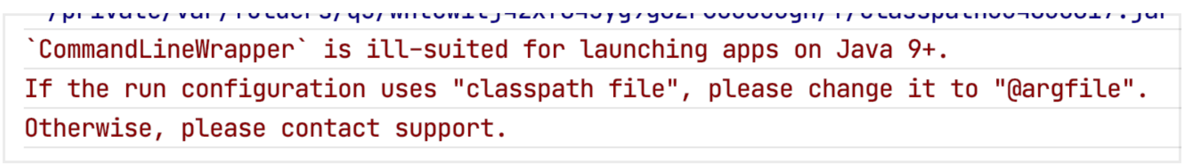
Kotlin Basics

Download IntelliJ IDEA from here (if the link is invalid just type “IntelliJ download” in google):
https://www.jetbrains.com/idea/download/
(Choose Community)

We will use Android Studio Kotlin REPL (Read Evaluate Print-loop) [tools -> Kotlin -> REPL]
Problem with opening REPL Run > Edit Configurations... > Templates > Java Scratch > Shorten command line to @argfile (Java 9+) and restart Android Studio.



```
`CommandLineWrapper` is ill-suited for launching apps on Java 9+.
If the run configuration uses "classpath file", please change it to "@argfile".
Otherwise, please contact support.
```

var - variables
val - finals
val always preferred

No need for semicolon
Kotlin is type inferred and type safety
Meaning this won’t work:
var x = 7
 x = 7.4

Kotlin has no primitives only objects (Int, Float, String)

consts are compile time constants. Meaning that their value has to be assigned during compile time, unlike vals, where it can be done at runtime. This means, that consts can never be assigned to a function or any class constructor, and only to a String or primitive.

const val WEBSITE_NAME = "Baeldung"
the Kotlin compiler inlines the const val values into the locations where they’re used

If you try to type it in the REPL you will get the following error: “const 'val' are only allowed on top level or in objects”, like this for example:

const val VALUE: String = "constant"

```
fun main() {
   println("$VALUE is inlined")
}
```

At first glance, we might think that the Kotlin compiler gets a static field value from a class and then concatenates it with the ″ is inlined″ text. However, since const vals are inlined, the compiler will copy the “constant” literal wherever the VALUE constant is used. This is why they must get a value in compile time. **This constant inlining is much more efficient than getting a static value from a class**.

Kotlin Nullability

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

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). Regular objects cannot be null (String, Int…) - prevent null pointer exception

The main advantage is that If we check a property of null objects we get a compile time and not runtime error
To bypass the compile time check use !! - but be careful from KotlinNullPointerException

If we are not sure we use - safe call - use ?. - str?.length - > will return null and not crash

When inferring types, the compiler assumes non-null for variables that are initialized with a value.
var inferredNonNull = "The compiler assumes non-null"
If we assign null on initialization the ? Type will be auto inferred

class **Nothing -** Nothing has no instances. You can use Nothing to represent "a value that never exists": for example, if a function has the return type of Nothing, it means that it never returns (always throws an exception).

There are a few techniques for using or accessing the nullable variable. One of them is safe call ?. and another one is null check !! but before figuring out the difference among both, let’s understand what they are in detail first:

```
+------------+---------------------+---------------------+-----------------------+
| a: String? |            a.length |           a?.length |           a!!.length |
+------------+---------------------+---------------------+-----------------------+
|      "cat" | Compile time error |                   3 |                     3 |
|       null | Compile time error |                null | NullPointerException |
+------------+---------------------+---------------------+-----------------------+
```

?. - safe call

The best way to access nullable property is safe call operator ?.
This calls the method if the property is not null or returns null if that property is null without throwing an NPE (null pointer exception).
Safe calls are useful in chains. For example, if Bob, an Employee, may be assigned to a Department (or not), that in turn may have another Employee as a department head, then to obtain the name of Bob's department head (if any), we write the following
bob?.department?.head?.name
Such a chain returns null if any of the properties in it is null.
Since the return value is null we can combine this operator with **let** function we will see later on

The !! Operator

This operator is used to explicitly tell the compiler that the property is not null and if it's null, please throw a null pointer exception (NPE). If you are sure that the property value is not null use ?. instead of !!. Usually we use this when passing a must non null value to a function. But again be careful while using it.

Elvis Operator (?:)

This one is similar to safe calls except the fact that **it can return a non-null value if the calling property is null**
The Elvis operator will evaluate the left expression and will return it if it's not null or else will evaluate the right side expression. Please note that the right side expression will only be evaluated if the left side expression is null.
Elvis is usually used to give default values in case of null:

