ANDROID

SESSION ONE

General Rules

1. Respect Everyone
2. Never Down talk anyone who is younger than you or has a lower position (we are all students, here to learn)
3. Differentiate between Work and Joking around (having fun)
4. If you have any issues with a member or an instructor (or any person) always get back to your Committee HR or Vice HR responsible for your committee to resolve your issue
5. What happens in Your committee stays in Your committee
6. Anyone who doesn’t attend the session without any excuse will receive a Punishment
7. Punishment will go as follows:
    a. First incident: buzz
    b. Second incident: Warning
    c. Third incident: Warning
    d. Fourth incident: Firing interview.
8. Bonus Points
9. Tasks should be delivered in the deadline
10. All sessions will be uploaded in the drive

Hierarchy

President

Head HR , Vice president

Project Manager

Heads , Vice Head HR

Vice Heads , Team leaders HR

Team leaders , Members HR

Agenda

- Data Types
- Variables
- How to take input
- Operator in Kotlin
- Controlling Program Flow (if ,switch ,loops)
- Functions (Top-Level Functions,Extension Functions,Lambda)

Why Kotlin over Java?

- Less Boilerplate Code
- Type Inference
- Null Safety
- Extension Functions
- Interoperability with Java
- Coroutines for Asynchronous

Data Types

we have the regular data types

Int -> Integer

Double -> Fractions

String -> Texts

Boolean -> True , False

Char -> Characters

Variables

Not like Java here, we have two types of variables

var

its referee to variable which I can read and write in its value

```
var x = 6

x = 10
```

val

its referee to value which it is read only

```
val x = 6

x = 10
```

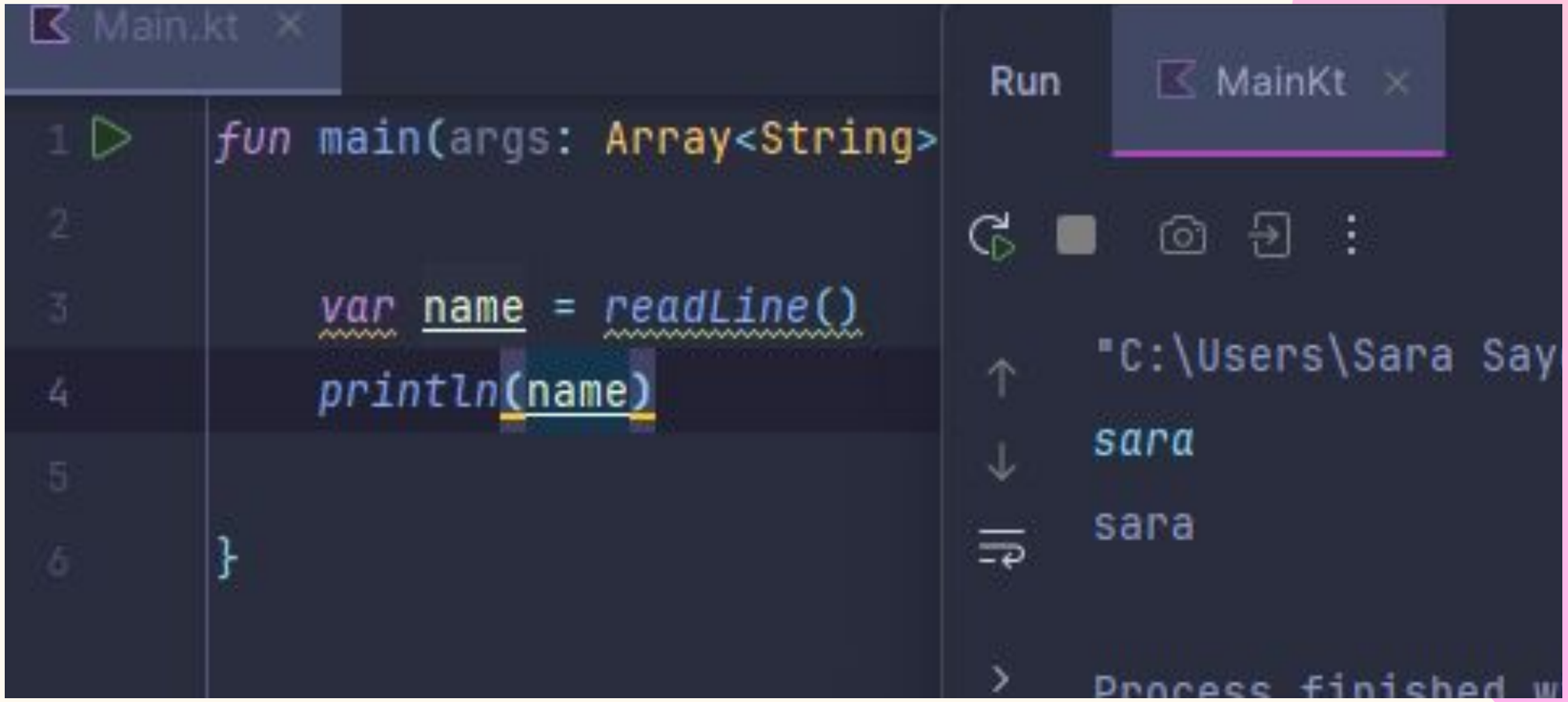
How to use?

```
var number: Int
var fraction: Double
var text: String
var boolean: Boolean
var character: Char
```

How to take input!!

We have two ways to take an input

- **First:** readline()



