Android – Introduction to Kotlin

Lab Session 2

There are mainly 3 methods to run Kotlin code

The first method is to use IntelliJ IDEA. It is JetBrains' flagship IDE and perfectly supports Kotlin.

The second method is to run Kotlin code online.

To help developers quickly experiment with Kotlin, JetBrains built a website that can run Kotlin code.
The web address is https://try.kotlinlang.org,

The third method is to use Android Studio. Unfortunately, as an IDE specifically designed to develop Android apps, you can only create Android project instead of Kotlin project inside Android

we can just create an Android project
and then write a Kotlin main() method,

then we can independently run Kotlin code.

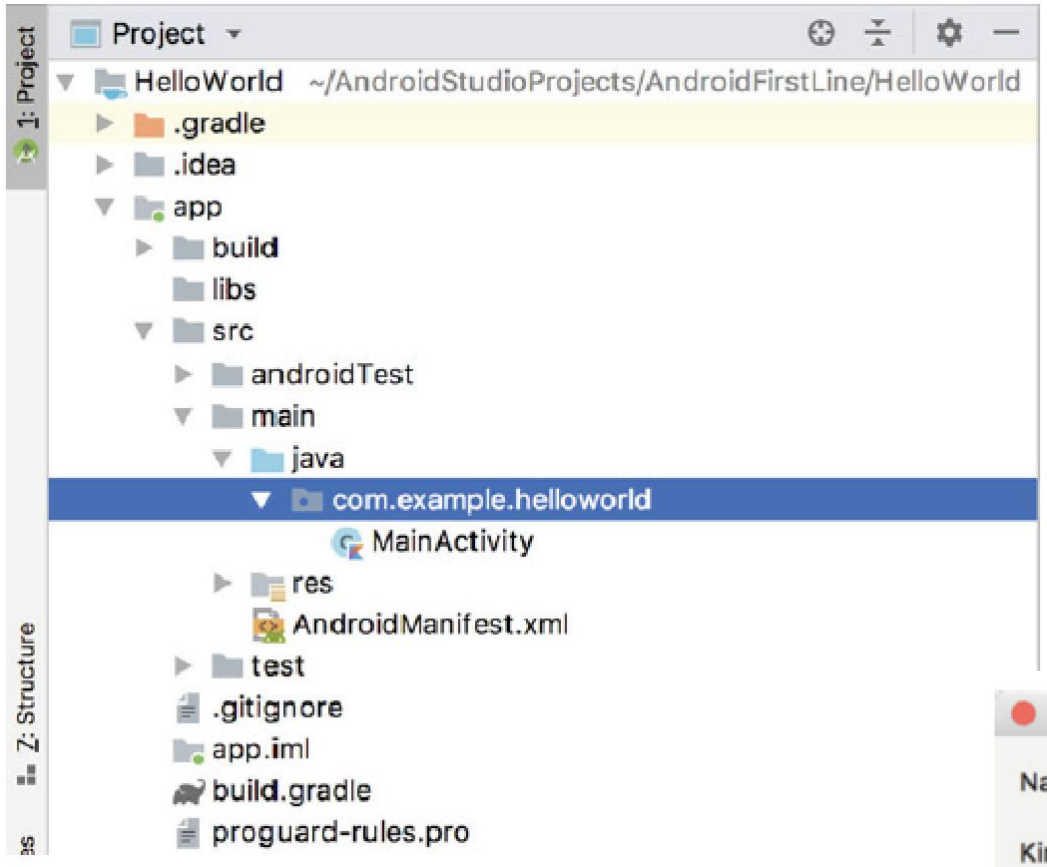
Let us just open the HelloWorld project

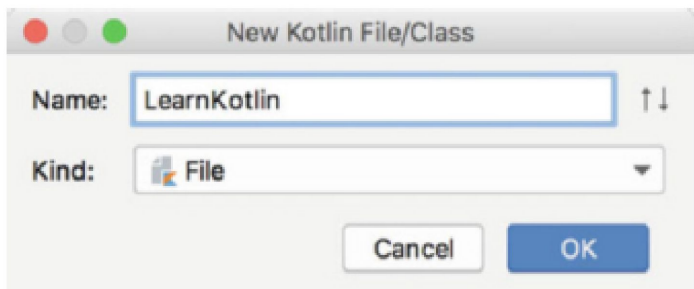
create a *myKotlin1* file under the same package as MainActivity.

