CP3406_CP5307 Codelab 1.1: Kotlin Basics

Contents

1. Benefits of Kotlin
2. How to use Kotlin data types, operators, and variables
3. How to work with Booleans and conditions
4. The difference between nullable and non-nullable variables
5. How arrays, lists and loops work in Kotlin
6. Summary

1. Benefits of Kotlin

Kotlin is a modern programming language that helps developers be more productive. It focuses on clarity, conciseness, and code safety. Let’s briefly recap some of the key features of Kotlin.

Robust code

The creators of Kotlin made various design decisions about the language to help programmers create robust code. For example, null-pointer exceptions in software have caused financial losses and spectacular computer crashes, and have resulted in countless hours of debugging. So Kotlin distinguishes between nullable and non-nullable data types, which helps catch more errors at compile time. Kotlin is strongly typed, and it does a lot to infer the types from your code. It has lambdas, coroutines, and properties, which allow you to write less code with fewer bugs.

Mature platform

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. It's included with the IntelliJ IDEA as well as Android Studio 3.0 and later.

Concise, readable code

Code written in Kotlin can be very concise, and the language is designed to eliminate boilerplate code such as getters and setters. For example, consider the following Java code:

```
public class Aquarium {
   private int mTemperature;
   public Aquarium() { }
   public int getTemperature() {
       return mTemperature;
   }
   public void setTemperature(int mTemperature) {
       this.mTemperature = mTemperature;
   }
   @Override
   public String toString() {
       return "Aquarium{" +
               "mTemperature=" + mTemperature +
               '}';
   }
}
```

It can be written concisely like this in Kotlin:

```
data class Aquarium(var temperature: Int = 0)
```

Sometimes the goals of conciseness and readability are at odds with each other. Kotlin is designed to use "just enough boilerplate code" to ensure readability while keeping things concise.

Interoperable with Java

Kotlin code compiles so that you can use Java and Kotlin code side-by-side, and continue to use your favorite Java libraries. You can add Kotlin code to an existing Java program, or if you want to migrate your program completely, IntelliJ IDEA and Android Studio both include tools to migrate existing Java code to Kotlin code.

2. Hello Kotlin

Following the time-honoured tradition, let's start off with a simple "Hello World" program in Kotlin.

Open up your web browser of choice and go to https://play.kotlinlang.org/. If you've arrived at the right place, you will see a code editor with the following function:

```
fun main() {
    println("Hello, world!!!")
}
```

If you click the Run button, the code inside the function *main* will run, and you'll see "Hello, world!!!" printed to the console. Let's write one more function. On a new line beneath the existing code, type in the following:

```
fun greeting() {
    println("Hello, Kotlin!!!")
}
```

Take a quick look at this Kotlin code. The `fun` keyword designates a function, followed by the name, in this case `greeting`. As with other programming languages, the parentheses are for function parameters, if there are any, and the curly braces frame the code for the function (formally,they provide a scope). There is no return type because this function doesn't return anything. Also note Kotlin does not require semicolons at the ends of lines (though if you did put semicolons at the end of the lines, Kotlin won't complain).

If you run the program, nothing new happens. Why do you think that is? You've defined a new function, but it needs to be called in the main method. Modify the code in the function `main` as follows:

```
fun main() {
    println("Hello, Kotlin!!!")
    greeting()
}
```

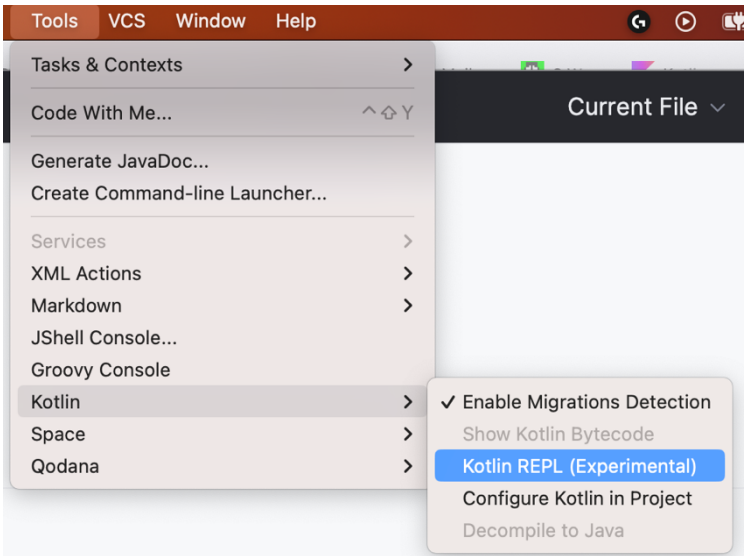
Now when you run the program, you should see both greetings. Congratulations! You've written your first Kotlin program.

