Lab 0: Kotlin basics

In case you found something to improve, please tell us!
https://forms.gle/3U6WrZNyNx2nBXQ38

1 Introduction

Today's lab will introduce the main concepts of the Kotlin programming language. If you are familiar with C, some of the features of Kotlin will be familiar. If you are familiar primarily with the Java programming language, you may be amazed at how much more concise and readable your code can be. Kotlin is focused on clarity, conciseness, and code safety. Kotlin has been around since 2011, and was released as open source in 2012. It reached version 1.0 in 2016, and since 2017 Kotlin has been an officially supported language for building Android apps.

Kotlin allows for both functional and object-oriented programming styles. Herein, we will cover aspects from both with code samples. We will only scratch the surface of what Kotlin has to offer, as a stepping stone towards implementing modern Android apps. Much more can be found in the official Kotlin documentation[1].

2 Today's lab

If not already done yet, install Android Studio. Computers in class have the version **2024.3.1 Patch 2** (April 2025) pre-installed. It is available for download at the Android Studio website[2]. If you want to install Android Studio on your own computer, please select this same version in order to avoid compatibility issues. In this lab, you learn the basics of the Kotlin programming language, including data types, operators, variables, control structures, classes, and nullable versus non-nullable variables.

[1] https://kotlinlang.org/docs/home.html
[2] https://developer.android.com/studio/archive

3 Kotlin as a functional language

3.1 First Kotlin Code

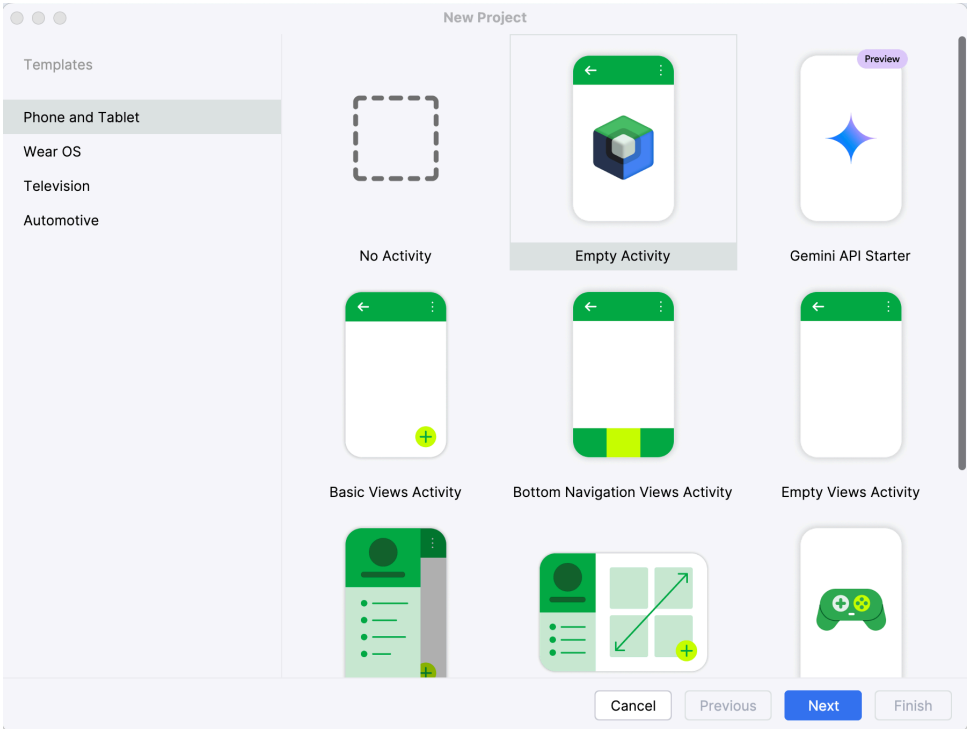

Figure 1: Creating an Empty project in Android Studio

For this lab, open **Android Studio** and click on **New Project**, and choose **No Activity** in the **Phone and Tablet** tab, as shown in Figure 1. Then, click on the **Next** button to go to Configure Your Project. Fill the configuration with a name **Lab0**, package name **com.epfl.esl.lab0**, and an appropriate save location. Choose the **Language** as **Kotlin**. Then, click on **Finish** to start the new project.

In the directory list in the left hand side window, go to **app** -> **src** -> **Kotlin+java** -> **com.epfl.esl.lab0**. Right-click on the package name directory and then, **New** -> **Kotlin Class/File**. Select **File** in the pop-up window, and give as a file name **TestFile**.