Right click com.example.helloworld New Kotlin File/Class.

In the dialog, type *myKotlin1*

Click “OK” to finish creating the file.


1: Project
Project
HelloWorld ~/AndroidStudioProjects/AndroidFirstLine/HelloWorld
.gradle
.idea
app
build
libs
src
androidTest
main
java
com.example.helloworld
MainActivity
res
AndroidManifest.xml
test
.gitignore
app.iml
build.gradle
proguard-rules.pro
2: Structure


New Kotlin File/Class
Name:
LearnKotlin
Kind:
File
Cancel
OK


LearnKotlin.kt
1
2
3
4
5
package com.example.helloworld
fun main() {
println("Hello Kotlin!")
}


Run:
com.example.helloworld.LearnKotlinKt
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Hello Kotlin!
Process finished with exit code 0
TODO
Terminal
Build
6: Logcat
4: Run

The Foundation of Programming: Variables and Functions

Variables

Kotlin only allows you to use two keywords when declaring a variable:
val and var.

val (short for value) is used to declare a value variable that is immutable.

This kind of variable cannot be assigned to another value after initialization,
corresponds to the final variable in Java.

var (short for variable) is used to declare a mutable variable,

which means that this variable can change to another value after initialization which corresponds to non-final variables in Java

you may wonder how can the compiler know which type is the variable if only these two key words are available?

This has something to do with Kotlin type inference

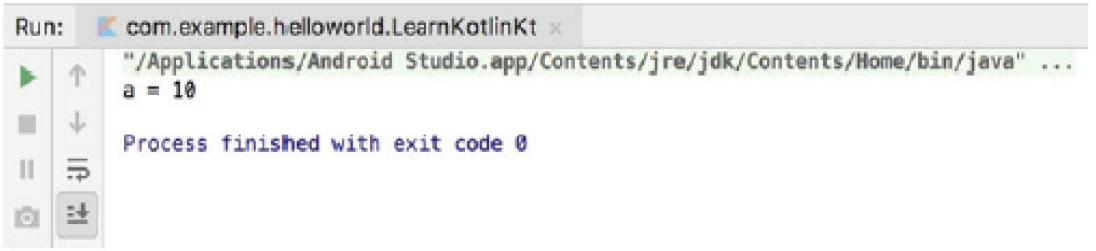
```
fun main() {
val a = 10
println("a = " + a)
}
```

Notice that there is no semi-colon at the end of statements, this is something you need to adapt if you are used to Java

In the code above, we use val to declare a variable a and assign value of 10 to a,
a will be inferred as an integer variable.

if you assign an integer value to a variable, then it has to be integer type.

If you assign a string to a then a has to be string type. This is how Kotlin type inference works.



However, type inference doesn’t always work in Kotlin.
For instance, if we late initiate a variable, Kotlin cannot automatically infer the type of the variable, and we need to declare the type.

The corresponding syntax is as follows:
val a: Int =10

we declare a as integer type, and Kotlin will no longer try to infer type for this variable.

If you assign a string to a, compiler will throw out type mismatch exception

Kotlin totally eliminates the primitive types in Java and treats everything as class.

Java int is a key word, but
Kotlin Int is a class that has its own methods and inheritance relationships

Java primitives	Kotlin class types	Data type
Int	Int	Integer
Long	Long	Long
Short	Short	Short
Float	Float	Float
Double	Double	Double
Boolean	Boolean	Boolean
Char	Char	Character
Byte	Byte	Byte

Table 2.1 Java and Kotlin data types comparison

Now let us try to operate on these numbers. Let us try to make a 10 times larger, and you may have the following code:

```
fun main() {
  val a: Int = 10
  a = a * 10
  println("a = " + a)
}
```

compiler will throw an error: Val cannot be reassigned.
It tells us that the variable declared by val cannot be reassigned.

To solve this problem, as mentioned before, val is used to declare an immutable variable, and var. is used to declare a mutable variable; thus, we only need to replace val with var., as shown below:

```
fun main() {
  var a: Int = 10
  a = a * 10
  println("a = " + a)
}
```

In Java, unless you declare final for a variable, it will be mutable, and this is not a good idea.

When the project is getting more complex, an increasing number of people join the team, and you will not be able to know when a mutable variable will be mutated by whom

though this variable should never be mutated which leads to some really hard-to-debug bugs.

