JET BRAINS Kotlin

Functional Programming

@kotlin | Developed by JetBrains

What is it?

We are already familiar with **object-oriented programming (OOP)**,

but Kotlin also borrows concepts from **functional programming (FP)**.

FP is a programming paradigm where programs are constructed by **applying** and **composing functions**.

```
var sum = 0
for (item in list) {
   if (item > 0) {
       sum += item * item
   }
}
```

VS

```
list.filter { it > 0 }.map { it * it }.sum()
```

Introduction

In Kotlin, functions are considered first-class citizens.

As you saw, it’s not mandatory to define them within a class; they can be written within the file.

Lambda expressions and **anonymous functions** are ***function literals***.

Function literals are functions that are not declared but are passed immediately as an expression.

Lambda expressions

The full syntactic form of lambda expressions is as follows:

```
val lambdaIdentifier = { inputs          → operation and output }
val sum              = { x: Int, y: Int → x + y }
```

A lambda expression is **always** surrounded by curly braces.

Parameter declarations in the full syntactic form go inside curly braces.

The body goes after the ->

If the inferred return type of the lambda is not Unit,

the last (or possibly single) expression inside the lambda body is treated as the return value.

Lambda Expressions

Invoking lambda expressions:

```
val sum = { x: Int, y: Int → x + y }

fun main(){
    var result = sum(1 , 2)
    result = sum.invoke(2 , 3)
}
```

Function Type

Since we were able to assign a functionality to a variable.
Then, that variable must have a type.

Hence, Kotlin uses **function types**.

All function types have a parenthesized list of parameter types and a return type

```
(A, B) → C
```

A type that represents functions that take two arguments of types A and B and return a value of type C. The list of parameter types may be empty, as in () → A.
The Unit return type **cannot** be omitted.

```
val sum : (Int, Int) → Int = { x: Int, y: Int → x + y }
```

Type Alias

You can also give a function type an alternative name by using a type alias:

```
typealias requiredName = ExistingType
```

```
typealias TwoIntType = (Int, Int) → Int
val sum : TwoIntType = { x: Int, y: Int → x + y }
```

Anonymous Function

The lambda expression syntax is missing the ability to specify the function's return type.

In most cases, this is unnecessary because the return type can be inferred automatically.
However, if you do need to specify it explicitly,
you can use an alternative syntax: an anonymous function.

```
val anonymousSum = fun(x1: Int, x2:Int): Int{
    return x1 + x2
}
```

Defining Anonymous function

```
fun main() {
    var result = anonymousSum(1, 2)
    result = anonymousSum.invoke(2, 3)
}
```

Calling Anonymous function

Functional Interfaces

An interface with only one abstract method is called a **functional interface**, or a **Single Abstract Method (SAM)** interface.

The functional interface can have several non-abstract members but only one abstract member.

To define a **functional interface** in Kotlin, we prefix the interface with the **fun** keyword

```
fun interface FunctionalInterfaceName {
    //Only ONE abstract method is allowed here
    fun myAbstractMethod()
    fun myConcreteMethod() {
    }
}
```

Functional interfaces (SAM)

Single Abstract Method (SAM) interface

- Interface that has **one abstract method**.
- Kotlin allows us to use a lambda instead of a class definition to implement a SAM.

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}
```

```
val isEven = object : IntPredicate {
   override fun accept(i: Int): Boolean {
       return i % 2 == 0
   }
}
```

VS

```
val isEven = IntPredicate { i ->
          i % 2 == 0
}
```

```
fun main() {
   println("Is 7 even? - ${isEven.accept(7)}")
}
```

Higher order functions (HOFs)

Functions that take other functions as **arguments** are called **higher order functions**.

In Kotlin, you frequently encounter them when working with collections:

```
list.partition { it % 2 == 0 }  OR  list.partition { x → x % 2 == 0 }
```

Everything Kotlin allows you do with functions,
which means that “functions in Kotlin are first-class citizens.”

Higher order functions (HOFs)

```
fun calculate(num1: Int, num2:Int, operation: (Int, Int)→ Int): Int{
    val result = operation.invoke(num1, num2)
    return result
}
val sum = { x: Int, y: Int → x + y }

fun main() {
    var result = calculate(1, 2, sum)
}
```

We can pass a different implementation to the operation that applies the rules of having two inputs of the Int type and returns

Trailing Lambda

According to Kotlin Convention when the last parameter of a function is a lambda then it can be placed outside the parentheses.