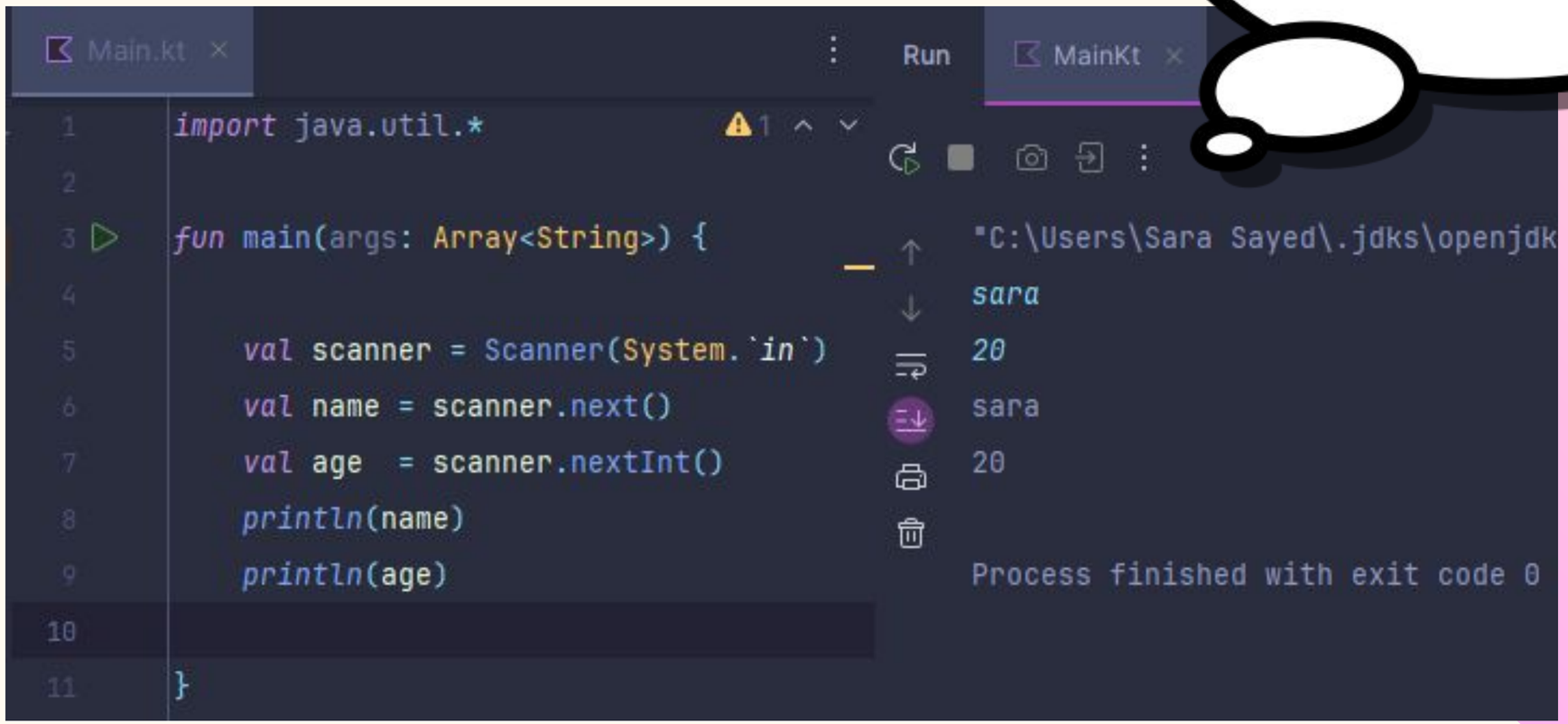
How to take input!!

We have two ways to take an input

- **Second:** Scanner class

it like scanner in java we should

- import ”Scanner class”
- get instance from scanner



OPERATORS

Operators

- **Arithmetic Operators**

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 from another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value by 1	++x
--	Decrement	Decreases the value by 1	--x

Operators

- **Assignment Operators**

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Operators

- **Comparison Operators**

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Operators

- **Logical Operators**

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	

Operators

- **Operators cont’d.**

Operator	Meaning
!!	Asserts that an expression is non-null.
?.	Performs a safe call(calls a method or accesses a property if the receiver is non-null.
?:	Takes the right-hand value is null (Elvis Operator).
::	Creates a member reference or a class reference.
$	References a variable or expression in a string template.
_	Substitutes an unused parameter.
?	Marks a type as nullable.

- **The $ operator**

  :: is used for embeding variables or expressions inside strings.

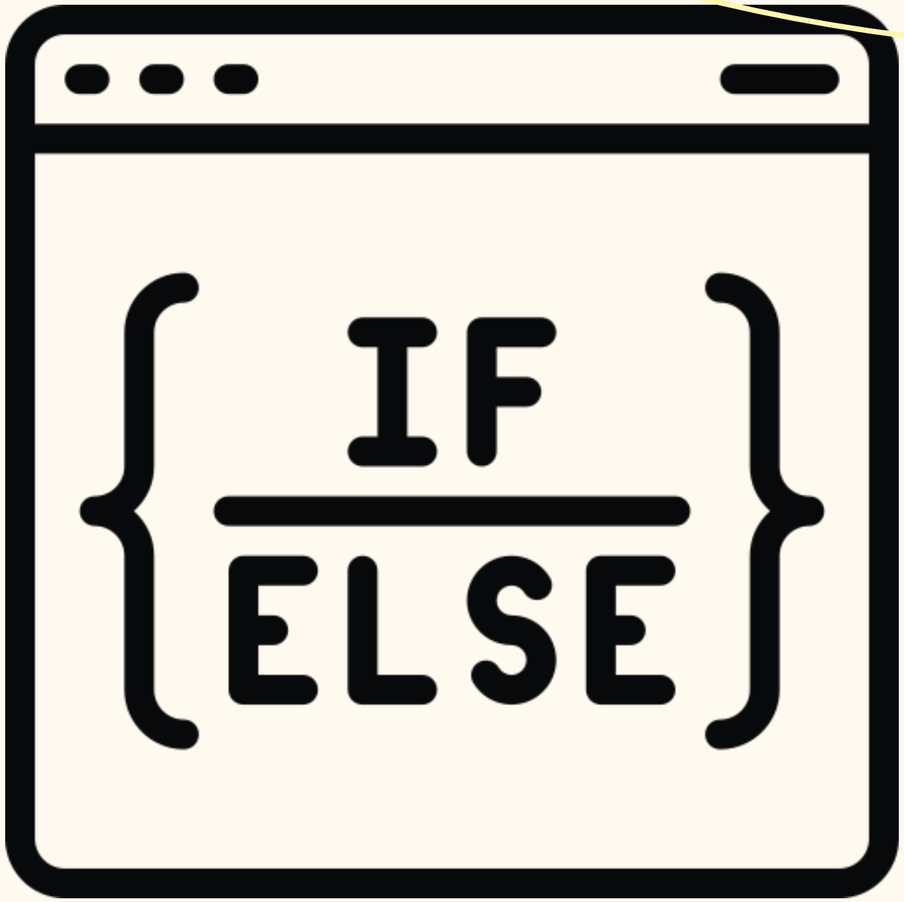
```
fun main() {
    val name = "John"
    val age = 25
    println("My name is $name and I am $age years old.")

}
```

My name is John and I am 25 years old.