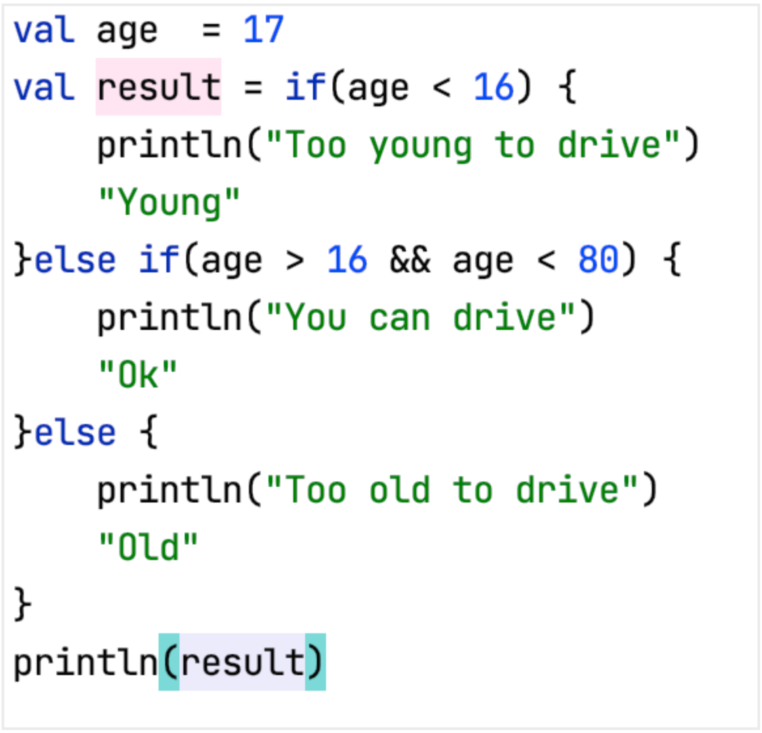
val str : String? = null
val strLength = str?.length ?: -1 - > strLength equals to -1

We can always preform an explicit null check if(x != null) x.do()
In this case if x is val or local property he will be smart casted to non-nullable and all is safe. But if it is a var class property the compiler won't smart cast him and if you do it by force you will get the following error: "smart cast to '['non null type] is impossible, because '['variable name] is a mutable property that could have been changed by this time" - try to always use vals

If and when

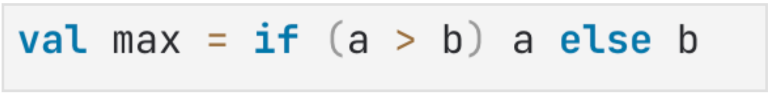
if syntax like Java but each block can return it's last line - if is not a statement but an expression!

Branches of if can be blocks. In this case, the last expression is the value of a block
'if' must have both main and 'else' branches if used as an expression
println - prints on screen
Unit - Kotlin’s void



```
val age  = 17
val result = if(age < 16) {
    println("Too young to drive")
    "Young"
}else if(age > 16 && age < 80) {
    println("You can drive")
    "Ok"
}else {
    println("Too old to drive")
    "Old"
}
println(result)
```

In Kotlin, if is an expression: it returns a value. Therefore, there is no ternary operator (condition ? then : else) because ordinary if works fine in this role:



```
val max = if (a > b) a else b
```

when expression

Similar to switch but No need for case just write the value and **->** if it is more then one line use blocks
Use **in** or **!in** for ranges - Kotlin lets you easily create ranges of values using the rangeTo() function from the kotlin.ranges package and its operator form ..
Usually, rangeTo() is complemented by in or !in functions.
https://kotlinlang.org/docs/operator-overloading.html#equality-and-inequality-operators

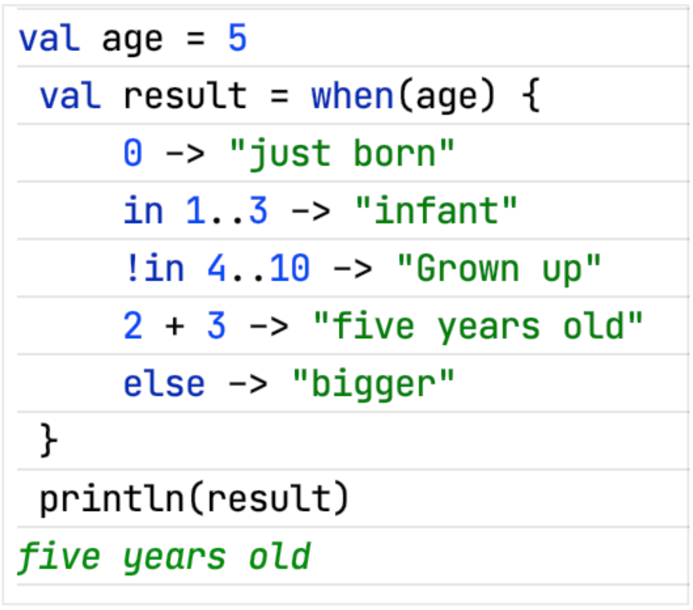
Binary operations

Arithmetic operators

Expression	Translated to
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`
`a..b`	`a.rangeTo(b)`