You will use this file to execute some **Kotlin** code, which we present in the following. For now, simply add the following lines to the file:

```
fun main() {
    println("Hello World!")
}
```

To run the main() function, right-click on the file name on the left and choose *Run 'TestFileKt'*. The code should show **HelloWorld**! in the output window (to see the output, select the **Run** tab, at the bottom).

Now, follow the following sections step by step, and try to use the code to understand better the Kotlin language and concepts.

3.2 Variables

As with other languages, Kotlin uses +, -, * and / for plus, minus, times and division. Kotlin also supports different number types, such as Int, Long, Double, and Float. For strings, + is used for concatenation. Here are few examples:

```
fun main() {
    val i: Int = 6
    val b0: Double = i.toFloat()*50.0
    val b2: String = b0.toString()
    println("Hello " + b2 + " Worlds!")
}
```

Kotlin supports two types of variables: changeable and unchangeable. With **val**, you can assign a value once. If you try to assign something again, you get an error. With **var**, you can assign a value, then change the value later in the program. Try copy-pasting these statements inside the **main()** function:

```
var fish = 1
fish = 2
val aquarium = 1
aquarium = 2
```

If you try to build the resulting code, you will see in the **Build** tab in the bottom window an error such as "val cannot be reassigned".

Strings in Kotlin work pretty much like strings in any other programming language, using " for strings and ' for single characters. You can create string templates by combining them with values; the **$variable** name is replaced with the text representing the value. This is called variable interpolation.

```
val numberOfFish = 5
val numberOfPlants = 12
println("I have $numberOfFish fish" + " and $numberOfPlants plants")
```

Like other languages, Kotlin has booleans and boolean operators such as less than, equal to, greater than, and so on (<, ==, >, !=, <=, >=).

```
val numberOfFish = 50
val numberOfPlants = 23
if (numberOfFish > numberOfPlants) {
    println("Good ratio!")
} else {
    println("Unhealthy ratio")
}

val fish = 50
if (fish in 1..100) {
    println(fish)
}
```

For more complicated conditions, there's a nicer way to write that series of if/else if/else statements in Kotlin, using the **when** statement, which is like the **switch** statement in other languages. Conditions in a **when** statement can use ranges, too. Note that **when** automatically breaks at the end of each branch.

```
when (numberOfFish) {
    0  -> println("Empty tank")
    in 1..39 -> println("Got fish!")
    else -> println("That's a lot of fish!")
}
```

3.3 Lists and loops

Lists are a fundamental type in Kotlin, and are similar to lists in other languages. You can use **listOf** to declare a list. This list cannot be changed. A list that can be changed should be declared using **mutableListOf**.

```
val school = listOf("mackerel", "trout", "halibut")
val myList = mutableListOf("tuna", "salmon", "shark")
school.remove("trout") // Error
myList.remove("shark")
```

Like other languages, Kotlin has arrays. Unlike lists in Kotlin, which have mutable and immutable versions, there is no mutable version of an Array. Once you create an array,

the size is fixed. You can't add or remove elements, except by copying to a new array. The rules about using val and var are the same with arrays as with lists.

```
val school_array = arrayOf("shark", "salmon", "minnow")
val mix = arrayOf("fish", 2)
val numbers = intArrayOf(1,2,3)
```

One nice feature of Kotlin is that you can initialize arrays with code instead of initializing them to 0. Look at this example:

```
val array = Array (5) { it * 2 }
```

The initialization code is between the curly braces, **{}**. In the code, **it** refers to the array index, starting with 0.

Now that you have lists and arrays, looping through the elements works as you might expect.

```
for (element in school) {
    print(element + " \n")
}

for ((index, element) in school.withIndex()) {
    println("Item at $index is $element\n")
}
```

You can specify ranges of numbers or of characters, alphabetically. And as in other languages, you don't have to step forward by 1. You can step backward using **downTo**.

```
for (i in 1..5) print(i)
-> 12345

for (i in 5 downTo 1) print(i)
-> 54321

for (i in 3..6 step 2) print(i)
-> 35

for (i in 'd'..'g') print (i)
-> defg
```

