

    val myBrand: String = "bmw"
    get() {
        return field.toUpperCase()
    }

    var maxSpeed: Int = 250
    get() = field
    set(value) {
        field = if(value >= 0 && value <= 250) value else throw IllegalArgumentException("Invalid speed")
    }

    var myModel: String = "M5"
    private set

    init {
        this.owner = "Frank"
    }

}
```

```kotlin
// data class
data class User(val id: Long, var name: String)

fun main() {
    val user1 = User(1, "Rashik Ansar")

    println(user1.name)
    println(user1.id)

    println(user1.component1())
    println(user1.component2())


    val (id, name) = user1
    println("id: $id \t name: $name")

    user1.name = "Alpha"
    println(user1.name)

    val user2 = user1.copy(name="Alpha")
    println(user1.equals(user2))
    println(user2.name)
}
```

8.3 Inheritance

The class that inherits the features of another class is called the **Sub class** or the **Child class** or the **Derived class**. The class whose features are inherited is known as **Super class** or **Parent class** or **Base class**

> All classes in Kotlin have a common superclass Any, that is the default superclass for a class with no supertypes declared
>
> `Any` has three methods: `equals()`, `hashCode()` and `toString()`. Thus, they are defined for all Kotlin classes.

By default, Kotlin classes are **`final`**: they **can't be inherited**. To make a class inheritable, mark it with the `open` keyword.

```
fun main() {
   var audiA3 = Car("A3", "Audi")
   var teslaS = ElectricCar("S-Model", "Tesla", 85.0)

   audiA3.drive(200.0)
   teslaS.drive(200.0)
   teslaS.drive()
}

// Super class of electric car
open class Car(val name: String, val brand: String) {
    open var range: Double = 0.0

    fun extendedRange(amount: Double) {
        if (amount > 0) {
            range += amount
        }
    }

    open fun drive(distance: Double) {
        println("Drove for $distance KMs")
    }
}

// sub class of Car
class ElectricCar(name:String, brand: String, batteryLife: Double): Car(name, brand) {
    override var range = batteryLife * 6

    override fun drive(distance: Double) {
        println("Drove for $distance KMs on battery")
    }

    fun drive() {
        println("Drove for $range KMs on electricity")
    }
}
```

8.4 Abstract class

A class and some of its members may be declared abstract. An abstract member does not have an implementation in its class. Note that we do not need to annotate an abstract class or function with open – it goes without saying.

We can override a non-abstract open member with an abstract one

> Abstract classes are always `open`. You do not need to explicitly use `open` keyword to inherit subclasses from them.

```
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 jack = Teacher("Jack Smith")
    jack.displayJob("I'm a mathematics teacher.")
    jack.displaySSN(23123)
}
```

Kotlin interfaces are similar to abstract classes. However, interfaces cannot store state whereas abstract classes can. Interfaces cannot have constructors.

8.5 Interface

Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations.

- Using interface supports functionality of multiple inheritance.
- It can be used achieve to loose coupling.
- It is used to achieve abstraction.

```
interface Drivable {
    val maxSpeed: Double
    fun drive(): String
    fun brake() {
        println("Vehicle is slowing down")
    }
}

// Super class of electric car
open class Car(override val maxSpeed: Double, val name: String, val brand: String): Drivable {
    open var range: Double = 0.0

    fun extendedRange(amount: Double) {
        if (amount > 0) {
            range += amount
        }
    }

    open fun drive(distance: Double) {
        println("Drove for $distance KMs")
    }

    override fun drive(): String {
        return "Driving......"
    }
}

// sub class of Car
class ElectricCar(
    maxSpeed: Double,
    name:String,
    brand: String,
    batteryLife: Double
): Car(maxSpeed, name, brand) {
    override var range = batteryLife * 6

    override fun drive(distance: Double) {
        println("Drove for $distance KMs on battery")
    }

    override fun drive(): String {
        return "Drove for $range KMs on electricity"
    }

    override fun brake() {
        super<>.brake()
        println("Brake from electric car")
    }
}

fun main() {
   var audiA3 = Car(220.0, "A3", "Audi")
   var teslaS = ElectricCar(240.0, "S-Model", "Tesla", 85.0)

   audiA3.drive(200.0)
   teslaS.drive(200.0)
   teslaS.drive()

   teslaS.brake()
   audiA3.brake()
}
```

To override the non-abstract method `brake()` we need to specify interface name with method using **`super`** keyword as **`super`**`<interface_name>.methodName()` for immediate parent. For sub class then we only use the **`super`**`.methodName()`.

9 Miscellaneous

9.1 Array List

Kotlin ArrayList class is used to create a dynamic array. Which means the size of ArrayList class can be increased or decreased according to requirement. ArrayList class provides both read and write functionalities.

Kotlin ArrayList class follows the sequence of insertion order. ArrayList class is non synchronized and it may contains duplicate elements. The elements of ArrayList class are accessed randomly as it works on index basis.