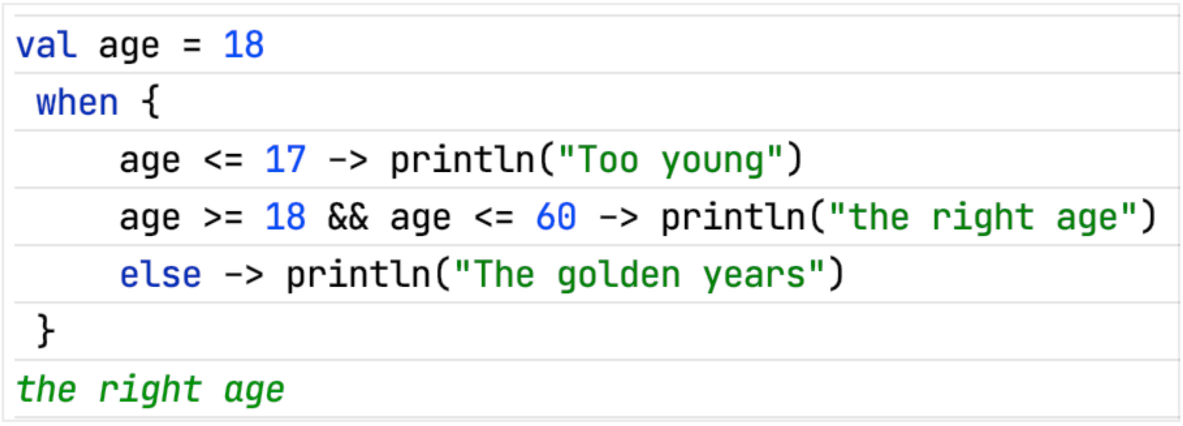
Expression	Translated to
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

No need for **break** - won't got into the next case even without break
Instead of **default** use **else**
You can even call functions or operators in the cases
Same as if it can return values

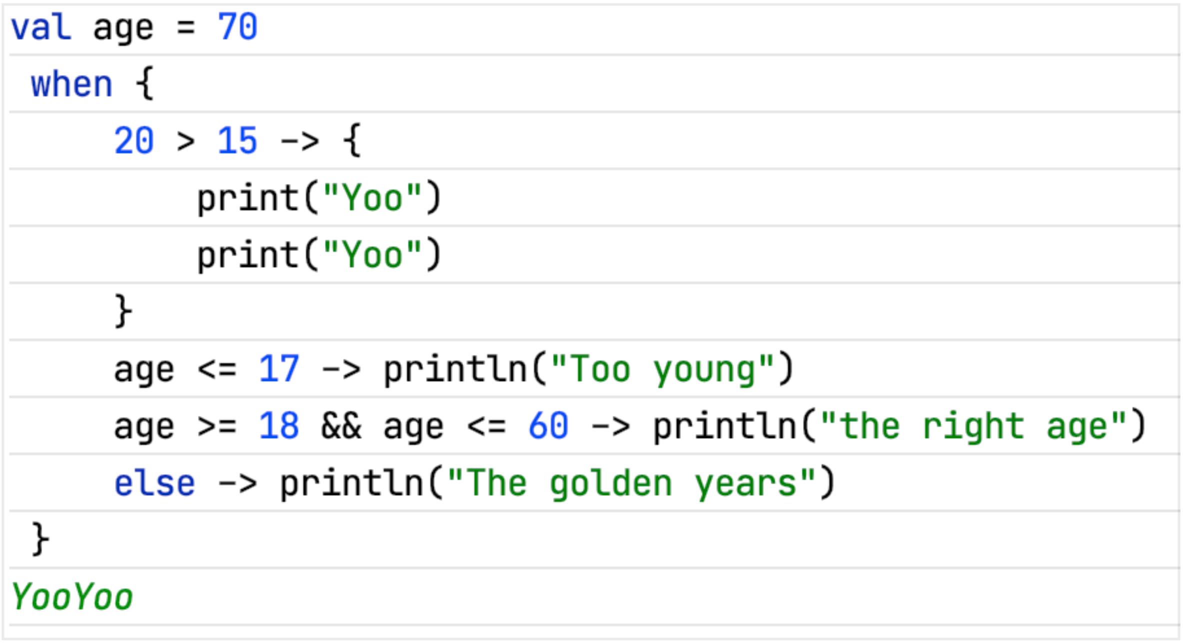


```
val age = 5
 val result = when(age) {
     0 -> "just born"
     in 1..3 -> "infant"
     !in 4..10 -> "Grown up"
     2 + 3 -> "five years old"
     else -> "bigger"
 }
 println(result)
five years old
```

You can use when without the variable name
And not even on the same variable! It is just a collection of boolean conditions
You can use {} inside when for multiple code lines

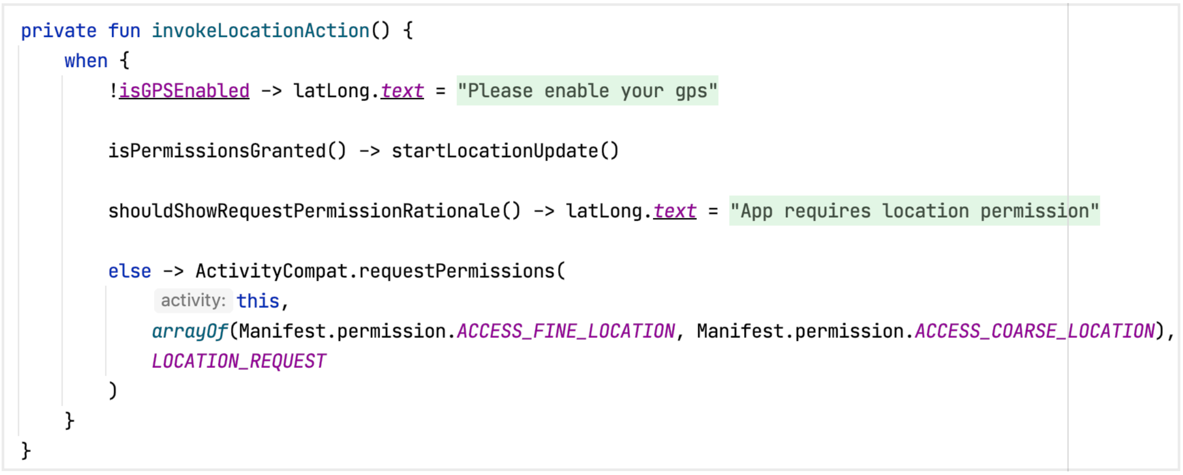


```
val age = 18
 when {
     age <= 17 -> println("Too young")
     age >= 18 && age <= 60 -> println("the right age")
     else -> println("The golden years")
 }
the right age
```



```
val age = 70
 when {
     20 > 15 -> {
         print("Yoo")
         print("Yoo")
     }
     age <= 17 -> println("Too young")
     age >= 18 && age <= 60 -> println("the right age")
     else -> println("The golden years")
 }
YooYoo
```