Like other languages, Kotlin has **while** loops, **do...while** loops, and ++ and -- operators. Kotlin also has **repeat** loops.

```
var bubbles = 0
while (bubbles < 50) {
   bubbles++
}
println("$bubbles bubbles in the water\n")

do {
   bubbles--
} while (bubbles > 50)
println("$bubbles bubbles in the water\n")

repeat(2) {
    println("A fish is swimming")
}
```

Kotlin has several ways of controlling flow. As the name implies, **return** returns from a function to its enclosing function. Using a **break** is like **return**, but for loops. Kotlin gives you additional control over loops with what's called a labeled break. A **break** qualified with a label jumps to the execution point right after the loop marked with that label. This is particularly useful when dealing with nested loops. Any expression in Kotlin may be marked with a label. Labels have the form of an identifier followed by the @ sign.

```
outerLoop@ for (i in 1..100) {
        print("$i ")
        for (j in 1..100) {
            if (i > 10) break@outerLoop  // breaks to after outer loop
       }
   }
```

3.4 Nullability

Programming errors involving nulls have been the source of countless bugs. Kotlin seeks to reduce bugs by introducing non-nullable variables. In Kotlin, by default, variables cannot be null. Use the question mark operator, ?, after the type to indicate that a variable can be null.

```
var fishFoodTreats: Int? = null
```

You can test for null with the ? operator, saving you the pain of writing many if/else statements.

```
if (fishFoodTreats != null) {
    fishFoodTreats = fishFoodTreats.dec()
}
```

Now look at the Kotlin way of writing it, using the **?** operator.

```
fishFoodTreats = fishFoodTreats?.dec()
```

You can also perform nullability tests with the **?:** operator. Look at this example:

```
fishFoodTreats = fishFoodTreats?.dec() ?: 0
```

It's shorthand for "if fishFoodTreats is not null, decrement and use it; otherwise use the value after the **?:**, which is 0." If fishFoodTreats is null, evaluation is stopped, and the dec() method is not called. The **?:** operator is sometimes called the "Elvis operator," because it's like a smiley on its side with a pompadour hairstyle, the way Elvis Presley styled his hair.

If you really love NullPointerExceptions, Kotlin lets you keep having them. The not-null assertion operator, **!!** (double-bang), converts any value to a non-null type and throws an exception at run-time if the value is null.

```
val len = s!!.length   // throws NullPointerException if s is null
```

3.5 Functions

Look at this **printHello()** function. Functions are defined using the **fun** keyword, followed by the name of the function. As with other programming languages, the parentheses () are for function arguments, if any. Curly braces **{}** frame the code for the function. There is no return type for this function, because it doesn't return anything.

```
fun printHello() {
    println ("Hello World")
}
```

Copy-paste this function in TextFile.kt, and invoke it from the **main()** to print "Hello World" in the output (**Run**) window.

Functions can accept inputs and return outputs. The input parameters list is specified inside the parenthesis after the function name, while return type should be indicated after the parameters list and the semicolumn (**:**):

```
fun fishFood (day : String) : String {
    val food : String
    when (day) {
        "Monday" -> food = "flakes"
        "Wednesday" -> food = "redworms"
        "Thursday" -> food = "granules"
        "Friday" -> food = "mosquitoes"
        "Sunday" -> food = "plankton"
        else -> food = "nothing"
    }
    return food
}
```

Call **fishFood()** from **main()** and println the returned string.

In Kotlin, you can pass arguments by parameter name. You can also specify default values for parameters: if an argument isn't supplied by the caller, the default value is used. Here is an example:

```
fun shouldChangeWater (day: String, temperature: Int = 22,
                       dirty: Int = 20): Boolean {
    return when {
        temperature > 30 -> true
        dirty > 30 -> true
        day == "Sunday" ->  true
        else -> false
    }
}
```

Compact functions, or single-expression functions, are a common pattern in Kotlin. When a function returns the results of a single expression, you can specify the body of the function after an = symbol, omit the curly braces **{}**, and omit the **return**.

```
fun isTooHot(temperature: Int) = temperature > 30

fun isDirty(dirty: Int) = dirty > 30

fun isSunday(day: String) = day == "Sunday"
```

In addition to traditional named functions, Kotlin supports **lambdas**. A lambda is an expression that makes a function. But instead of declaring a named function, you declare a function that has no name. Part of what makes this useful is that the lambda expression can now be passed as data. In other languages, lambdas are called **anonymous functions**, **function literals**, or similar names.