Thus, a good programming habit is to add final to all the variables unless a variable absolutely needs to be mutated.

Functions

the main() function we used previously is a special function that serves as the entry point of the program.

This means that when the whole program gets started, it will start running main() function first

Kotlin also allows user to define functions freely. The syntax is as follows:

```
fun methodName(param1: Int, param2: Int): Int {
return 0
}
```

we need to use fun(short for function)keyword to declare a function.

Between pair of parentheses you can declare params that this function will take, and you can declare as many params as you want to

The way to declare param is “param name: param type“.

If the function doesn’t need to take param, then just keep it empty between parentheses.

The return type of a function which is before the right brace is optional
If your function doesn’t need to return any data, then you can omit this part.

Functions

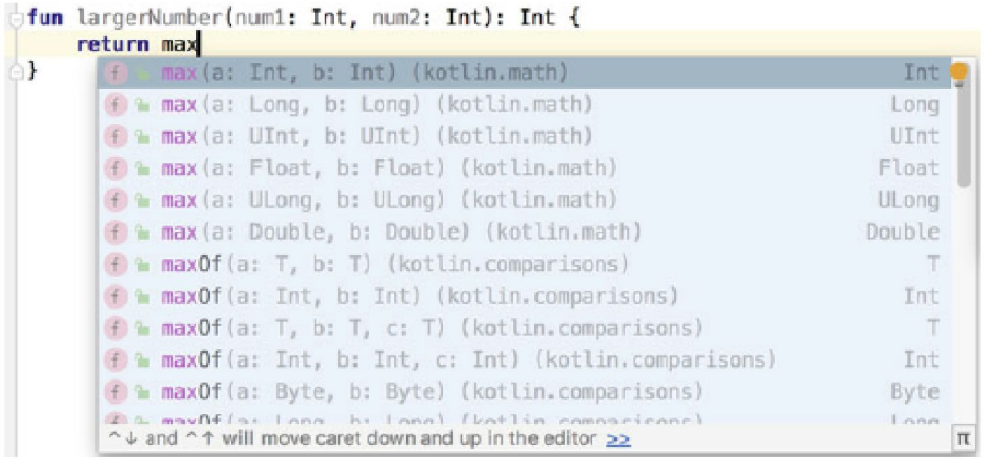
Between the braces are the contents of the function body which expresses the logic of the function.

```
fun largerNumber(num1: Int, num2: Int): Int {
  return max(num1, num2)
}
```

this function used a function called max() which is an internal function provided by Kotlin that will return the larger number of two arguments, thus largerNumber() is actually a wrapper for the internal function max().

When you start to type max, Android Studio will have the prompt as shown

Android Studio has excellent support for code autocompletion



```
fun main() {
  val a = 37
  val b = 40
  val value = largerNumber(a, b)
  println("larger number is " + value)
}

fun largerNumber(num1: Int, num2: Int): Int {
  return max(num1, num2)
}
```

if you manually type max(), there will be red error
This is because you didn’t import the package that has max() function.

There are a few ways to import the package.

By moving the cursor to the red error, you will see the shortcut of importing the package, but the best way to do so is always using the autocompletion as it is done automatically.

use the autocompletion to write the max() function and you will find that at the header of the myKotlin1 file, max package gets imported and no more errors, as shown in Fig

```
1   package com.example.helloworld
    java.lang.Integer.max? (multiple choices...) ⌥⏎
4   fun largerNumber(num1: Int, num2: Int): Int {
5       return max(num1, num2)
6   }
```

```
package com.example.helloworld

import kotlin.math.max

fun largerNumber(num1: Int, num2: Int): Int {
    return max(num1, num2)
}
```

```
Run:  com.example.helloworld.LearnKotlinKt
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
larger number is 40

Process finished with exit code 0
```

When there is only one line of code in a function, Kotlin allows to omit the braces by using an equal sign.

Let's use largerNumber() function as an example, and it can be simplified like the code below:

```
fun largerNumber(num1: Int, num2: Int): Int = max(num1, num2)
```

By using this syntax, even the return keyword can get omitted as the equal sign is enough to serve the same purpose.

Since max() returns Int type, and largerNumebers () just returns the value of max(), then Kotlin will infer that largerNumber() will also return an Int type.