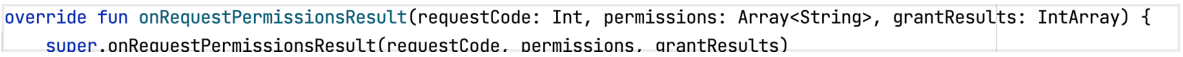
The advantage is that if one is true then the rest won’t be checked - think when you want to start the app in condition that all permission granted and all features enabled and so on, with when you can check all of them in a single block of code



```
private fun invokeLocationAction() {
    when {
        !isGPSEnabled -> latLong.text = "Please enable your gps"

        isPermissionsGranted() -> startLocationUpdate()

        shouldShowRequestPermissionRationale() -> latLong.text = "App requires location permission"

        else -> ActivityCompat.requestPermissions(
            activity: this,
            arrayOf(Manifest.permission.ACCESS_FINE_LOCATION, Manifest.permission.ACCESS_COARSE_LOCATION),
            LOCATION_REQUEST
        )
    }
}
```

Arrays

val array = **arraOf**(….) -> array of objects Kotlin uses Array<Int>
val array: Array<Long> = arrayOf(1,2,3)
joinToString() - > returns string representation of the array (toString just return the instance address).
val result = arrayOf(1,4,6,8,9)
result.joinToString()
res54: kotlin.String = 1, 4, 6, 8, 9
array[0]
Kotlin provides a selection of classes that become primitive arrays when compiled down to JVM byte-code.
You can create an array of java primitives with intArrayOf, charArrayOf….
You can sometimes find them in the Android API:



```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
```

Kotlin.collections

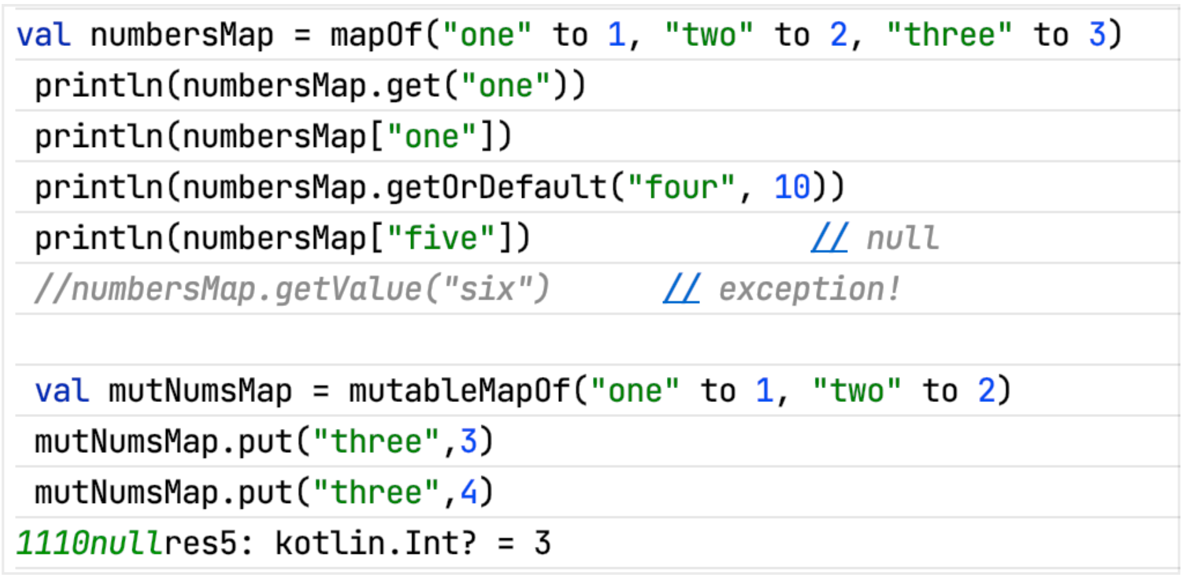
List\MutableList(the regular List of java)
val list = **listOf**(…)
val mutList = mutableListOf(6,8,7)
mutList.add(7)
list[0] = 90 - > error when its not a mutable list
Please note the Kotlin's list not only can't change it's size but also it's content

val set = **setOf**(6,6) only one will stay

val map = **mapOf**(Pair(1,"Moshe))
To add to the collections use the Mutable(mutable list, set, map…)
map's put function returns the overridden value
Instead of Pair your can use **to** infix function — more on infix very soon - but in short there is **to** infix extension function to most of the classes that accept another argument and return a pair with both of them and replaces the dot notation

