BASICS OF KOTLIN

working with collections, generics and delegates

Miloš Vasić,
29.03.2017.

Miloš Vasić

- Software engineer at Robert Bosch d.o.o.
- Author of Fundamental Kotlin and open source enthusiast.

- Website: milosvasic.net
- Book: fundamental-kotlin.com
- GitHub: github.com/milos85vasic
- E-Mail: milos85vasic@gmail.com

Kotlin user group Serbia

- https://twitter.com/kotlin_serbia
- https://www.facebook.com/kotlinserbia/
- https://www.meetup.com/Serbia-Kotlin-User-Group

Collections

Kotlin has two types of collections:

- mutable (we can change collection's content)
- immutable (we can't change collection's content)

Most frequently used collections are:

- lists
- maps
- sets

Collections

Differences between mutable and immutable collections are the easiest to explain on example of lists.

Immutable lists implement interface **List<out T>** which gives class the following functionalities: **size** i **get**.

Mutable functionalities are gained by implementing **MutableList<T>** interface, which gives **add**, **addAll** and **remove** functionalities.

Collections

Instantiation of immutable collections:

```
/**
 * Immutable lists
 */
val immutableList = listOf(2, 4, 6)
val immutableList2 = listOf("Some", "Word")

/**
 * Immutable maps
 */
val immutableMap = mapOf("something" to 1, "else" to 2)
val immutableMap2 = mapOf(Pair(1, "Plane"), Pair(2, "Car"))
```

Collections

```
/**
 * Immutable sets
 */
val immutableSet = setOf(2, 2, 3) // It only has members 2 and 3.
// Outputs 2 and 3 (:: meaning direct access to reference)
immutableSet.forEach(::println)
```

Collections

Instantiation of mutable collections:

```
/**
 * Mutable lists
 */
val mutableList = mutableListOf(2, 4, 6)
val mutableList2 = mutableListOf("Some", "Word")
val mutableList3 = mutableListOf<String>()

// We can also remove some members.
mutableList2.remove("Word")

// or add
mutableList3.add("Car")
mutableList2.add("Plane")
mutableList2.addAll(mutableList3)
```

Collections

```
/**
 * Mutable maps
 */
val mutableMap = mutableMapOf("something" to 1, "else" to 2)
val mutableMap2 = mutableMapOf(Pair(1, "Plane"), Pair(2, "Car"))


/**
 * Mutable sets
 */
 // It will have only 2 and 3 without any duplicates.
val mutableSet = mutableSetOf(2, 2, 3)
mutableSet.add(1)
mutableSet.add(4)
mutableSet.add(5)
mutableSet.add(55)
```

Collections

```
// We remove all members that are > 2
mutableSet.removeIf { x -> x > 2 }

// Outputs 2 and 1 (:: meaning direct access to reference)
mutableSet.forEach(::println)
```

Collections

Accessing to members:

```
// Access to element at first position:
val x = mutableList[0]

// We get element mapped to "something" key
val y = mutableMap["something"]
```

Collections

Iterating through collections:

```
mutableList.forEach { x -> doSomething(x) }
// or (conversion of lambda to reference)
mutableList.forEach(::doSomething)

// if we need index for each element:
mutableList.forEachIndexed {
        index, item -> doSomething(index, item)
}

// or we can apply filter if needed:
mutableList
    .filter { x -> x >= 4 }
    .forEach(::println)
```

Generics

- **Generic classes:**

```
/**
 * Simple class that can take anything and use it to print.
*  'in' means that this type can only be consumed
*  but never produced.
 */
class Printer<in T> {
        fun print(item: T) = println("Item [ $item ]")
}
```

Generics

```
/**
 * Builder that generates instances of T
 * based on parameter of P type.
 * 'out' means that this type can only be produced
 * but never consumed.
 */
abstract class Builder<in P, out T> {
    abstract fun build(param: P): T
}
```

Generics

```
/**
 * Builder realization
 */
class IntegerBuilder : Builder<String, Int>() {
    override fun build(param: String): Int {
        return param.toInt()
    }
}
```

Generics

```
/**
 * Class takes collection and exposes min and max values.
 * T is produced and consumed and there is no 'in' or 'out'.
* T in this example also must extend Comparable.
 */
class Sorter<T : Comparable<T>>(items: List<T>) {
  private val sorted = items.sorted()

  fun getMax(): T {
      return sorted.last()
  }

  fun getMin(): T {
      return sorted.first()
  }
}
```

Generics

- *val integerPrinter = Printer<Int>()*
  val stringPrinter = Printer<String>()

  integerPrinter.print(2)
  stringPrinter.print("Something")

- *// ---*

- *val intBuilder = IntegerBuilder()*
  val x = intBuilder.build("1")
  println("We build [$x]")

Generics

- *val list = listOf(2, 5, 1, 2, 6, 6, 8, 2, 1, 10, 3)*

- *// We will not instantiate it via: Sorter<Int>(list)*
  // since Kotlin takes type from arguments.
- *val sorter = Sorter(list)*