Outputs

The sum of 10 and 5 is 15.

```
fun main() {
    val x = 10
    val y = 5
    println("The sum of $x and $y is ${x + y}.")
}
```


IF
ELSE

If..Else Expressions

- Use if to specify a block of code to be executed if a condition is **true**.
- Use else to specify a block of code to be executed if the condition is **false**.

```
val x = 20
val y = 18
if (x > y) {
  println("x is greater than y")
}
```

The Output ➤ x is greater than y

```
val time = 20
if (time < 18) {
  println("Good day.")
} else {
  println("Good evening.")
}
```

The Output ➤ Good evening

If..Else Expressions

- In Kotlin, you can also use if..else statements as ***expressions*** (assign a value to a variable and return it):

```
val time = 20
val greeting = if (time < 18) {
  "Good day."
} else {
  "Good evening."
}
println(greeting)
```

Outputs

Good evening

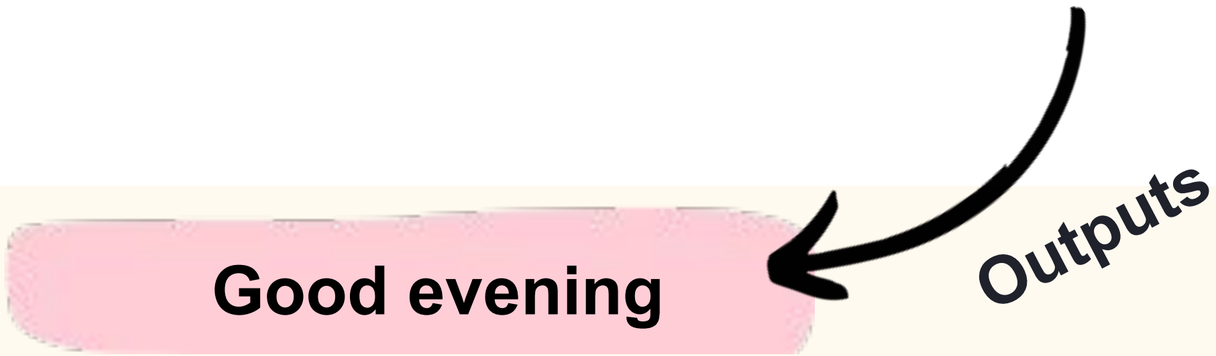
If..Else Expressions

Notes :

- When using if as an expression, **you must also include *else*** (required).

- You can ommit the curly braces {} when if has only one statement:

```
fun main() {
  val time = 20
  val greeting = if (time < 18) "Good day." else "Good evening."
  println(greeting)
}
```



When

- The when expression is similar to the ***switch*** statement in Java.
- Instead of writing many if..else expressions, you can use the when expression, which is much easier to read .

```
fun main() {
  val day = 4

  val result = when (day) {
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
    7 -> "Sunday"
    else -> "Invalid day."
  }
  println(result)
}
```

Outputs

Thurusday

STRINGS

Strings

A string value is a sequence of characters in double quotes. (“)

Kotlin has two types of string literals:

- **Raw String (multiline):** can contain newlines and arbitrary text , with (“””):

```
val rawString = """
\nFirst Line
Second Line with \t"quotes"
Third Line with a backslash """
println(rawString)
```

Outputs

```
\nFirst Line
Second Line with \t"quotes"
Third Line with a backslash
```

- **Escaped strings:** can contain escaped characters, with (“).

\n : Newline

\t : Tab

**** : Backslash

\” : Double Quote

```
val escapedString = "Hello,\nWorld!\tThis is Kotlin."
println(escapedString)
```

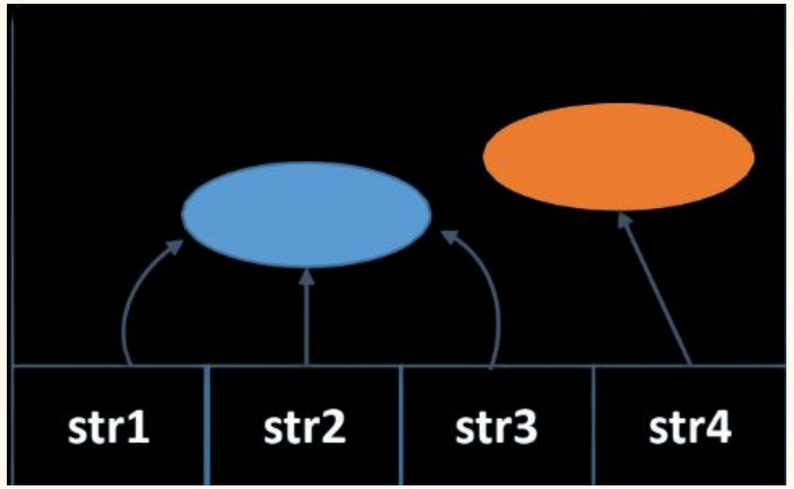
Outputs

```
Hello,
World!  This is Kotlin.
```

Strings

- **Strings are Immutable.**
- **To access the characters (elements) of a string, you must refer to the index number inside square brackets.**
- **String Pool**

  1. The string pool is a memory optimization technique used to store unique instances of strings,(Instead of creating multiple identical string objects, the string pool ensures that identical strings share the same memory reference.).



Strings

- **Interpolation :**

```
val items = 3
val message = "You bought $items items."
println(message)
```

Outputs → **You bought 3 items.**

- **String Equality :**

1. **Structural Equality (==)**

```
val str1 = "Kotlin"
val str2 = "Kotlin"
val str3 = "Java"

println(str1 == str2) // true
println(str1 == str3) // false
```

Checks if two strings have the same content.

2. **Referential Equality (===)**

```
val str1 = "Kotlin"
val str2 = "Kotlin"
val str3 = String(charArrayOf('K', 'o', 't', 'l', 'i', 'n'))

println(str1 === str2) // true (string literals are interned)
println(str1 === str3) // false
```

Checks if two references point to the same object (memory reference).

Booleans

For this, Kotlin has a Boolean data type, which can take the values true or false.

A boolean type can be declared :