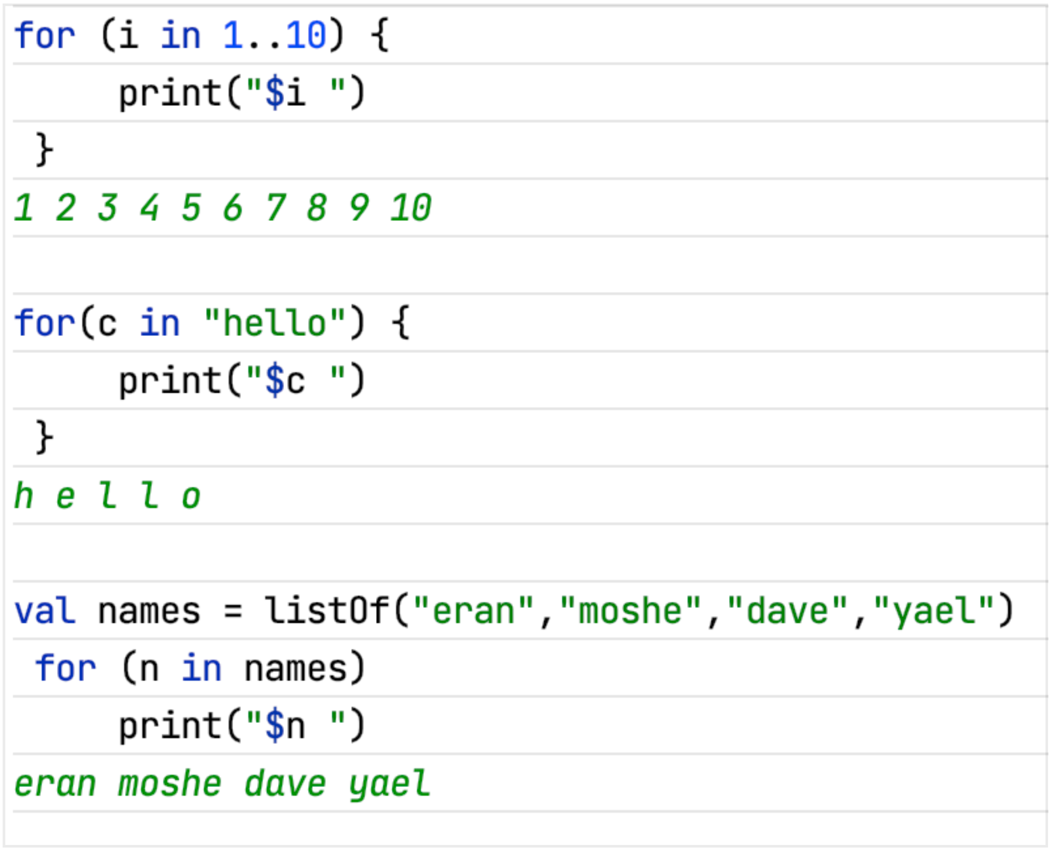
For retrieving a value from a map, you must provide its key as an argument of the get() function. The shorthand [key] syntax is also supported. If the given key is not found, it returns null. There is also the function getValue() throws an exception if the key is not found in the map - if we must pass a non null reference. Additionally, you have two more options to handle the key absence:

- getOrDefault() returns the specified default value if the key is not found.
- getOrElse() the values for non-existent keys are returned from the given lambda function (more on lambda in the next chapter).



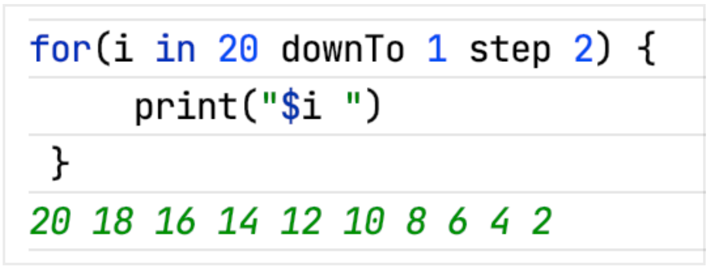
```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
 println(numbersMap.get("one"))
 println(numbersMap["one"])
 println(numbersMap.getOrDefault("four", 10))
 println(numbersMap["five"])                 // null
 //numbersMap.getValue("six")        // exception!

 val mutNumsMap = mutableMapOf("one" to 1, "two" to 2)
 mutNumsMap.put("three",3)
 mutNumsMap.put("three",4)
1110nullres5: kotlin.Int? = 3
```

Loops



```
for (i in 1..10) {
     print("$i ")
 }
1 2 3 4 5 6 7 8 9 10

for(c in "hello") {
     print("$c ")
 }
h e l l o

val names = listOf("eran","moshe","dave","yael")
 for (n in names)
     print("$n ")
eran moshe dave yael
```

downTo - inline function for descending order
step - for creating spaces



```
for(i in 20 downTo 1 step 2) {
     print("$i ")
 }
20 18 16 14 12 10 8 6 4 2
```