- *// Outputs: [1][10]*
- *println("[${sorter.getMin()}][${sorter.getMax()}]")*

Generics

- **Generic functions:**

- ```
  class EngineDiagnostics {
      /**
       *  Method check engine is generic.
       * It only takes classes that are extending Engine class.
       */
      fun <T : Engine> checkEngine(engine: T) {
          println(engine)
      }
  }
  ```
  ```

Generics

```
/**
 * Engine abstraction
 */
abstract class Engine {
    abstract val power: Long

    override fun toString(): String {
        return "${this::class.simpleName} (power=$power)"
    }
}
```

Generics

```
/**
 * Rocket engine
 */
class RocketEngine : Engine() {
    override val power: Long
        get() = 1000
}

/**
 * Truck engine
 */
class TruckEngine : Engine() {
    override val power: Long
        get() = 100
}
```

Generics

- *val truckEngine = TruckEngine()*
  val rocketEngine = RocketEngine()
  val diagnostics = EngineDiagnostics()
  diagnostics.checkEngine(truckEngine)
  diagnostics.checkEngine(rocketEngine)

- Output:

- *TruckEngine (power=100)*
- *RocketEngine (power=1000)*

Delegates

- **Delegating behavior:**

- *// We delegate Flying to passed Flying instance.*
- *class Traveling(fly: Flying) :* ***Flying by fly***

- ```
  interface Flying {
      fun fly()
  }

  class Plane : Flying {
      override fun fly() {
          println("PLANE")
      }
  }

  class Zeppelin : Flying {
      override fun fly() {
          println("ZEPPELIN")
      }
  }
  ```
  ```

Delegates

- *val plane = Plane()*
  val zeppelin = Zeppelin()

  val travelByPlane = Traveling(plane)
  val travelByZeppelin = Traveling(zeppelin)

  travelByPlane.fly() // Outputs: PLANE
  travelByZeppelin.fly() // Outputs: ZEPPELIN

Delegates

- **Delegating properties:**

- ```
  /**
   * We delegate salary property to SalaryDelegate class.
   */
  class Worker {
      var salary: Int by SalaryDelegate(BaseSalaryCalculation())
  }
  ```
  ```

Delegates

```
/**
 * To become property delegate first we must implement
 * ReadWriteProperty.
 */
class SalaryDelegate(val calculation: SalaryCalculation) :
        ReadWriteProperty<Any, Int>
{
   private var salary = 0

   override fun getValue(thisRef: Any, property: KProperty<*>): Int {
      return salary
   }

   override fun setValue(thisRef: Any, property: KProperty<*>, value: Int) {
      salary = calculation.calculate(value)
   }
}
```

Delegates

```
/**
 * Salary.
 */
class BaseSalaryCalculation : SalaryCalculation {
    override fun calculate(salaryBase: Int): Int {
        return salaryBase * 100
    }
}
```

```
interface SalaryCalculation {
    fun calculate(salaryBase: Int): Int
}
```

Delegates

```
/**
 * Finally, we run our code.
 */
fun main(args: Array<String>) {
    val worker = Worker()
    worker.salary = 10

// Outputs 1000.
// by executing the code from BaseSalaryCalculation class.
    println("Worker earned [ ${worker.salary} ]")
}
```

Delegates

- **Lazy initialization:**

```
/**
 * Class that uses lazy initialization to initialize its field.
 */
class PostgreClient {

    /**
     * Lazy initialization.
     */
    val database by lazy { // by using 'lazy' delegate
        Database("PostgreSQL")
    }

}
```

Delegates

```
class Database(val type: String) {
    init {
        println("Initializing $type")
    }
}

fun main(args: Array<String>) {
    val client = PostgreClient()

    // Initialization will be executed here
    val database = client.database
    // and each time when we access property, it will not be
    // executed again.
}
```

Delegates

- **Observable delegate:**

```
/**
 * Stock value with observer delegate.
 */
class Stock {
    var value : Int by Delegates.observable(0) {
        property, old, new ->
                println(
                        "${property.name}: [ $old ] -> [ $new ]"
                )
    }
}
```

Delegates

```
/**
 * We will set stock values here.
 */
fun main(args: Array<String>) {
    val stock = Stock()
    stock.value = 10   // Outputs: value: [ 0 ] -> [ 10 ]
    stock.value = 100  // Outputs: value: [ 10 ] -> [ 100 ]
    stock.value = 1000 // Outputs: value: [ 100 ] -> [ 1000 ]
}
```

References:

1. http://kotlinlang.org/
2. Fundamental Kotlin, Miloš Vasić, 2016

Code used in examples is located here:

- https://github.com/milos85vasic/Kotlin-Serbia

- **Git repository:**
- https://github.com/milos85vasic/Kotlin-Serbia.git

Miloš Vasić

- Software engineer at Robert Bosch d.o.o.
- Author of Fundamental Kotlin and open source enthusiast.

- Website: milosvasic.net
- Book: fundamental-kotlin.com
- GitHub: github.com/milos85vasic
- E-Mail: milos85vasic@gmail.com

Kotlin user group Serbia

- https://twitter.com/kotlin_serbia
- https://www.facebook.com/kotlinserbia/
- https://www.meetup.com/Serbia-Kotlin-User-Group