- with the Boolean keyword

```
val isKotlinFun : Boolean = true
```

- only take the values true or false

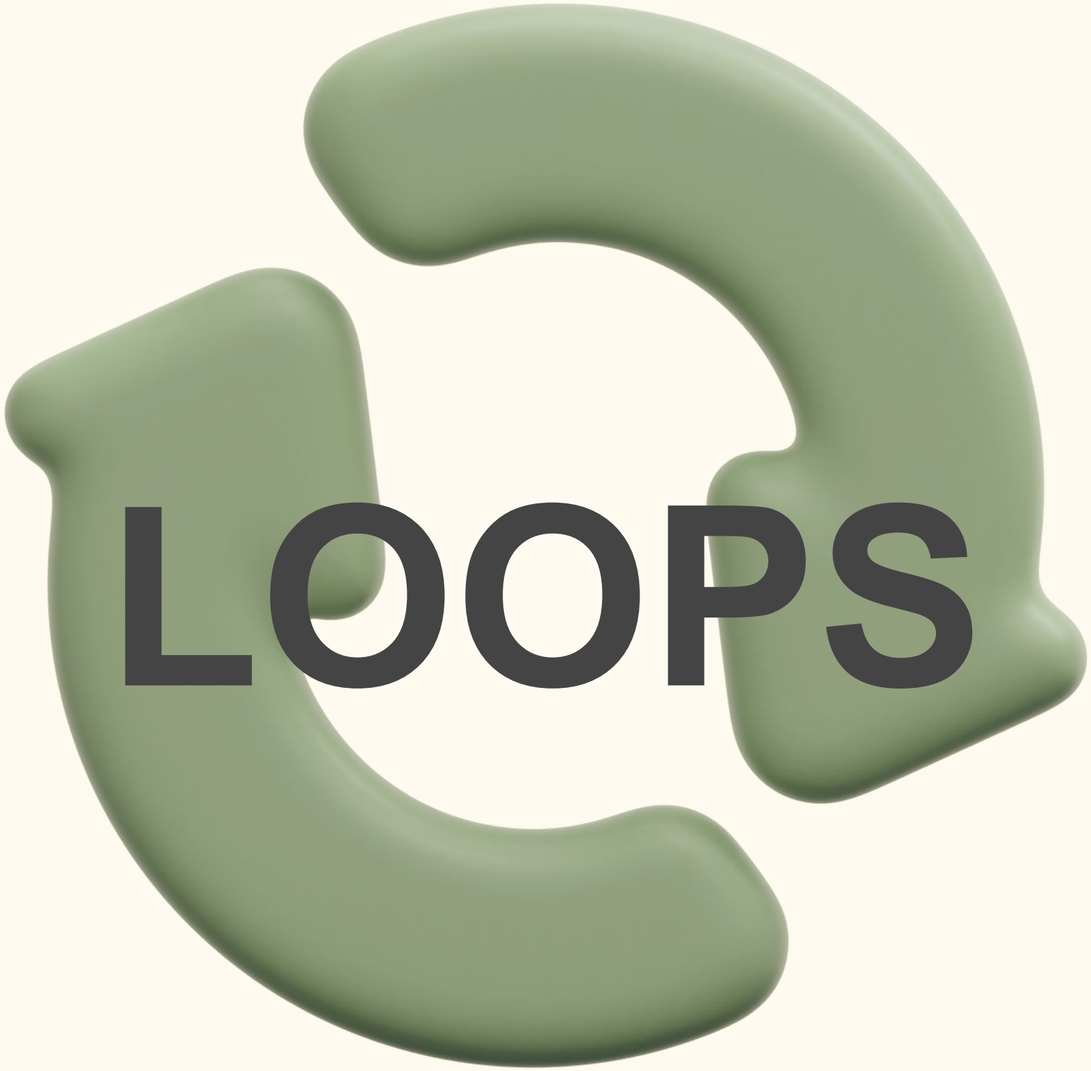
```
val isKotlinFun = true
```

Boolean Expression

A Boolean expression returns a Boolean value: true or false, you can use a comparison operator or use the equal to (==).

```
val x = 10
val y = 9
println(x > y)
```

```
val x = 10;
println(x == 10);
```


LOOPS

Loops-While loop

- Repeats the entire code block while the condition is true.
- Evaluates the condition **before** executing the code block.

```
fun main(){
    var x =0
     while (x<=12){
         println("X now equal $x")
        x=x+2
     }
}
```

```
X now equal 0
X now equal 2
X now equal 4
X now equal 6
X now equal 8
X now equal 10
```

Output

Loops-Do While loop

- Repeats the entire code block while the condition is true, ensuring the code executes at least once.
- Checks the condition **after** the first execution

```
fun main(){
    var x =1
    do {
        println("first iteration")
        x++
    }while (x==0)
}
```

```
first iteration
```

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

```
x= 1
x= 5
x= 9
```

Loops-For loop

- Used when the number of iterations is **known or predetermined.**
- Checks the condition **before** starting the loop.

```
fun main(){
    for (i in 1 ≤ .. ≤ 5){
        println("We in iteration number $i")
    }
}
```

```
We in iteration number 1
We in iteration number 2
We in iteration number 3
We in iteration number 4
We in iteration number 5
```

in keyword :

Checks if a value is part of a group,Used to see if a value is inside a range, list, or any collection.

Range supports performing :

1.Step

- Modifies the step size in a range, allowing you to **increase or decrease** the values by a specific increment.
- Example: Increasing steps: 1, 3, 5, 7 (step size of 2)
- Decreasing steps: 5, 3, 1 (step size of -2).

```
fun main(){
    for (i in 1 ≤ .. ≤ 5 step 2){
        println("We in iteration number $i")
    }
}
```

```
fun main(){
    for (i in 5 ≤ .. ≤ 1 step -2){
        println("We in iteration number $i")
    }
}
```

```
We in iteration number 1
We in iteration number 3
We in iteration number 5
```

```
ion Create breakpoint : Step must be positive, was: -2.
```

Range supports performing :

2.DownTO

To count down until we reach a specific value

```
fun main(){
    for (i in 5 ≥ downTo ≥ 1 step 2){
        println("i = $i")
    }
}
```

```
i = 5
i = 3
i = 1
```

3. Untill

Until is **exclusive**: The last value is **not** included, so 1 until 10 includes 1 to 9, but not 10.

```
fun main(){
    for (i in 1 ≤ until < 5 step 2){
        println("i = $i")
    }
}
```

```
i = 1
i = 3
```

Repeat()

- Runs the given function a specific **number of times**.
- Passes the current index (**starting from zero**) to the function.

```
fun main(){
    repeat(5){ it: Int
        println("Hello user number $it")
    }
```

```
Hello user number 0
Hello user number 1
Hello user number 2
Hello user number 3
Hello user number 4
```

Output

Break and contunie

Break:

- Ends the nearest loop.
- Jumps to the **first statement after the loop**.

Continue:

- Used only with loops.
- Skips the **current loop iteration** and moves to the next one.

```
fun main(){
    for (i in 1 ≤ .. ≤ 5){
        if (i ==3) break
        println("$i")
    }
println("The loop is ended")
```