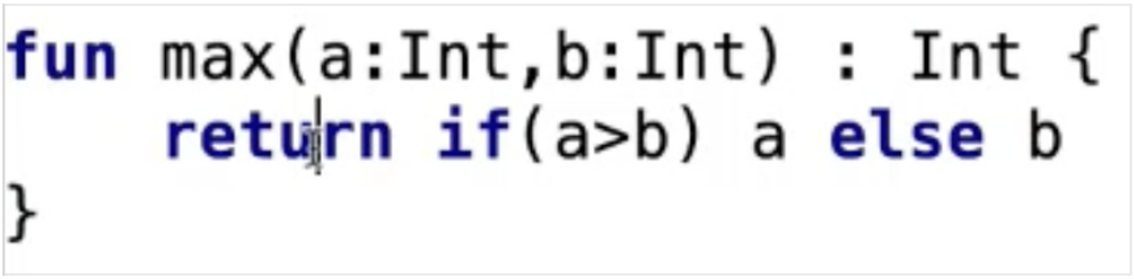
Once int become Int we get these infix functions and much more
while, do-while - same as Java

Functions

fun - keyword for function declaration

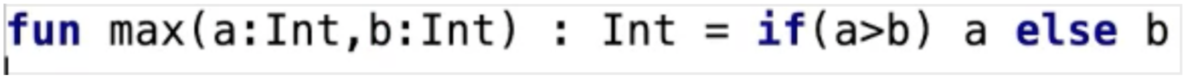
fun [name](params) : [return value]

If no return type is specified then Unit (Java's void) is omitted - unless it is a single line function and the return value is inferred



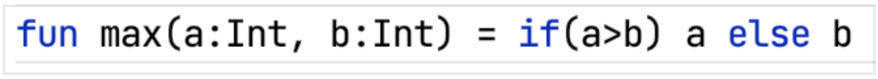
```
fun max(a:Int,b:Int) : Int {
    return if(a>b) a else b
}
```

If the function is only one line no need for {} just use =



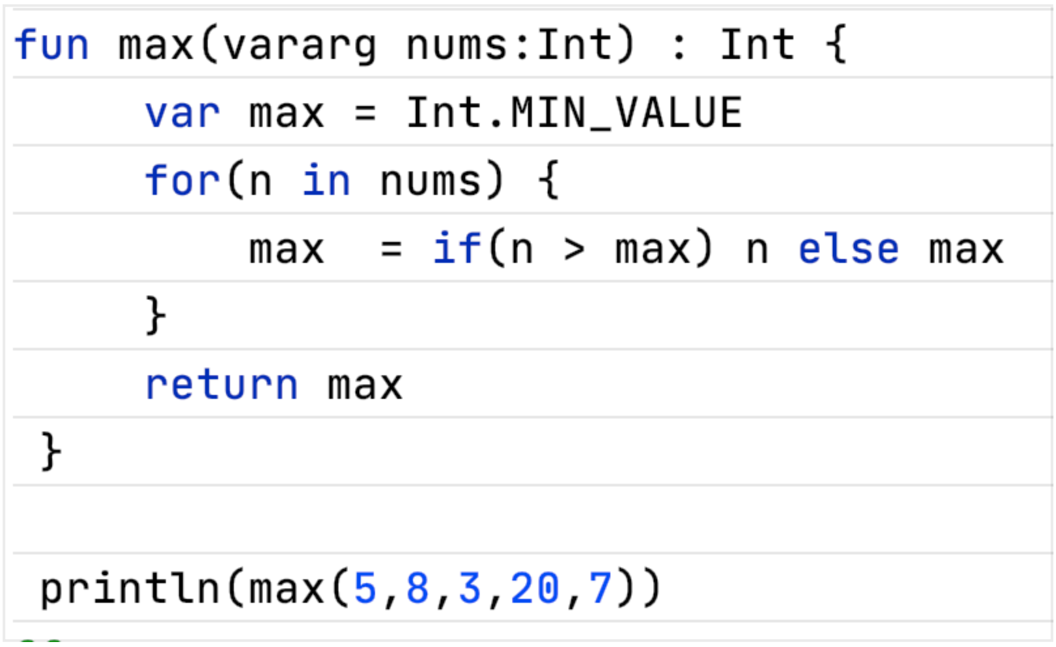
```
fun max(a:Int,b:Int) : Int = if(a>b) a else b
```

And in that case the return type can be inferred!



```
fun max(a:Int, b:Int) = if(a>b) a else b
```

varag - for unknown number of params



```
fun max(vararg nums:Int) : Int {
     var max = Int.MIN_VALUE
     for(n in nums) {
         max  = if(n > max) n else max
     }
     return max
 }

 println(max(5,8,3,20,7))
```

Move to IntelliJ IDEA create a new project, make sure your JDK is defined and in it create a new Kotlin file.

Kotlin enables top class variables and functions - no need for class to write functions
Kotlin also allows us to define inner functions - function nested inside other functions

fun main(args:Array<String>) {} [you can shorten by writing main followed by enter]
Print your first Hello World in Kotlin!!!

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

Create infix functions
Functions marked with the infix keyword can also be called using the infix notation (omitting the dot and the parentheses for the call). Infix functions must meet the following requirements:

- They must be member functions or extension functions - they must have this.
- They must have a single parameter.
- The parameter must not accept variable number of arguments (varargs) and must have no default value.