Thus, there is no need to explicitly define the type of the return data, and code can be further simplified to the following:

```
fun largerNumber(num1: Int, num2: Int) = max(num1, num2)
```

Flow Control

Code can run in serial, conditionally and repeatedly

we need to
introduce the conditional statements and loop statements

if Statement

There are two ways to make condition check in Kotlin:
if and when

```
fun largerNumber(num1: Int, num2: Int): Int {
var value = 0
if (num1 > num2) {
   value = num1
} else {
    value = num2
}
return value
}
```

In Kotlin if has one extra function compared with Java.

It can return value of the if statement

```
fun largerNumber(num1: Int, num2: Int): Int {
val value = if (num1 > num2) {
  num1
} else {
  num2
}
return value
}
```

if uses the value to return and assign to val value.

Since there is no reassigning, we can now use val to define value.

We can simplify the code even further as follows:

```
fun largerNumber(num1: Int, num2: Int): Int {
return if (num1 > num2) {
  num1
 } else {
  num2
 }
}
```

but there is still room for improvement.

```
fun largerNumber(num1: Int, num2: Int) = if (num1 >
num2) {
 num1
} else {
 num2
}
```

And you can make it even simpler by compressing it into truly one line:

```
fun largerNumber(num1: Int, num2: Int) = if (num1 > num2) num1 else num2
```

when Statement

when statement in Kotlin is like the switch statement in Java but way more powerful

In Java, there are limitations of switch statement.

```
1.switch(expression){
2.case value1:
3. //code to be executed;
4. break;  //optional
5.case value2:
6. //code to be executed;
7. break;  //optional
8.....
9.
10.default:
11.  code to be executed if all cases are not matched;
12.}
```

when Statement

First, switch can only take integer or type that is shorter than integer for condition check.

Although JDK 1.7 and later versions started to support strings, if other data types are to be used for condition check, switch won't support it.

Secondly, after each case in switch statement, we must add a break there, otherwise code will continue to execute
And this caused numerous bugs.

The when statement solves all the problems above and provides more powerful features and sometimes is even simpler and easier to use than if.

We're going to build a function that can be used to return a student's score with the student's name.

Let us first use if statement to implement this function and write the following code inside myKotlin2 file:

```
fun getScore(name: String) = if (name == "Tom") {
 86
} else if (name == "Jim") {
 77
} else if (name == "Jack") {
 95
} else if (name == "Lily") {
 100
} else {
 0
}
```

We define a getScore() function which will take in student's name as param and use if to get the student's score and return it.

The code above uses the single line function syntax sugar again.

Though it can work as expected, it looks super redundant to write so many if and else.

We should use when statement when there are many condition checks and let’s rewrite the code as follows:

```
fun getScore(name: String) = when (name) {
"Tom" -> 86
"Jim" -> 77
"Jack" -> 95
"Lily" -> 100
else -> 0
}
```

when statement allows using any data type as arguments and you can define the conditions inside the body of when statement, the format is:

Matching value -> {corresponding logic}

When you only have one line of code, {} can be omitted.

Besides data matching, when statement also allows type matching.

We use another example to explain what is type matching.

define a checkNumber() function as follows:

```
fun checkNumber(num: Number) {
when (num) {
  is Int -> println("number is Int")
  is Double -> println("number is Double")
else -> println("number is not supported")
}
}
```

In the code above, *is* keyword is the key for type matching which is analogous to *instanceof* in Java.

checkNumber function takes a param of Number type
and it is an abstract class in Kotlin.

Number-related types like Int, Long, Float and Double are all sub class of Number class.

Thus, we can use type matching to decide what type of the argument is, and if it is Int or Double, then, print the type name

otherwise print this type is not supported.

Now we can call checkNumber() inside the main() as shown below:

```
fun main() {
val num = 10
checkNumber(num)
}
```

Now let’s change to Long type:

```
fun main() {
val num = 10L
checkNumber(num)
}
```

There is another way of using when statement that is not used very often but can be extensible in certain scenarios.
We use getScore() function as an example,

if we don’t pass in the argument we can
write like this:

```
fun getScore(name: String) = when {
name == "Tom" -> 86
name == "Jim" -> 77
name == "Jack" -> 95
name == "Lily" -> 100
else -> 0
}
```

We simply explicitly do condition check inside the statement body,

also notice that in Kotlin we can use == to determine if two strings or objects are equal instead of using equals() as Java does.