3. Learn about operators and types

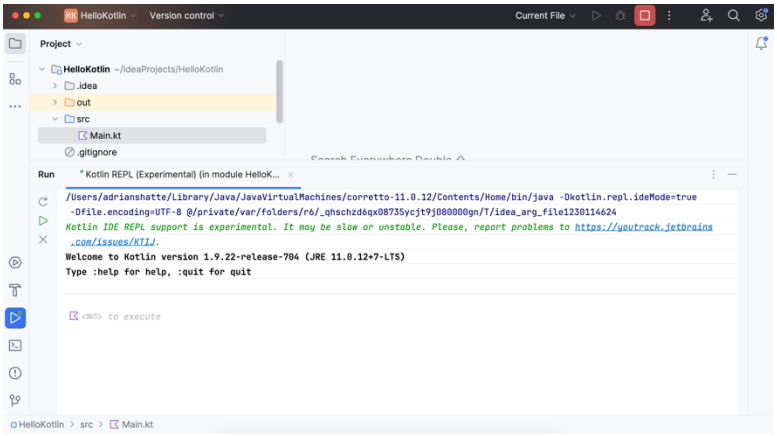
Let's now explore the basics of the Kotlin programming language: data types, operators, variables, control structures, and nullable versus non-nullable variables.

Step 1: Explore numeric operators

1. Open IntelliJ IDEA. In the Welcome to IntelliJ IDEA window, click + New Project.
2. In the New Project window, enter the name "Hello Kotlin". Select language as Kotlin, build system IntelliJ, and select the latest version of JDK that is have installed. Click Create.
3. Go to the *Tools* menu, and select *Kotlin > Kotlin REPL (Experimental)*.



4. You should now see the Kotlin REPL open in a panel within IntelliJ IDEA.



   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`.

5. Enter the following expressions in the REPL. Execute the code and see the result by pressing `Control + Enter` (`Command + Enter` on a Mac) after each one. Note that you can enter multiple lines before executing the code.

   Note: In this codelab, a line beginning with ⇒ represents the output.

   - `1+1`
     `⇒ res8: kotlin.Int = 2`
   - `53-3`
     `⇒ res9: kotlin.Int = 50`
   - `50/10`
     `⇒ res10: kotlin.Int = 5`
   - `1.0/2.0`
     `⇒ res8: kotlin.Int = 0.5`
   - `2.0*3.5`
     `⇒ res8: kotlin.Int = 7.0`

   Note that results of operations keep the types of the operands, so `1/2` = 0, but `1.0/2.0` = 0.5. Is that what you expected?

6. Try some expressions with different combinations of integer and decimal numbers.

   - `6*50`
     `⇒ res13: kotlin.Int = 300`
   - `6.0*50.0`
     `⇒ res14: kotlin.Double = 300.0`
   - `6.0*50`
     `⇒ res15: kotlin.Double = 300.0`

7. Call some methods on numbers. Kotlin keeps numbers as primitives, but it lets you call methods on numbers as if they were objects.

- `2.times(3)`
  `⇒ res5: kotlin.Int = 6`

- `3.5.plus(4)`
  `⇒ res8: kotlin.Double = 7.5`

- `2.4.div(2)`
  `⇒ res9: kotlin.Double = 1.2`

Note: It is possible to create actual object wrappers around numbers, which is known as *boxing*. Boxinghappens automatically, such as for collections, where numbers are boxed and unboxed as needed.

Warning: Using object wrappers requires more memory than storing just a number primitive. Do not useboxing unless it is needed, such as in a collection, which is covered later.

Step 2: Practice using types

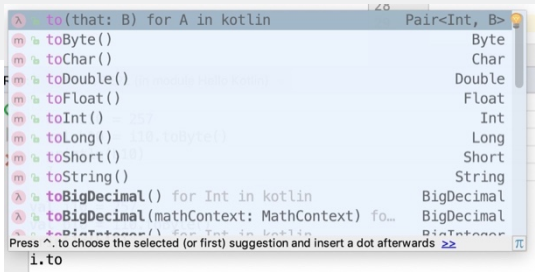
Kotlin does not allow implicit type conversions. So, you can't assign a short value directly to a long variable, or an `Int` to a `Long`. This is because implicit number conversion is a common source of errors in programs. You can always assign values of different types by casting.

1. To see some of the casts that are possible, define a variable of type `Int` in the REPL.

```
val i: Int = 6
```

2. Create a new variable, then enter the variable name shown above, followed by `.to`.

```
val b1 = i.to
```



IntelliJ IDEA displays a list of possible completions. This auto-completion works for variables and objects of any type.