```
public fun repeat(times: Int, action: (Int) → Unit) {
```

```
repeat(5, { println("Hello") })
```

```
//or
```

```
repeat(5){ println("Hello") }
```

Higher order functions (HOFs)

Often in the context of FP, it is necessary to operate with the following functions: **`map`**, **`filter`**, and **`fold`**.

`map` allows us to perform a function over *each* element in a collection.

```
val list = listOf(1, 2, 3)
list.map { it * it } // [1, 4, 9]
```

Higher order functions (HOFs)

`filter` returns a list containing only elements that match a given predicate:

```
val list = listOf(1, 2, 3)
list.filter { it % 2 == 0 } // [2]
```

Higher order functions (HOFs)

Lambdas are not the only functions that can be passed as arguments to functions expecting other functions, as ***references*** to already defined functions can be as well:

```
fun isEven(x: Int) = x % 2 == 0

val isEvenLambda = { x: Int → x % 2 == 0 }
```

Same results, different calls:

- `list.partition { it % 2 == 0 }`
- `list.partition(::isEven)` `// function reference`
- `list.partition(isEvenLambda)` `// pass lambda by name`

Lazy Properties

`lazy()` is a function that takes a lambda and returns an instance of `Lazy<T>`, which can serve as a **delegate for implementing a lazy property**.

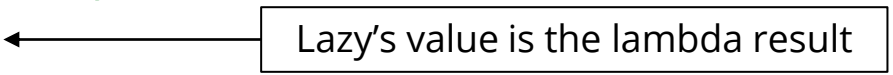
The first call to `get()`
executes the lambda passed to `lazy()` and remembers the result.

Subsequent calls to `get()` simply return the remembered result.

Lazy Properties

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```



Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())


    [One, one, was, a, race, horse, , Two, two, was, one, too, , One, one, won,
    one, race, , Two, two, won, one, too, , ]
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())
    .filter { it.isNotEmpty() }

    [One, one, was, a, race, horse, Two, two, was, one, too, One, one, won,
    one, race, Two, two, won, one, too]
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())
    .filter { it.isNotEmpty() }
    .map { it.lowercase() }


    [one, one, was, a, race, horse, two, two, was, one, too, one, one,
    won, one, race, two, two, won, one, too]
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())
    .filter { it.isNotEmpty() }
    .map { it.lowercase() }
    .groupingBy { it }
    .eachCount()
```

OR

```
string
    .split(...)
    .filter { it.isNotEmpty() }
    .groupBy({ it.lowercase() }, { it })
    .mapValues { (key, value) →
        value.size
    }
```

```
{one=7, was=2, a=1, race=2, horse=1, two=4, too=2, won=2}
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())
    .filter { it.isNotEmpty() }
    .map { it.lowercase() }
    .groupingBy { it }
    .eachCount()
    .toList()

    [(one, 7), (was, 2), (a, 1), (race, 2), (horse, 1), (two, 4),
    (too, 2), (won, 2)]
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())
    .filter { it.isNotEmpty() }
    .map { it.lowercase() }
    .groupingBy { it }
    .eachCount()
    .toList()
    .sortedBy { (_, count) -> count }

    [(a, 1), (horse, 1), (was, 2), (race, 2), (too, 2), (won, 2), (two, 4), (one, 7)]
```

Higher order functions (HOFs)

```
val string = """
    One-one was a race horse.
    Two-two was one too.
    One-one won one race.
    Two-two won one too.
""".trimIndent()

val result = string
    .split(" ", "-", ".", System.lineSeparator())
    .filter { it.isNotEmpty() }
    .map { it.lowercase() }
    .groupingBy { it }
    .eachCount()
    .toList()
    .sortedBy { (_, count) -> count }
    .reversed()
```

```
[(one, 7), (two, 4), (won, 2), (too, 2), (race,
2), (was, 2), (horse, 1), (a, 1)]
```

OR

```
string
    .allFunnyFuncs(...)
    .toList()
    .sortedWith { l, r ->
        r.second - l.second
    }
```

OR

```
string
    .allFunnyFuncs(...)
    .toList()
    .sortedByDescending { (_, c) ->
        c
    }
```

Labs

- Create a `calculate()` that accepts two integers and an operation as a function type. Let the calculate method support:
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Calculating the power

Thank You