Example(an Int extenssion function that concatenating a string to himself a given times

```kotlin
infix fun Int.times(string: String) = string.repeat( n: this)
```

Usage:

```kotlin
print(7 times "Mush")
```

Infix notation also works on members functions (methods):

```kotlin
class Person(val name: String) {
    private val likedPeople = mutableListOf<Person>()
    infix fun likes(other: Person) { likedPeople.add(other) }
}

fun main() {
    val sophia = Person( name: "Sophia")
    val claudia = Person( name: "Claudia")
    sophia likes claudia
}
```

Please note: We will discuss classes later but for now notice that there is no new in Kotlin just the class name followed by it's constructor call.

Look at the smart cast we have seen before in IntelliJ and functions.
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 - in the following example maybeString is smart casted to String

```
fun describeString(maybeString: String?): String {
    if (maybeString != null && maybeString.length > 0) {
        return "String of length ${maybeString.length}"
    } else {
        return "Empty or null string"
    }
}
```

Default arguments

Function parameters can have default values, which are used when you want to skip the corresponding argument. This significantly reduces the number of overloads and saves us ALLOT of code.

A default value is defined using = after the type.

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

Think about how many lines of code you just saved (please note that ByteArray is java array of bytes - it is not Array<Byte>)

```
val b = byteArrayOf()
read(b)
read(b, off: 100)
read(b,len = 100)
read(b, off: 100, len: 100)
```

If a default parameter precedes a parameter with no default value, the default value can only be used by calling the function with named arguments:

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }

foo(baz = 1) // The default value bar = 0 is used
```

We usually put the parameters with default values after parameter with no default values(no save for the need of using parameters names)

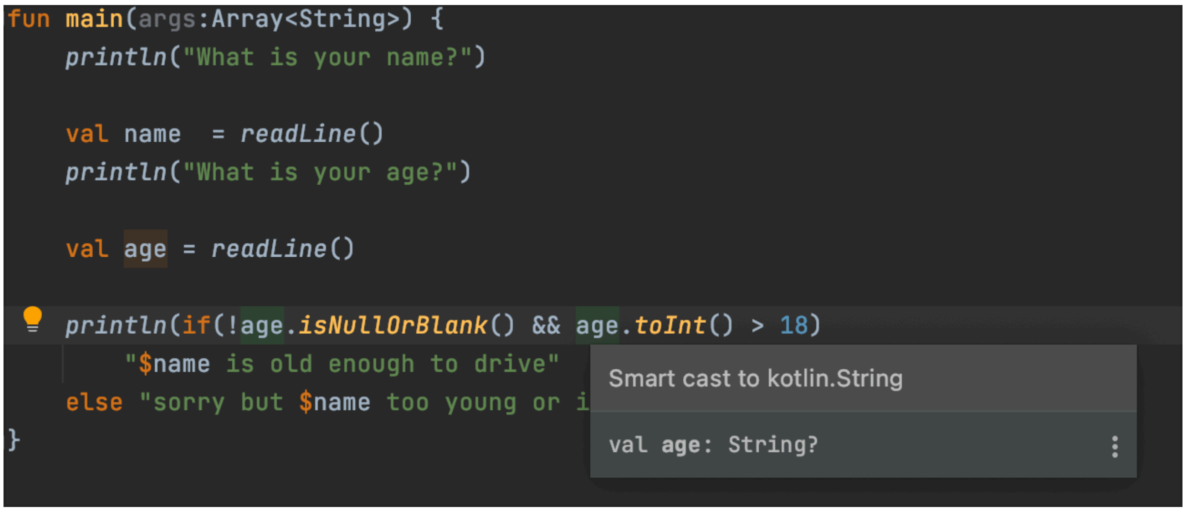
Please note that with default values we still have to write the type. The main reason is compilation performance. When we look at a method call, it helps a lot to have explicit parameter types which don't need to be inferred. Inferring the parameter type from a string constant is trivial, but if you have fun foo(a = bar()) and bar() also has an inferred return type, understanding the actual parameter type of a becomes very expensive.

Drills

1. Ask the user for his name and his age and print out if he is old enough to drive (please use the [String].toInt() function but be aware that it works on String and not String? - what that the redaLine() returns)

Spoilers:
readLine() returns String?
toInt() can be applied only to String
You must use ? Or !!
While ? Can return null you can't use it in a boolean expression (can't compare null)
So the only option is to use !! but be sure to check before using
If you check before then you don't need !! Kotlin will automatically smart cast your object to String before applying the toInt() function

Solution:



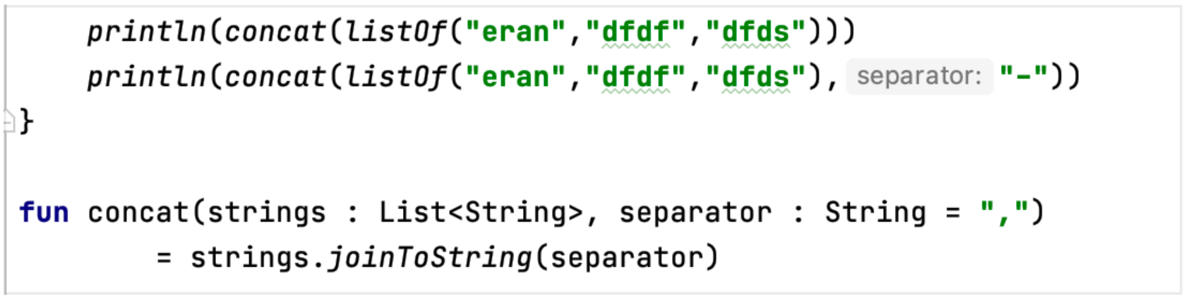
```
fun main(args:Array<String>) {
    println("What is your name?")