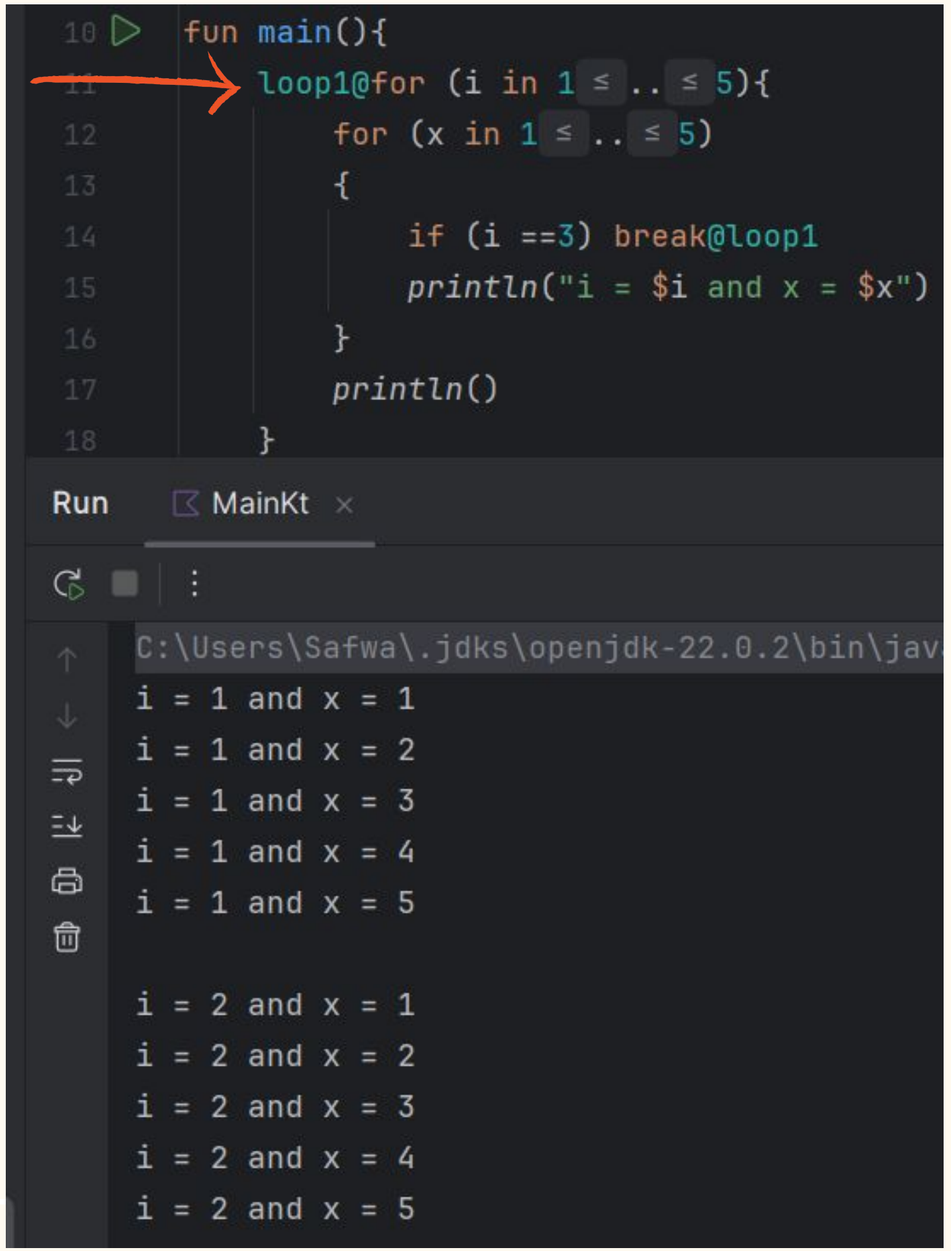
```
Run  MainKt
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin
1
2
The loop is ended
```

```
fun main(){
    for (i in 1 ≤ .. ≤ 4){
        if (i ==3) continue
        println("$i")
    }
```

```
Run  MainKt
C:\Users\Safwa\.jdks\openjdk-22.0.2\
1
2
4
```

Break and contunie

- In Kotlin, any statement can be marked with a label.
- A label is an identifier **followed by the @** symbol (e.g., loop@).
- Labels allow you to control the **flow of break, continue, and return** by qualifying them with a label, and can only be used by them



Loop- Foreach

The forEach() function loops through **collections and ranges**, performing a specified action on each element. It simplifies working with each item in a list, array, or other data structure.

1.Foreach with ranges:

```
fun main(){
    (1 ≤ .. ≤ 5).forEach{ println("Iteration number $it")}
```

```
Iteration number 1
Iteration number 2
Iteration number 3
Iteration number 4
Iteration number 5
```

Loop- Foreach

2.Foreach with Lists:

```
listOf(1,2,3,4,5).forEach { it: Int
    println(it)
}
```

```
1
2
3
4
5
```

3.Foreach with Index:

```
fun main(){
    var order =listOf("first","second","third")
    order.forEachIndexed { index, it->
        println("$it of index $index")
    }
}
```

```
first of index 0
second of index 1
third of index 2
```

Loop- Foreach

```
listOf(1,2,3,4,5,6,7,8).forEach { it: Int
    if (it == 3) break
    println(it)
}
```

'break' and 'continue' are only allowed inside loops.

Continue in for each:

```
fun main(){
    listOf(1,2,3,4,5,6,7,8).forEach { it: Int
        if (it == 3) return@forEach
        println(it)
    }
}
```

Loop- Foreach

```
listOf(1,2,3,4,5,6,7,8).forEach { it: Int
    if (it == 3) break
    println(it)
}
```

```
'break' and 'continue' are only allowed inside loops.
```

Break in for each:

```
fun main(){
    listOf(1,2,3,4,5,6,7,8).forEach { it: Int
        if (it == 3) return
        println(it)
    }
}
```

```
1
2
```

FUNCTIONS

Introduction to Functions in Kotlin

Definition:

A function is a reusable block of code that performs a specific task. parameters are passed by value.

```
fun functionName(parameter1: Type1, parameter2: Type2): ReturnType {
    // Function body
    return result
}
```

Example:

Return Function

```
fun add(x:Int , y:Int ) : Int
{
    return x+y
}
```

Void Function

```
fun add(x:Int , y:Int ) : Unit
{
    println("result $x + $y = ${x+y}")
}
```

Top-Level Functions

Functions defined directly in a Kotlin file, outside any class

```
fun main()
{
greats()
}
fun greats()
{
    println("HI")
}
```

Named Parameters

you can freely change the order they are listed in

```
fun main() {
    createPerson( name: "Ali",  age: 30)
    createPerson(age=18, name = "Ahmed")
    createPerson( name: 18, age: "abdallah")
    createPerson(city= "obour city",name="adham",age=21)
}
fun createPerson(name: String, age: Int, city: String = "Unknown") {
    println("My Name: $name")
    println("Age: $age")
    println("City: $city")
}
```

Default Values

```
fun main() {
    repeatChar()
    repeatChar(char = '#')
    // repeatChar(5) // Error
    repeatChar(char = '#', repeatNumber = 5)
}

fun repeatChar(char: Char = '*', repeatNumber: Int = 2) {
    repeat(repeatNumber) {
        print("$char")
    }
}
```

output:

1- **

2- ##

3-syntax error

4-#####

Single-Expression Function

You can write your function as a Single Expression Function

if:

- The function has a return type
- The function body contains one line only

```
fun square(x: Int): Int {
    return x * x  // Full definition
}

fun square(x: Int) = x * x  // Expression body
```

Example:

```
fun main() {
    println(cheak( x: 5))

}
//fun cheak(x:Int)
//{
//    if(x%2==0)
//        return "Even"
//    else
//        return "odd"
//}
fun cheak(x:Int) = if (x%2==0) "Even" else "odd"
```