Like named functions, lambdas can have parameters. For lambdas, the parameters (and their types, if needed) go on the left of what is called a function arrow **->**. The code to execute goes to the right of the function arrow. Once the lambda is assigned to a variable, you can call it just like a function.

```
val waterFilter = { dirty : Int -> dirty / 2}
```

In this example, the lambda takes an Int named **dirty**, and returns **dirty / 2**.

Kotlin's syntax for function types is closely related to its syntax for lambdas. Use this syntax to cleanly declare a variable that holds a function:

```
val waterFilter: (Int) -> Int = { dirty -> dirty / 2 }
```

Here's what the code says:

- Make a variable called waterFilter.
- waterFilter can be any function that takes an Int and returns an Int.
- Assign a lambda to waterFilter.
- The lambda returns the value of the argument dirty divided by 2.

Note that you don't have to specify the type of the lambda argument. If this is the case, the type is calculated by type inference.

3.6 Higher order functions

Now, let's see a useful use-case for lambdas. They can be used in higher-order functions. A higher-order function is a function that takes functions as parameters and/or returns a function. We are going to use the waterFilter lambda from the previous example:

```
fun updateDirty(dirtyLevel: Int, operation: (Int) -> Int): Int {
    return operation(dirtyLevel)
}
```

Higher-order functions can be called just like any other functions, by providing a lamda or function parameter. Here is an example using the waterFilter lambda

```
fun main(){
    var dirtyLevel = 4
    dirtyLevel = updateDirty(dirtyLevel, waterFilter)
    println("Water pollution level is " + dirtyLevel.toString())
}
```

Here **operation** is a function that is passed as a parameter to the function **updateDirty**, which uses **operation** and returns its result.

What if waterFilter is defined as a function, and not a lambda? In this case, we will need to use the **::** operator, which provides a method reference when we are calling the function.

```
fun waterFilterFun(dirty: Int): Int {
    return dirty / 2
}

fun main(){
    dirtyLevel = updateDirty(dirtyLevel, ::waterFilterFun)
}
```

4 Kotlin as an object oriented language

4.1 Classes

The following programming terms should already be familiar to you:

- **Classes** are blueprints for objects. For example, an Aquarium class is the blueprint for making an aquarium object.
- **Objects** are instances of classes; an aquarium object is one actual Aquarium.
- **Properties** are characteristics of classes, such as the length, width, and height of an Aquarium.
- **Methods**, also called member functions, are the functionality of the class. Methods are what you can ”do” with the object. For example, you can **fillWithWater()** an Aquarium object.

Let’s take an example of an object, for instance a bicycle. Bicycles have a state, represented by the current gear, pedal cadence and speed, and a behavior, such as changing gear, changing pedal cadence, speed up, applying brakes, and so on.

Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming. Software objects are conceptually similar to

real-world objects: they consist of state and related behavior. An object stores its state in fields (properties) and describes its behavior through member functions (methods).

With respect to our previous example, each bicycle is built from the same set of blueprints and therefore contains the same main components that describe them as bicycles, but each bicycle is physically different from each other. In object-oriented terms, we say that **aBicycle** and **anotherBicycle** are two instances of the class of objects known as **Bicycle**. Therefore, a class is the blueprint from which individual objects are created (instantiated).

An implementation of the **Bicycle** class may look as follows. Notice that the **Bicycle** class does not contain a **main(...)** method. This is due to the fact that this class is not a complete application, it just represents the **category** of bicycles from which different objects (specific bicycles) can be instantiated.

```
class Bicycle {
    var cadence = 0
    var speed = 0
    var gear = 1
    fun changeCadence(newValue: Int) {
        cadence = newValue
    }

    fun changeGear(newValue: Int) {
        gear = newValue
    }

    fun speedUp(increment: Int) {
        speed = speed + increment
    }

    fun applyBrakes(decrement: Int) {
        speed = speed - decrement
    }

    fun printStates() {
        println("cadence:$cadence")
        println("speed:$speed")
        println("gear:$gear")
    }
}
```

(Public) class methods can be called from other objects or from functions:

```
fun main() {
    var aBycicle = Bicycle()
    aBycicle.speedUp(3)
    aBycicle.printStates()
}
```

4.2 Creating and modifying objects

Create a new file by right-click on the package name directory and then, **New -> Kotlin Class/File**. Set the file name as **Aquarium** because we want to make a class with the same name. The following example defines a class and initialize its parameters with default values. class names by convention start with a capital letter

```
class Aquarium {
    var width: Int = 20
    var height: Int = 40
    var length: Int = 100
}
```

Under the hood, Kotlin automatically creates getters and setters for the properties you defined in the **Aquarium** class, so you can access the properties directly, for example, **myAquarium.length**.