    val name  = readLine()
    println("What is your age?")

    val age = readLine()

    println(if(!age.isNullOrBlank() && age.toInt() > 18)
        "$name is old enough to drive"
    else "sorry but $name too young or i
}
```

Smart cast to kotlin.String

```
val age: String?
```

2. Create a concat function that receives a list of strings and a separator and return one string containing the strings separated with the separator. If no separator supplied use comma. invoke her twice, once with the default separator and one with you own.

Solution:



```
    println(concat(listOf("eran","dfdf","dfds")))
    println(concat(listOf("eran","dfdf","dfds"), separator: "-"))
}

fun concat(strings : List<String>, separator : String = ",")
        = strings.joinToString(separator)
```

2 - Bonus - do you think you can supply both parameter but in a different order without changing the function title?

Solution(use parameters names):
val result1 = *concat*(separator = **"-"**,list = *listOf*(**"eran"**,**"moshe"**,**"dave"**))

3. Function syntax - Create the following functions:

- A simple function that takes a parameter of type String and returns Unit.
- A function that takes two strings message and a prefix. the second parameter is optional with default value “Info”. The function will not return anything but this time use omitted Unit return value and print to the screen the prefix followed by the message.
- A function that receives two integers and returns their sum.
- A single-expression function that returns an integer (inferred) - the function will receives two Int and returns their multiplication.
- A function that takes String varargs and prints them

- Infix function called “onto” that works on two strings (this and the parameter and return a new Pair containing both of them) - Pairs can be add to maps like we have seen before. use your infix function with map initialization.
- Create the main function and test all your functions

Solutions:

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

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

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

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

```kotlin
fun printAll(vararg messages: String) {
    for (m in messages) println(m)
}
printAll("Hello", "Hallo", "Salut", "Hola", "你好")
```

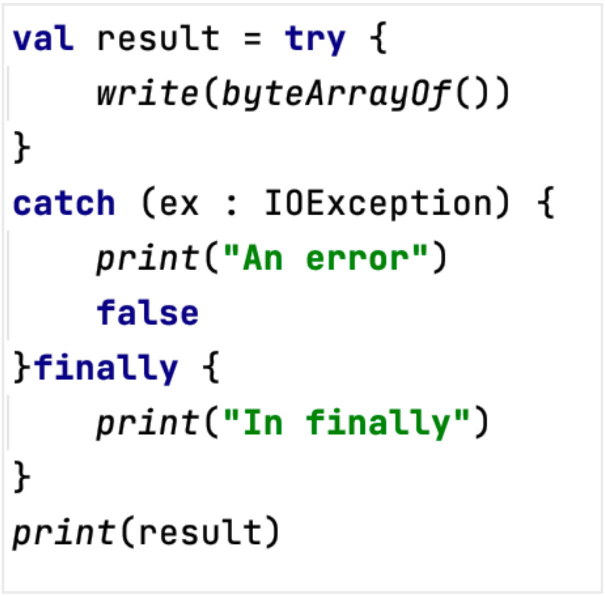
```kotlin
infix fun String.onto(other: String) = Pair(this, other)
val myPair = "McLaren" onto "Lucas"
```

For more reasons adding on basic functions syntax
https://kotlinlang.org/docs/functions.html

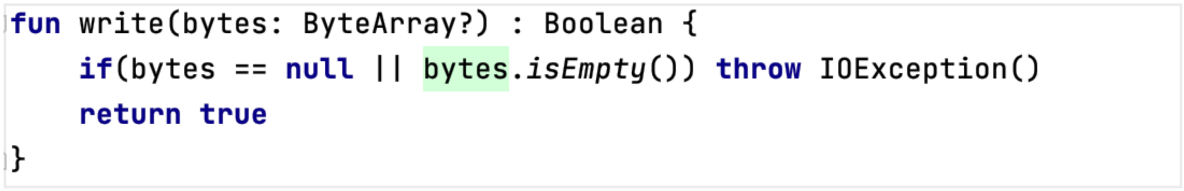
Exceptions

Kotlin solves a very common problem with try-catch block and variable scopes: In java we sometimes have to define a variable outside the try block and initialize it to null, we can’t define it inside the try block because of the scope - if we define him in the try block then it wouldn’t exist outside of it.

Kotlin solve this by with a try - catch block that is also an expression and thus return a value, the last line in the block is the retuned value :



```
val result = try {
    write(byteArrayOf())
}
catch (ex : IOException) {
    print("An error")
    false
}finally {
    print("In finally")
}
print(result)
```



```
fun write(bytes: ByteArray?) : Boolean {
    if(bytes == null || bytes.isEmpty()) throw IOException()
    return true
}
```

Please note the we can throw IOException without surrounding it with try catch block or declare it in the method constructor!

This is not possible in Java cause IOException is a checked exception but **in Kotlin There are no checked exceptions!! like C# and Ruby** All is un-checked exceptions meaning it's your responsibility!
Thats why the keyword throws does not exists in Kotlin

Be responsible! Kotlin is more interested in saving us lines of code then in being our Mom and Dad :)