Assume the score of all the students' name that start with Tom is 86,

then we cannot use when statement with the argument.

Instead we can write something like the following:

```
fun getScore(name: String) = when {
name.startsWith("Tom") -> 86
name == "Jim" -> 77
name == "Jack" -> 95
name == "Lily" -> 100
else -> 0
}
```

Now whether the argument is Tom or Tommy, anyone whose name starts with Tom
will have score of 86.

Loop Statement

There are mainly two kinds of loop statement in Java:
while and for.

Kotlin also provides while and for loop,
and while loop is the same as in Java,

how to use for loop inside Kotlin.

Kotlin made many improvements to the syntax of for loop,
for example it eliminates the for-i loop; instead of using for-each loop,
Kotlin uses for-in loop.

we need to introduce the concept of range which doesn’t exist in Java.
For instance, we can use the following Kotlin code to represent a range:

```
val range = 0..10
```

It creates a closed range between 0 and 10

which means 0 and 10 are all included in this range

and in mathematical representation it is [0,10].

The keyword to create range is .. and by specifying the two ends, we can create a range.

With that, we can use for-in to iterate the range and write code inside main() functions

```
fun main() {
for (i in 0..10) {
    println(i)
  }
}
```

In most cases, closed range is not as useful as semi-open range.

Why? This is because the index of arrays starts with 0 and for an array of length 10,
its index range is from 0 to 9

and a left-closed and right-open range is more common in programming.
until keyword in Kotlin is used to create a range that is left closed and right open as shown below:

```
val range = 0 until 10
```

mathematical representation [0,10) by replacing the .. operator with until.

Run the code again, you will find that it won't print 10.

By default, for-in loop will increase by 1 in every iteration which is equivalent to i++ in Java's for-i loop

if step over of certain elements is necessary , *step* keyword is used

```
fun main() {
for (i in 0 until 10 step 2) {
    println(i)
  }
}
```

when iterating through [0,10), the step difference is 2
and is equivalent to i = i + 2 in for-i loop.

Notice that by using .. operator and until keyword, we are creating a range

that the right side is larger than the left side which is an ascending interval.

For create a descending interval, you should use downTo keyword as follows:

```
fun main() {
for (i in 10 downTo 1) {
    println(i)
  }
}
```

Here, we created a [10, 1] descending interval

The step keyword can also be applied to descending interval

To read a line of string in Kotlin, you can use readline() function.

Take input from user using readline() method –

```
fun main(args : Array<String>) {
    print("Enter text: ")
    var input = readLine()
    print("You entered: $input")
}
```

If you are taking input from user other than String data type, you need to use Scanner class.

To use Scanner first of all you have to import the Scanner on the top of the program.

```
import java.util.Scanner
```

```
fun main(args: Array<String>) {

    // create an object for scanner class
    val number1 = Scanner(System.`in`)
    print("Enter an integer: ")
    // nextInt() method is used to take
    // next integer value and store in enteredNumber1 variable
    var enteredNumber1:Int = number1.nextInt()
    println("You entered: $enteredNumber1")

    val number2 = Scanner(System.`in`)
    print("Enter a float value: ")

    // nextFloat() method is used to take next
    // Float value and store in enteredNumber2 variable
    var enteredNumber2:Float = number2.nextFloat()
    println("You entered: $enteredNumber2")

    val booleanValue = Scanner(System.`in`)
    print("Enter a boolean: ")
    // nextBoolean() method is used to take
    // next boolean value and store in enteredBoolean variable
    var enteredBoolean:Boolean = booleanValue.nextBoolean()
    println("You entered: $enteredBoolean")
}
```

Output:

```
Enter an integer: 123
You entered: 123
Enter a float value: 40.45
You entered: 40.45
Enter a boolean: true
You entered: true
```

Take input from user without using the Scanner class –
Here, we will use readline() to take input from the user and no need to import Scanner class.