Extension Functions

- Allows adding new functions to existing classes without modifying their code
- Syntax: fun ClassName.functionName()

```
fun main() {
    val str: String = "This string has a length of: 26"
    println("Is this string of even length? ${str.isEvenLength()}")
}

fun String.isEvenLength(): Boolean {
    return if (this.length%2==0) true else false
}
```

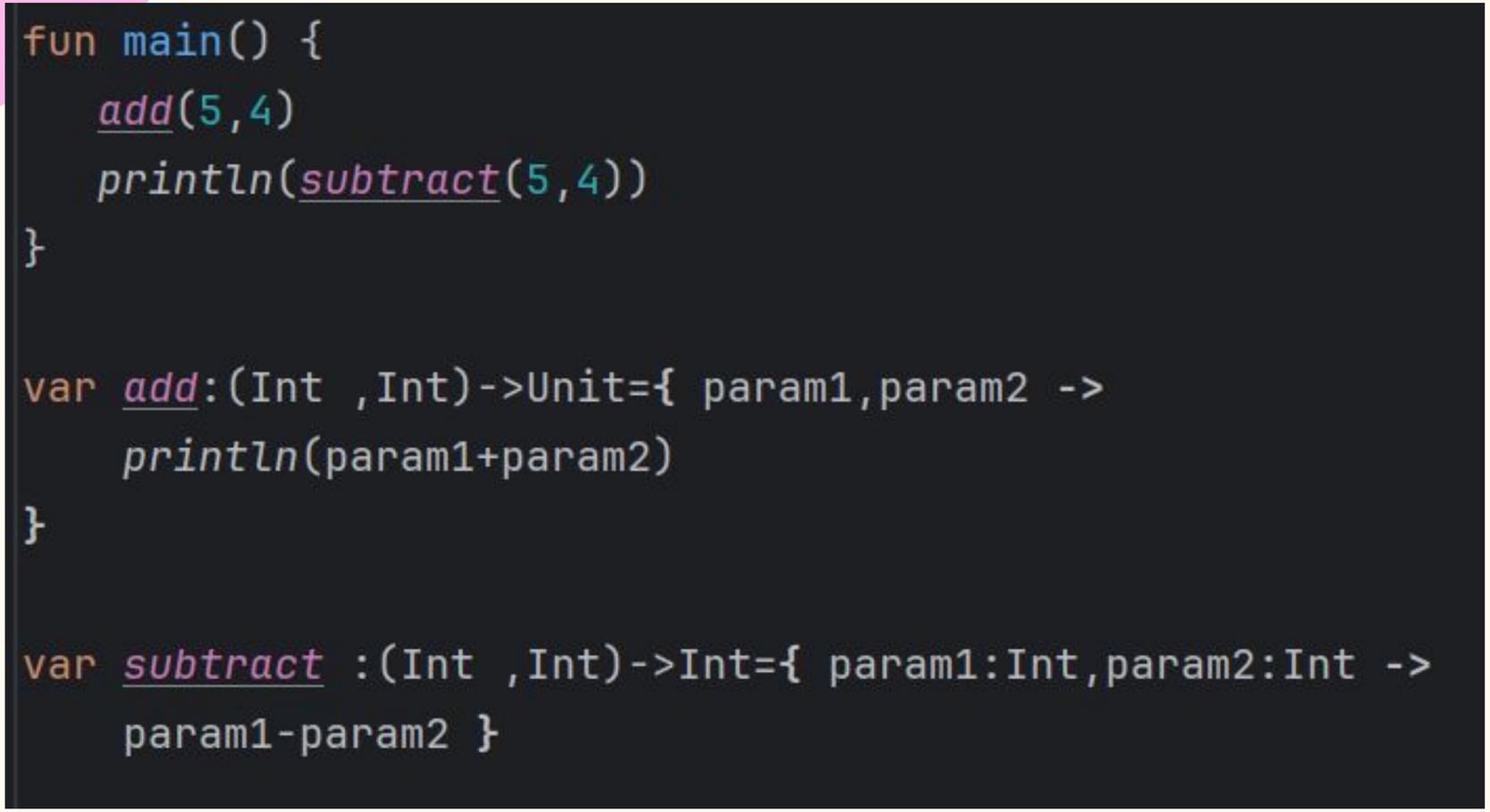
output:
true

Lambda Function

- A lambda function is an anonymous function that can be treated as a value.
- Syntax :

```
val lambdaName: (Type1, Type2) -> ReturnType = { parameter1, parameter2 ->
    // Function body
}
```

Example:



```
fun main() {
    add(5,4)
    println(subtract(5,4))
}

var add:(Int ,Int)->Unit={ param1,param2 ->
    println(param1+param2)
}

var subtract :(Int ,Int)->Int={ param1:Int,param2:Int ->
    param1-param2 }
```

Bouns Example:

what is **X** and **y** datatype ?

x : (Int,Int)->Int

y: Int

```
fun main() {
  var x=subtract
  var y=x(2,1)
}
var subtract :(Int ,Int)->Int={ param1:Int,param2:Int ->
    param1-param2 }
```

Lambda functions are a powerful feature in Kotlin, but why should we learn them?

- We should learn lambda functions because they can be used as function parameters

How can a function be a parameter in another function?

- Lambda functions are treated like variables .

Higher-Order Function

- A higher-order function is a function that can take other functions as parameters.
- Syntax :

```
inline fun higherOrderFunction(param: (Type1, Type2) -> ReturnType): ReturnType {
    // Function body
}
```

inline

- The inline keyword suggests to the compiler that the function should be inlined at the call site.

- **Best Practice**: Using inline is considered a best practice when working with higher-order functions, as it can help reduce the performance cost associated with passing functions as parameters.

Example:

- **first** : defineHigher-Order Function

```
//Higher-Order Function
inline fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b) // Call the passed function with arguments a and b
}
```

- **second**: Define a lambda Functions

```
// Define a lambda for addition
val add: (Int, Int) -> Int = { x, y -> x + y }
// Define a lambda for multiplication
val multiply: (Int, Int) -> Int = { x, y -> x * y }
```

Example:

- **finally** : call the higher order function

```
fun main() {
    // Call the higher-order function with different operations
    val sum = performOperation( a: 5,  b: 3, add)       // Output: 8
    val product = performOperation( a: 5,  b: 3, multiply) // Output: 15

    println("Sum: $sum")                          // Output: Sum: 8
    println("Product: $product")                   // Output: Product: 15
}
```

Example:

the last code but full ! !

```
fun main() {
    // Call the higher-order function with different operations
    val sum = performOperation( a: 5,  b: 3, add)       // Output: 8
    val product = performOperation( a: 5,  b: 3, multiply) // Output: 15

    println("Sum: $sum")                        // Output: Sum: 8
    println("Product: $product")                // Output: Product: 15
}
//Higher-Order Function
fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b) // Call the passed function with arguments a and b
}
// Define a lambda for addition
val add: (Int, Int) -> Int = { x, y -> x + y }
// Define a lambda for multiplication
val multiply: (Int, Int) -> Int = { x, y -> x * y }
```