3. Select `toByte()` from the list, then print the variable.

```
val b1 = i.toByte()
println(b1)
```

`⇒ 6`

4. Assign a `Byte` value to variables of different types.

```
val b2: Byte = 1 // OK, literals are checked statically
println(b2)
⇒ 1
val i1: Int = b2
⇒ error: type mismatch: inferred type is Byte but Intwas expected

val i2: String = b2
⇒ error: type mismatch: inferred type is Byte but String was
expected

val i3: Double = b2
⇒ error: type mismatch: inferred type is Byte but Double was expected
```

5. For the assignments that returned errors, try casting them instead.

```
val i4: Int = b2.toInt() // OK!
println(i4)
⇒ 1
val i5: String = b2.toString()
println(i5)
⇒ 1
val i6: Double = b2.toDouble()
println(i6)
⇒ 1.0
```

6. To make long numeric constants more readable, Kotlin allows you to place underscores in the numbers, where it makes sense to you. Try entering different numeric constants.

```
val oneMillion = 1_000_000
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

Note: The Kotlin compiler tries to infer the type for variables, so you don't need to explicitly declare the type if an initial value is provided.

Step 3: Learn the value of variable types

Kotlin supports two types of variables: changeable (mutable) and unchangeable (immutable). They are specified with `var` and `val`, respectively. 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.

1. Define variables using `val` and `var` and then assign new values to them.

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

⇒ `error: val cannot be reassigned`

You can assign `fish` a value, then assign it a new value, because it is defined with `var`. Trying to assign a new value to `aquarium` gives an error because it is defined with `val`.

The type you store in a variable is inferred when the compiler can figure it out from context. If you want, you can always specify the type of a variable explicitly, using the colon notation.

2. Define some variables and specify the type explicitly.

```
var fish: Int = 12
var lakes: Double = 2.5
```

Once a type has been assigned by you or the compiler, you can't change the type, or you get an error.

Step 4: Learn about strings and characters

Strings in Kotlin are sequences of characters. You define them similar to strings in other programming languages by using `"` for strings and `'` for single characters. You can concatenate strings with the `+` operator. Kotlin also allows you to use string template expressions in a string literal. These templates are pieces of code that are evaluated and concatenated into the string literal.

1. Create a string template.

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

⇒ `res20: kotlin.String = I have 5 fish and 12 plants`

2. Create a string template with an expression in it. As in other languages, the value can be the result of an expression. Use curly braces `{}` to define the expression.

```
"I have ${numberOfFish + numberOfPlants} fish and plants"
```

⇒ `res21: kotlin.String = I have 17 fish and plants`

4. Compare conditions and booleans

In this task, you learn about booleans and checking conditions in the Kotlin programming language. Like other languages, Kotlin has booleans and boolean operators such as less than, equal to, greater than, and so on (<, ==, >, !=, <=, >=).

1. Write an `if`/`else` statement.

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

⇒ `Good ratio!`

2. Kotlin offers the ability to easily define a succession of values with starting and terminating endpoints. This is called a range. The easiest way to create a range in Kotlin is with the ".." operator. Try using a range in an `if` statement. In Kotlin, the condition you test can use ranges, too.

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

⇒ `50`

3. Write an `if` with multiple cases. For more complicated conditions, use logical and `&&` and logical or `||`. As in other languages, you can have multiple cases by using `else if`.

```
if (numberOfFish == 0) {
    println("Empty tank")
} else if (numberOfFish < 40) {
    println("Got fish!")
} else {
    println("That's a lot of fish!")
}
```

⇒ `That's a lot of fish!`

4. Try out a `when` statement. A `when` statement can be a convenient way to write that series of`if`/`else` statements in Kotlin. The `when` statement is like the `switch` statement in other languages. Conditions in a `when` statement can use ranges, too.

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

⇒ `That's a lot of fish!`

5. Learn about nullability

In this task, you learn about nullable versus non-nullable variables. Programming errors involving nulls have been the source of countless bugs. Kotlin seeks to reduce bugs by introducing non-nullable variables.

Step 1: Learn about nullability

By default, variables cannot be `null`.

1. Declare an `Int` and assign `null` to it.

```
var rocks: Int = null
```

⇒ `error: null cannot be a value of a non-null type Int`

2. Use the question mark operator, `?`, after the type to indicate that a variable can be null. Declare an `Int?` and assign `null` to it.

```
var marbles: Int? = null
```

When you have complex data types, such as a list:

- You can allow the elements of the list to be null.
- You can allow for the list to be null, but if it's not null its elements cannot be null.
- You can allow both the list or the elements to be null.

Lists and some other complex data types are covered in a later task.

Step 2: Learn about the ? and ?: operators

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

1. Write some code the longer way to check whether the `fishFoodTreats` variable is not `null`. Then decrement that variable.