readline()!! take the input as a string and followed by (!!) to ensure that the input value is not null.

```
fun main(args: Array<String>) {

    print("Enter an Integer value: ")
    val string1 = readLine()!!

    // .toInt() function converts the string into Integer
    var integerValue: Int = string1.toInt()
    println("You entered: $integerValue")

    print("Enter a double value: ")
    val string2= readLine()!!

    // .toDouble() function converts the string into Double
    var doubleValue: Double = string2.toDouble()
    println("You entered: $doubleValue")
}
```

```java
// Fig. 7.2: InitArray.java
// Creating an array.

public class InitArray
{
   public static void main( String args[] )
   {
      int array[]; // declare array named array

      array = new int[ 10 ]; // create the space for array

      System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings

      // output each array element's value
      for ( int counter = 0; counter < array.length; counter++ )
         System.out.printf( "%5d%8d\n", counter, array[ counter ] );
   } // end main
} // end class InitArray
```

Arrays in Kotlin

In Kotlin, arrays are not a native data type,
but **a mutable collection of similar items which are represented by the Array class**.

There are two ways to define an array in Kotlin. We can use the library function arrayOf() to create an array by passing the values of the elements to the function.

```
val num = arrayOf(1, 2, 3, 4) //implicit type declaration
```

```
val num = arrayOf<Int>(1, 2, 3) //explicit type declaration
```

```
fun main()
{
    // declaring an array using arrayOf()
    val arrayname = arrayOf(1, 2, 3, 4, 5)
    for (i in 0..arrayname.size-1)
    {
        print(" "+arrayname[i])
    }
    println()
    // declaring an array using arrayOf<Int>
    val arrayname2 = arrayOf<Int>(10, 20, 30, 40, 50)
    for (i in 0..arrayname2.size-1)
    {
        print(" "+arrayname2[i])
    }
}
```

Output:

```
1 2 3 4 5
10 20 30 40 50
```

Using the Array constructor –
Since Array is a class in Kotlin, we can also use the Array constructor to create an array.

The constructor takes **two** parameters:

1. The size of the array, and
2. A function which accepts the index of a given element and returns the initial value of that element.

Syntax:

```
val num = Array(3, {i-> i*1})
```

Kotlin program of creating array using constructor –

```
fun main()
{
    val arrayname = Array(5, { i -> i * 1 })
    for (i in 0..arrayname.size-1)
    {
        println(arrayname[i])
    }
}
```

Output:

```
0
1
2
3
4
```

```
package com.zetcode

import java.util.Arrays

fun main()
{ val nums = arrayOf(1, 2, 3, 4, 5)
val nums3 = IntArray(5, { i -> i * 2 + 3})

println(Arrays.toString(nums3))

}
```

```
val nums3 = IntArray(5, { i -> i * 2 + 3})
```

This line creates an array with `IntArray`. It takes the number of elements and a factory function as parameters.

```
[1, 2, 3, 4, 5]

[3, 5, 7, 9, 11]
```

```
package com.zetcode

import java.util.Arrays

fun main() {

    val nums = arrayOf(1, 2, 3, 4, 5)
    println(nums.get(0))

    nums.set(0, 0)
    println(Arrays.toString(nums))

    val nums2 = nums.plus(1)
    println(Arrays.toString(nums2))

    val slice = nums.sliceArray(1..3)
    println(Arrays.toString(slice))

    println(nums.first())
    println(nums.last())
    println(nums.indexOf(5))
}
```

we retrieve and modify array elements, create a slice, and get an index of an element.

```
println(nums.get(0))
```

We get an element with index 0 using the `get` function.

```
nums.set(0, 0)
```

The `set` method sets the array element at the specified index to the specified value.

```
val nums2 = nums.plus(1)
```

We add a new element to the array, creating a new array. (Remember that arrays are fixed sized; therefore, a new array was created).

```
val slice = nums.sliceArray(1..3)
```

With the `sliceArray` method, we create a slice from the array. The indexes are both inclusive.

```
println(nums.first())
println(nums.last())
```

We get the first and the last element of the array.

```
println(nums.indexOf(5))
```

We get the index of the first occurrence of the element 5.