Example:

- Using Higher-Order Functions with Collections

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5)

    val squaredNumbers = numbers.map { it * it }
    println(squaredNumbers) // Output: [1, 4, 9, 16, 25]

    val evenNumbers = numbers.filter { it % 2 == 0 }
    println(evenNumbers) // Output: [2, 4]
}
```

Example:

- send two lambda functions directly as arguments

1. first define Higher order function

```
// A higher-order function that takes a string and two lambda functions as parameters
fun playwithstring(
    str: String, // Input string to be manipulated
    print_string: (String) -> Unit, // Lambda function that takes a string
    reverse_string: (String) -> Unit // Lambda function that takes a string
) {
    // Call the lambda function
    print_string(str)
    // Call the lambda function
    reverse_string(str)
}
```

Example:

- send two lambda functions directly as arguments

2. call Higher order function, and pass parameters

```
fun main() {
    // Call the higher-order function 'playwithstring' with the input "MSP"
    // Provide two lambda functions directly as arguments:
    // 1. The first lambda function prints the string .
    // 2. The second lambda function reverses the string and then prints it.
    playwithstring( str: "MSP",
        { it -> println(it) }, // Prints "MSP"
        { str -> println(str.reversed()) } // Prints "PSM"
    )
}
```

SAFE CAST AND NULL SAFETY

Type checks and casts

In Kotlin, you can perform type checks to determine the type of an object at runtime when the type might not be known at compile time

Type casts enable you to convert objects to a different type.

is and !is operators

The **is** operator is used to check whether an object is of a specific type at runtime. If the object is of the type being checked, the is operator returns true; otherwise, it returns false.

```
val obj: Any = "Hello"

if (obj is String) {
    println(obj.length) // Smart cast to String happens here
}

if (obj !is String) { // Same as !(obj is String)
    print("Not a String")
} else {
    print(obj.length)
}
```

Output :

5

5

Smart casts

Once Kotlin verifies that the object is of the expected type using the is operator, it **smart casts** the object to that type automatically, meaning you do not need to explicitly cast it

As in the example , after confirming obj is String, Kotlin treats obj as a String, allowing you to call String methods like length.

```
val obj: Any = "Hello"

if (obj is String) {
    println(obj.length) // Smart cast to String happens here
}
```

Note: Smart casting only happens if the object is immutable (i.e., it cannot change its type between checks).

Type Casts

Type casting in Kotlin allows you to **explicitly** convert an object from one type to another, typically a subtype or supertype. There are two types of casting in Kotlin:

1. **Unsafe Cast (as):** The as operator is used for casting an object to another type, but it assumes the cast will succeed. If it doesn't, a ClassCastException is thrown at runtime.

```
val obj: Any = "Kotlin"

val str: String = obj as String // Safe, obj is a String
println(str.length)

val number: Int = obj as Int // This will throw ClassCastException at runtime
```

2.Safe Cast (as?) :

Kotlin provides a safer version of the cast operator called as?. It attempts to cast an object to the target type and returns null if the cast fails, avoiding the **ClassCastException**.

```
val obj: Any = "Kotlin"
val str: String? = obj as? String // Safe cast, returns "Kotlin"
println(str?.length) // Safe access with null-check

val number: Int? = obj as? Int // Safe cast, returns null
```

Null safety

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference results in a null reference exception. In Java, this would be the equivalent of a NullPointerException

Null safety

- In Kotlin, null safety ensures that variables cannot hold null values by default
- a variable must explicitly be declared as nullable by adding a **?** after the type (e.g., String**?**).
- Allow null-pointer exceptions using the **!!** operator
- You can test for null using the elvis operator (**?:**)
- The **let {}** function is a powerful tool for handling nullable variables in a safe way. It executes a block of code only if the

Safe Call Operator (?.)

The **?.** operator allows you to safely access properties or call methods on a nullable object. If the object is null, the expression evaluates to null instead of throwing an exception. If the object is not null, the method or property is accessed as usual.

```
    val name: String? = null

// Safe call, returns null without crashing
    println(name?.length)  // Prints "null"
```

Elvis Operator (?:)

The Elvis operator (**?:**) is used to provide a default value in case the expression to its left is null. It's named after Elvis Presley because the ?: symbol resembles a sideways Elvis hairstyle!

```
val name: String? = null

// Elvis operator provides a default value if name is null
val length = name?.length ?: 0  // If name is null, use 0 as default
println(length)  // Prints "0"
```

The name "Elvis operator" refers to the fact that when its common notation, ?: , is viewed sideways, it resembles an emoticon of Elvis Presley with his signature hairstyle.

Not-null assertion operator !!

The not-null assertion operator !! converts any value to a non-nullable type.

```
val name: String? = null

println(name!!.length)  // Using !! forces the null value, causing NPE

println(name?.length ?: "Unknown")  // Prints "Unknown" instead of causing an NPE
```

In this example, the !! operator is used to forcefully cast name to a non-null type. Since name is null, trying to access length causes a NullPointerException at runtime.

To avoid this, use safe calls (?.) or provide default values (?:), as demonstrated earlier.

Let { } Scope Function

The let function in Kotlin is a scope function that is primarily used for handling nullable values and executing a block of code only when a value is not null.

let is a type of scope function in Kotlin, which means it provides a temporary scope for the object it's called on. Within the block of let, the object is referred to as **it**.