Note: If you declared these properties with **val** instead of **var**, the properties would be immutable. You could only set them once, and all the instances of Aquarium would have the same dimensions.

Also note that Android Studio underlines the name of each **var** in your code, but not each **val**. Kotlin coding style prefers immutable data when possible, so Android Studio draws your attention to mutable data so you can minimize its use.

To create an instance, reference the class (e.g. in the **TestClass.kt** file, inside the **main()** function) as if it were a function, **Aquarium()**. This calls the constructor of the class and creates an instance of the **Aquarium** class, similar to using **new** in other languages.

```
val myAquarium = Aquarium()
```

In the earlier example, every instance of Aquarium is created with the same dimensions. You can change the dimensions once it is created by setting the properties, but it would be simpler to create it the correct size to begin with.

In some programming languages, the constructor is defined by creating a method within the class that has the same name as the class. In Kotlin, you define the constructor directly in the class declaration itself, specifying the parameters inside parentheses as if the class was a method. As with functions in Kotlin, those parameters can include default values.

```
class Aquarium(length: Int = 100, width: Int = 20, height: Int = 40) {
   // Dimensions in cm
   var length: Int = length
   var width: Int = width
   var height: Int = height
...
}
```

The more compact Kotlin way is to define the properties directly with the constructor, using **var** or **val**, and Kotlin also creates the getters and setters automatically. Then you can remove the property definitions in the body of the class.

```
class Aquarium(var length: Int = 100, var width: Int = 20,
               var height: Int = 40) {
...
}
```

When you create an Aquarium object with that constructor, you can specify no arguments and get the default values, or specify just some of them, or specify all of them and create a completely custom-sized Aquarium

```
val aquarium1 = Aquarium()

// default height and length
val aquarium2 = Aquarium(width = 25)

// default width
val aquarium3 = Aquarium(height = 35, length = 110)

// everything custom
val aquarium4 = Aquarium(width = 25, height = 35, length = 110)
```

The example constructors above just declare properties and assign the value of an expression to them. If your constructor needs more initialization code, it can be placed in one or more **init** blocks.

```
class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int = 40) {
    init {
        println("aquarium initializing")
    }
    init {
        // 1 liter = 1000 cm^3
        println("Volume: ${width * length * height / 1000} l")
    }
}
```

Notice that the init blocks are executed in the order in which they appear in the class definition, and all of them are executed when the constructor is called. Parameters of the primary constructor can be used in the initializer blocks. Any properties used in initializer blocks must be declared previously.

Kotlin automatically defines getters and setters when you define properties, but sometimes the value for a property needs to be adjusted or calculated. For example, above, you printed the volume of the Aquarium. You can make the volume available as a property by defining a variable and a getter for it. Because volume needs to be calculated, the getter needs to return the calculated value, which you can do with a one-line function.

```
val volume: Int
    get() = width * height * length / 1000  // 1000 cm^3 = 1 l
```

You can also create a new property setter for the volume, so that, if the volume changes, height is automatically adjusted.

```
var volume: Int
    get() = width * height * length / 1000
    set(value) {
        height = (value * 1000) / (width * length)
    }
```

There have been no visibility modifiers, such as **public** or **private**, in the code so far. That's because by default, everything in Kotlin is public, which means that everything can be accessed everywhere, including classes, methods, properties, and member variables.

In Kotlin, classes, objects, interfaces, constructors, functions, properties, and their setters can have visibility modifiers:

- **public** means visible outside the class. Everything is public by default, including variables and methods of the class.

- **`internal`** means it will only be visible within that module. A module is a set of Kotlin files compiled together, for example, a library or application.
- **`private`** means it will only be visible in that class (or source file if you are working with functions).
- **`protected`** is the same as private, but it will also be visible to any subclasses.

Properties within a class, or member variables, are public by default. If you define them with var, they are mutable, that is, readable and writable. If you define them with val, they are read-only after initialization.

4.3 Class inheritance

In the Kotlin language, classes can be derived from other classes. A class that is derived from another class is called a subclass (child class). The class from which the subclass is derived from is called a superclass (parent class).

`Any` is the only class in Kotlin that has no superclass. Every class is implicitly a subclass of **`Any`**. There is no limit in inheritance, and ultimately everything is derived from the topmost class, **`Any`**.