```
fun main(){
    val myArrayList: ArrayList<Double> = ArrayList()
    myArrayList.add(13.212312)
    myArrayList.add(23.151232)
    myArrayList.add(32.651553)
    myArrayList.add(16.223817)
    myArrayList.add(18.523999)
    var total = 0.0
    for (i in myArrayList){
        total += i
    }
    var average = total / myArrayList.size
    println("Avarage is " + average)
}
```

9.2 Lamda Functions

Lambda is a function which has no name. Lambda is defined with a curly braces {} which takes variable as a parameter (if any) and body of function. The body of function is written after variable (if any) followed by -> operator.

- The following two are same

```
fun main() {
    val sum: (Int, Int) -> Int = {a: Int, b: Int -> a + b}
    println(sum(10,5))
 }
```

```
fun main() {
    val sum = {a: Int, b: Int -> a + b}
    println(sum(10,5))
}
```

9.3 Visibility Modifiers

- Visibility modifiers are the `keywords` which are used to restrict the use of classes, interfaces, methods, and properties in Kotlin.
- These modifiers are used at multiple places such as class headers or method body.
- Visibility Modifiers are categorized into four different types

  1. Public - Default modifier
  2. Private - Accessible within the block in which properties, methods etc are declared.
  3. Protected - Visible to its class or sub class. It cannot be declared at toplevel in Packages.
  4. Internal - Accessible only inside the module in which it is implemented

> In Kotlin all classes are **`final`** by default, so they cannot be inherited by default. So, to make a class inheritable to other classes then we must mark the class with `open` keyword, else we get an error “type is final so cant be inherited.”

9.4 Nested Class

- A class which is created inside an another class.
- In Kotlin, a nested class is by default **static**, so its data members and member functions can be accessed without creating an object of the class.
- Nested Classes cannot access the data members of outer classes.

9.5 Inner Class

- A class which is created inside another class using `inner` keyword. In other words, nested class which is marked with `inner` keyword
- Inner class cannot be declared inside interfaces or non-inner nested classes.

- Unlike nested class, It is able access members of its outer class even they are in **`private`**.
- Inner class keeps a reference to an object of its outer class.

9.6 Unsafe Cast Operator: as

- Sometimes it's not possible to cast a variable and it throws an exception, this is called an **unsafe cast**.
- The unsafe cast is performed by the infix operator `as`.

```
// A nullable string (String?) cannot be cast to non-nullable string (String)
// This throws an exception
fun main() {
  val x: Any ?= null
  val y: String = x as String
  println(y)
}
```

Output: `Exception in thread "main"kotlin.TypeCastException: null cannot be cast to non-null type kotlin String.`

9.7 Safe cast operator: as?

- `as`? provides a safe cast operation to safely cast to a type.
- It return a null if casting is not possible rather than throwing a `ClassCastException`.

```
fun main() {
  val x: Any = "Kotlin"
  val safeString : String = x as? String
  val safeInt: Int = x as? Int
  println(safeString)  // Kotlin
  println(safeInt)    // null
}
```

9.8 Exception Handling

- EXception is a runtime problem which occur in the program and leads to program termination. There are two types of excpetions they are

1. **UnChecked Exception:** this exception type extends `RuntimeException` class
    - ArithmeticException
    - ArrayIndexOutOfBoundException
    - SecurityException
    - NullPointerException etc.
2. **Checked Exception:** this exception type extends `Throwable` class

- IOException
- SQLException etc.

- Exception handling is a technique which handles the runtime problems and maintains the flow of program execution.
- The four keywords used in exception handling are

1. `try` **:** contains a set of statements which might generate an exception. It must be followed by `catch` or `finally` or both.
2. `catch` **:** It catches the exception thrown from the try block
3. `finally` **:** It always executes whether exception is handled or not. So it is mostly used to execute important code statements (like closing buffer)
4. `throw` **:** Used to throw an exception explicitly

```kotlin
fun main (args: Array<String>){
    try {
        val data = 5 / 0
        println(data)
    } catch (e: ArithmeticException) {
        println(e)
    } finally {
        println("finally block always executes")
    }
}
```

10 References

- Kotlin Official Docs
- JavaTPoint Kotlin
- Interview Questions