```
var fishFoodTreats = 6
if (fishFoodTreats != null) {
    fishFoodTreats = fishFoodTreats.dec()
}
```

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

```
var fishFoodTreats = 6
fishFoodTreats = fishFoodTreats?.dec()
```

3. You can also chain null 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.

Note: 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.

A point about null pointers

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

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

Note: In programming slang, the exclamation mark is often called a "bang," so the not-null assertion operator is sometimes called the "double-bang" or "bang bang" operator. Because `!!` can throw an exception, it should only be used when it would be exceptional to hold a null value.

6. Explore arrays, lists, and loops

In this task, you learn about arrays and lists, and you learn different ways to create loops in the Kotlin programming language.

Step 1: Make lists

Lists are a fundamental type in Kotlin, and are similar to lists in other languages.

1. Declare a list using `listOf` and print it out. This list cannot be changed.

   ```
   val school = listOf("mackerel", "trout", "halibut")
   println(school)
   ```

   ⇒ `[mackerel, trout, halibut]`

2. Declare a list that can be changed using `mutableListOf`. Remove an item.

   ```
   val myList = mutableListOf("tuna", "salmon", "shark")
   myList.remove("shark")
   ```

   ⇒ `res36: kotlin.Boolean = true`

   The `remove()` method returns `true` when it successfully removes the item passed.

Note: With a list defined with `val`, you can't change which list the variable refers to, but you can still change the contents of the list.

Step 2: Create arrays

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.

Note: With an array defined with `val`, you can't change which array the variable refers to, but you can still change the contents of the array.

1. Declare an array of strings using `arrayOf`. Use the `java.util.Arrays.toString()` array utility to print it out.

```
val school = arrayOf("shark", "salmon", "minnow")
println(java.util.Arrays.toString(school))
```

⇒ `[shark, salmon, minnow]`

2. An array declared with `arrayOf` doesn't have a type associated with the elements, so you can mix types, which is helpful. Declare an array with different types.

```
val mix = arrayOf("fish", 2)
```

3. You can also declare arrays with one type for all the elements. Declare an array of integers using `intArrayOf()`. There are corresponding builders, or instantiation functions, for arrays of other types.

```
val numbers = intArrayOf(1,2,3)
```

Note: Using an array of a primitive type such as `Int` or `Byte` avoids the overhead of boxing.

4. Combine two arrays with the `+` operator.

```
val numbers = intArrayOf(1,2,3) val
numbers3 = intArrayOf(4,5,6)val foo2 =
numbers3 + numbers println(foo2[5])
```

⇒ `3`

5. Try out different combinations of nested arrays and lists. As in other languages, you can nest arrays and lists. That is, when you put an array within an array, you have an array of arrays—not a flattened array of the contents of the two. The elements of an array can also be lists, and the elements of lists can be arrays.

```
val numbers = intArrayOf(1, 2, 3)
val oceans = listOf("Atlantic", "Pacific")
val oddList = listOf(numbers, oceans, "salmon")
println(oddList)
```

⇒ `[[I@89178b4, [Atlantic, Pacific], salmon]`

The first element, `numbers`, is an `Array`. When you don't use the array utility to print it, Kotlin prints the address instead of the contents of the array.

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

```
val array = Array (5) { it * 2 }
println(java.util.Arrays.toString(array))
```

```
⇒ [0, 2, 4, 6, 8]
```

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

Step 3: Create loops

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

1. Create an array. Use a `for` loop to iterate through the array and print the elements.

```
val school = arrayOf("shark", "salmon", "minnow")
for (element in school) {
    print(element + " ")
}
```

```
⇒ shark salmon minnow
```

2. In Kotlin, you can loop through the elements and the indexes at the same time. Try this example:

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

```
⇒ Item at 0 is shark
  Item at 1 is salmon
  Item at 2 is minnow
```

3. Try different step sizes and ranges. 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
```

4. Try out some loops. 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")
}
```

```
⇒ 50 bubbles in the water
49 bubbles in the water
A fish is swimmingAfish is swimming
```

7. Summary

Kotlin is very similar to other languages when it comes to the basics like operators, lists, and loops, but there are some important differences.

The following features may be different in Kotlin than what you're used to in other languages:

- Kotlin types can't be implicitly converted - use casting.
- Variables declared with `val` can only be assigned once.
- Kotlin variables are not nullable by default. Use `?` to make variables nullable.
- With Kotlin, you can loop through the index and elements of an array at the same time in a `for` loop.

The following Kotlin programming constructs are similar to those in other languages:

- Arrays and lists can have a single type or mixed types.
- Arrays and lists can be nested.
- You can create loops with `for`, `while`, `do`/`while`, and `repeat`.
- The `when` statement is Kotlin's version of the `switch` statement, but `when` is more flexible.