9.2 Destructuring and Declarations:

Kotlin allows to declare multiple variables at once. This technique is called ***Destructuring*** declaration.

For example:

```
    val (name, age) = person
```

Example:

```
data class Student(val a :String,val b: String){

     var name:String = a

     var subject:String = b

}

fun main(args: Array<String>) {

     val s = Student("TutorialsPoint.com","Kotlin")

     val (name,subject) = s

     println("You are learning "+subject+" from "+name)

}
```

output:

```
    You are learning Kotlin from TutorialsPoint.com
```

https://kotlinlang.org/docs/reference/multi-declarations.html

Example:

```
val (name, age) = person

          println(name)

          println(age)
```

A destructuring declaration is compiled down to the following code:

```
          val name = person.component1()
          val age = person.component2()
```

The *component1()* and *component2()* functions are another example of the principle of conventions widely used in Kotlin (see operators like + and *, for-loops etc.).

Anything can be on the right-hand side of a destructuring declaration, as long as the required number of component functions can be called on it.

And, of course, there can be component3() and component4() and so on.

Destructuring declarations also work in for-loops: when you say:

```
for ((a, b) in collection) { ... }
```

Variables **a** and **b** get the values returned by **component1()** and **component2()** called on elements of the collection.

Example: Returning Two Values from a Function

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // computations
    return Result(result, status)
}

     // Now, to use this function:
     val (result, status) = function(...)
```

Example: Destructuring Declarations and Maps

Probably the nicest way to traverse a map is this:

```
for ((key, value) in map) {
   // do something with the key and the value
}
```

To make this work, we should

- present the map as a sequence of values by providing an iterator() function;
- present each of the elements as a pair by providing functions component1() and component2().

And indeed, the standard library provides such extensions:

```
operator fun <K, V> Map<K, V>.iterator():
                              Iterator<Map.Entry<K, V>> =
                                   entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

So you can freely use destructuring declarations in for-loops with maps (as well as collections of data class instances etc).

Underscore for unused variables (since 1.1)

If you don't need a variable in the destructuring declaration, you can place an underscore instead of its name:

```
     val (_, status) = getResult()
```

The *componentN()* operator functions are not called for the components that are skipped in this way.

Example: Returning Two Values from a Function

Let's say we need to return two things from a function. For example, a result object and a status of some sort. A compact way of doing this in Kotlin is to declare a data class and return its instance:

```
     data class Result(val result: Int, val status: Status)
     fun function(...): Result {
         // computations

         return Result(result, status)
     }

     Now, to use this function:
     val (result, status) = function(...)
```

Since data classes automatically declare componentN() functions, destructuring declarations work here.

Destructuring in Lambdas (since 1.1)

You can use the destructuring declarations syntax for lambda parameters.
If a lambda has a parameter of the Pair type (or Map.Entry, or any other type that has the appropriate componentN functions), you can introduce several new parameters instead of one by putting them in parentheses:

```
     map.mapValues { entry -> "${entry.value}!" }
     map.mapValues { (key, value) -> "$value!" }
```

Note the difference between declaring two parameters and declaring a destructuring pair instead of a parameter:

```
{ a -> ... } // one parameter
{ a, b -> ... } // two parameters
{ (a, b) -> ... } // a destructured pair
{ (a, b), c -> ... } // a destructured pair and another parameter
```

If a component of the destructured parameter is unused, you can replace it with the underscore to avoid inventing its name:

```
     map.mapValues { (_, value) -> "$value!" }
```

You can specify the type for the whole destructured parameter or for a specific component separately:

```
map.mapValues { (_, value): Map.Entry<Int, String>
                                             -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```