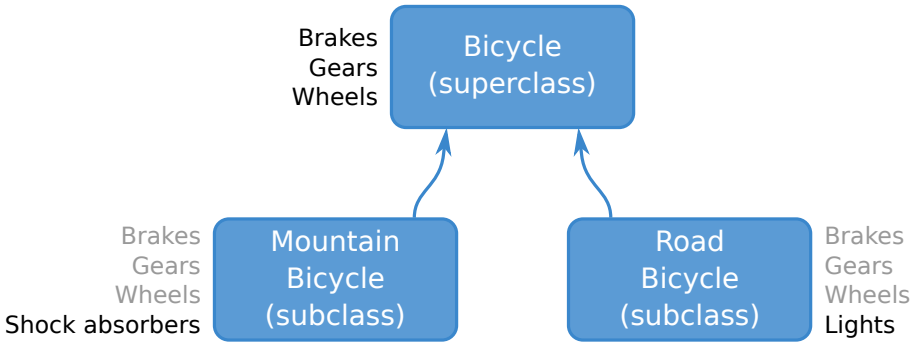

Figure 2: Two subclasses of the superclass **`Bicycle`** inheriting same features and adding more specific ones

The idea of inheritance is simple: creating a new class with the same features as one already existing and, also adding more specific features. Coming back to the **`Bicycle`** example, this class describes a generic bicycle with features such as gears, breaks, etc. If we want to go on a road trip, we might need a road bike that has lights for when it becomes dark in order to be seen. However, in our road trip we might need a mountain bike, since we want to go off track. As shown in Figure 2, we can create a class named **`MountainBicycle`**, that would have the same features as a generic bicycle, with some additional ones, such as shock absorbers. In both cases, the bicycles have wheels, gears and brakes, but they can be tuned for a specific use case. **`MountainBicycle`** and **`RoadBicycle`** are

subclasses of **Bicycle**. By doing this, we can reuse the fields and methods of the existing superclass without having to write them again. A subclass inherits all the members (fields, methods, and nested classes) from its superclass.

In Kotlin, by default, classes cannot be subclassed. Similarly, properties and member variables cannot be overridden by subclasses (though they can be accessed).

You must mark a class as **open** to allow it to be subclassed. Similarly, you must mark properties and member variables as **open**, in order to override them in the subclass.

```
open class Aquarium (open var length: Int = 100, open var width: Int = 20,
                     open var height: Int = 40) {
    open var volume: Int
        get() = width * height * length / 1000
        set(value) {
            height = (value * 1000) / (width * length)
        }
    }
```

Now, you can create a subclass of **Aquarium** called **TowerTank**, which implements a rounded cylinder tank instead of a rectangular tank. You can add **TowerTank** below **Aquarium**, because you can add another class in the same file as the **Aquarium** class. Override the volume property to calculate a cylinder. The formula for a cylinder is pi times the radius squared times the height. You need to import the constant PI from java.lang.Math.

```
class TowerTank (override var height: Int, var diameter: Int):
                 Aquarium(height = height, width = diameter,
                          length = diameter) {
    override var volume: Int
    // ellipse area = π * r1 * r2
        get() = (width/2 * length/2 * height / 1000 * PI).toInt()
        set(value) {
            height = ((value * 1000 / PI) / (width/2 * length/2)).toInt()
        }
}
```

You can instantiate **TowerTank**s similarly to **Aquarium**s:

```
val myTower = TowerTank(diameter = 25, height = 40)
```

4.4 Data classes

A data class is similar to a **struct** in some other languages—it exists mainly to hold some data—but a data class object is still an object. Kotlin data class objects have some extra benefits, such as utilities for printing and copying.

```
data class Decoration(val rocks: String, val wood: String, val diver: String){
}
```

You can instantiate objects of this data class as follows:

```
val d0 = Decoration("crystal", "wood", "diver")
```

To have access to the values of a data class you can use either variables or destructing.

```
val rock = d0.rocks
val wood = d0.wood
val diver = d0.diver

// Assign all properties to variables.
val (rock1, wood1, diver1) = d0
```

If you don't need one or more of the properties, you can skip them by using _ instead of a variable name, as shown in the code below.

```
    val (rock2, _, diver2) = d0
```

We here end our overview of the main features of the Kotlin language. In the next lab, we will use Kotlin to develop our first Android app.