```kotlin
val name: String? = "Kotlin"
// Code block that is executed if 'name' is not null
name?.let {
    println("The length of the name is: ${it.length}")
}
```

Note : It returns Unit (if nothing is returned explicitly) or any other value depending on the lambda body.

In Summary

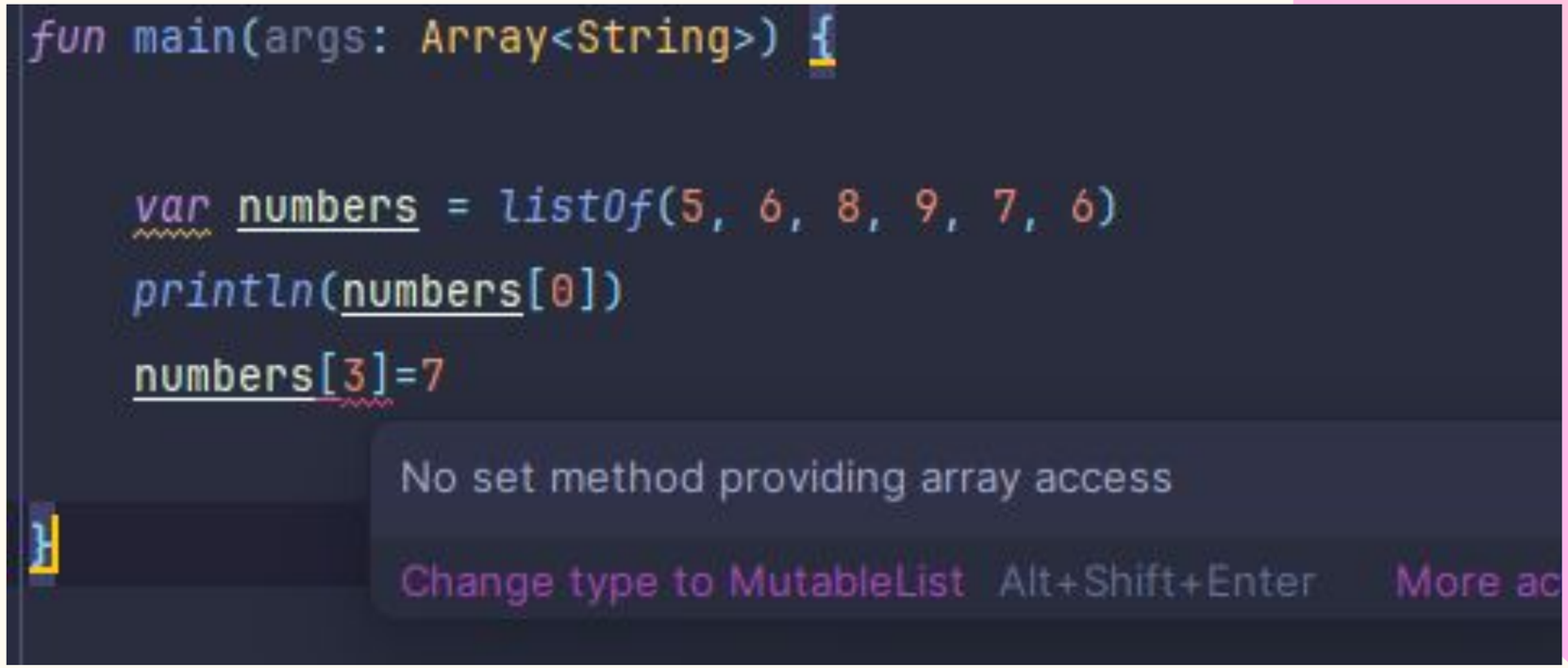
- Nullable types (?): Declare variables that can hold null values.
- Safe call operator (?.): Access properties or methods safely, returning null if the object is null.
- Elvis operator (?:): Provide a default value when the object is null.
- Non-null assertion (!!): Force a nullable variable to be non-null, throwing an exception if it is null.
- Safe casts (as?): Safely cast objects, returning null if the cast fails.
- Use let when you want to execute some logic only if a variable is not null.

```kotlin
fun main(){

    val name: String? = null

    // Safe call operator (?.) - safely accessing properties
    val length: Int? = name?.length
    println("Length: $length")  // Output: Length: null

    // Elvis operator (?:) - providing a default value
    val safeLength = name?.length ?: 0
    println("Safe Length: $safeLength")  // Output: Safe Length: 0

    // Non-null assertion ( !! ) - forces the variable to be non-null
    try {
        println(name!!.length)  // Throws NullPointerException
    } catch (e: NullPointerException) {
        println("Caught NullPointerException!")
    }

    // Safe casts (as?) - safe casting returns null if the cast fails
    val obj: Any = "Kotlin"
    val str: String? = obj as? String  // Safe cast
    val int: Int? = obj as? Int  // This will return null
    println("String cast: $str")  // Output: String cast: Kotlin
    println("Int cast: $int")  // Output: Int cast: null
```

COLLECTIONS

List

- Ordered collection: Elements are stored in a specific order.
- Duplicate elements: You can have multiple instances of the same element
- Read only you can’t change its size or any element of it



Mutable List

- Ordered collection: Elements are stored in a specific order.
- Duplicate elements allowed: You can have multiple instances of the same element.
- Mutable: Elements can be added, removed, or modified.

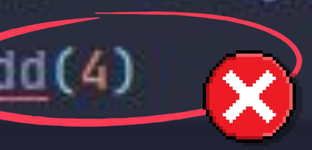
```
fun main(args: Array<String>) {


    var numbers = mutableListOf(5, 6, 8, 9, 7, 6)
    println(numbers[0])
    numbers[3]=7
    numbers.add(10)
```

Set

- Unordered collection: Elements are not stored in a specific order.
- No duplicate elements: Only one instance of each element is allowed.
- Read only
- to make any modify make it mutibleSet

Set & Mutable Set

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

    val numbers = setOf(1,2,3)
    numbers.add(4)
```



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

    val numbers = mutableSetOf(1,2,3)
    numbers.add(4)
```

Map

- Key-value pairs: Stores pairs of elements, where each key is associated with a value.
- Unique keys: Each key must be unique
- Read only map

Mutable Map

- Key-value pairs: Stores pairs of elements, where each key is associated with a value.
- Unique keys: Each key must be unique.
- Mutable: Key-value pairs can be added, removed, or modified.

Map & Mutable Map

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

    val numbers = mapOf(0 to "Zero",1 to "one")
    numbers[0] = "Two"

}
```

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

    val numbers = mutableMapOf(0 to "Zero",1 to "one")
    numbers[0] = "Two"
    numbers.put(4,"four")

}
```

	List	Set	Map (Dictionary)
Description	Ordered collection with access to elements by indices – integer numbers that reflect their position.	stores unique elements; their order is generally undefined.	set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates.
Order	Order is important	Undefined	Order isn't important
Accessing	Accessed via indices	Accessed via element	Accessed via keys
Uniqueness	Elements can occur more than once in a list	Unique	Unique Keys Can contain repeated values
Contains null	Yes and can be duplicated	Can contain only one null	Can contain only one null as a key
Equality	Lists are considered equal if they have the same sizes and structurally equal elements at the same positions	sets are considered equal if they have the same size, and for each element of a set there is an equal element in the other set	Two maps are considered equal if the containing pairs are equal regardless of the pair order.
Default Implementation	ArrayList	HashSet, LinkedHashSet	HashMap, LinkedHashMap

List operations

Operation	Function	Example
Retrieve elements by index	elementAt(), first(), last(), get(), getOrNull()	val numbers = listOf(1, 2, 3, 4) println(numbers.getOrNull(5))
Retrieve list parts	subList()	val numbers = (0..13).toList() println(numbers.subList(3, 6))
Find element positions	indexOf(number) lastIndexOf(number)	val numbers = listOf(1, 2, 3, 4, 2, 5) println(numbers.indexOf(2)) println(numbers.lastIndexOf(2))
List write operations	add(), set(), fill(), removeAt(), sort(), sortBy(), reverse(), shuffle()	val numbers = mutableListOf("one", "five") numbers.add(1, "two") numbers.addAll(2, listOf("three", "four"))

Thank You