Kotlin array indexing

```
package com.zetcode

fun main() {

    val nums = intArrayOf(1, 2, 3, 4, 5)

    println(nums[2])

    nums[0] = 11
    println(nums[0])
}
```

We use the indexing operations to get and modify an array value.
`println(nums[2])`

Kotlin array aggregate functions

Aggregates.kt

```
package com.zetcode

fun main() {

    val nums = intArrayOf(1, 2, 3, 4, 5)

    val avg = nums.average()
    println("The average is $avg")

    val nOfValues = nums.count()
    println("There are $nOfValues elements")

    val sumOfValues = nums.sum()
    println("The sum of values is $sumOfValues")

    val maxValue = nums.max()
    println("The maximum is $maxValue")

    val minValue = nums.min()
    println("The minimum is $minValue")
}
```

The example computes the average, sum, maximum, minimum, and the size of an array.

```
val avg = nums.average()
```
The `average` function calculates the average of the array values.

```
val nOfValues = nums.count()
```
The number of elements is determined with `count`.

```
The average is 3.0
There are 5 elements
The sum of values is 15
The maximum is 5
The minimum is 1
```

This is the output.

Kotlin array shuffle

The `shuffle` function randomly rearranges the array elements in-place.

Shuffling.kt

```
package com.zetcode

fun main() {

    val nums = arrayOf(1, 2, 3, 4, 5, 6)

    println(nums.joinToString())

    nums.shuffle()
    println(nums.joinToString())

    nums.shuffle()
    println(nums.joinToString())
}
```

In the example, we shuffle array elements twice. The `joinToString` function transforms the array to a string.

```
1, 2, 3, 4, 5, 6
1, 3, 5, 6, 4, 2
3, 2, 1, 5, 4, 6
```

Initialization of arrays using iterators

```
val squares = Array(5) { it * it }
```

Kotlin array map

The `map` function returns a list containing the results of applying the given transform function to each element of the array.

MapFun.kt

```
package com.zetcode

fun main() {

    val nums = arrayOf(1, 2, 3, 4, 5, 6)

    val res = nums.map { e -> e * 2 }
    println(res)
}
```

We have a list of integers. With `map`, each integer is multiplied by two.

```
[2, 4, 6, 8, 10, 12]
```

Kotlin random array elements

RandomInts.kt

```
package com.zetcode

import kotlin.random.Random

fun main() {

    val vals = Array(10) { Random.nextInt(0, 100)}
    println(vals.joinToString())
}
```

An array of ten random integers is created. Random integers are produced with Random.nextInt function.

Kotlin array traversal

Traverse.kt

```
package com.zetcode

fun main() {

    val nums = arrayOf(1, 2, 3, 4, 5, 6, 7)

    nums.forEach({ e -> print("$e ") })

    println()

    nums.forEachIndexed({i, e -> println("nums[$i] = $e")})

    for (e in nums) {
        print("$e ")
    }

    println()

    val it: Iterator<Int> = nums.iterator()

    while (it.hasNext()) {

        val e = it.next()
        print("$e ")
    }
}
```

The example loops over an array using four different ways of traversal.

```
nums.forEach({ e -> print("$e ") })
```

It traverses the array with forEach. This method applies an action on each element of the array.

```
nums.forEachIndexed({i, e -> println("nums[$i] = $e")})
```

The forEachIndexed performs the given action on each element, providing sequential index with the element.

```
for (e in nums) { print("$e ") }
```

Traversal over the array in a for loop.

```
val it: Iterator<Int> = nums.iterator()

while (it.hasNext())
{
   val e = it.next()
   print("$e ")
 }
```

Finally, the array is traversed with an iterator and a while loop.

```
1 2 3 4 5 6 7
nums[0] = 1
nums[1] = 2
nums[2] = 3
nums[3] = 4
nums[4] = 5
nums[5] = 6
nums[6] = 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
```

Sorting arrays in Kotlin

ArraySort.kt

```
package com.zetcode

fun main() {

    val nums = arrayOf(7, 3, 3, 4, 5, 9, 1)

    val sortedNums = nums.sortedArray()
    println(Arrays.toString(sortedNums))

    val sortedNumsDesc = nums.sortedArrayDescending()
    println(Arrays.toString(sortedNumsDesc))
}
```

The example sorts an array is ascending order with sortedArray and descending order with sortedArrayDescending. The methods create new sorted arrays.

```
[1, 3, 3, 4, 5, 7, 9]
[9, 7, 5, 4, 3, 3, 1]
```

This is the output.

Kotlin two-dimensional arrays

ArrayTwoDim.kt

```
package com.zetcode

fun main() {

    val array = arrayOf(intArrayOf(1, 2),
            intArrayOf(3, 4),
            intArrayOf(5, 6, 7))

    println(Arrays.deepToString(array))
}
```

The example creates a two-dimensional array by nesting intArrayOf function calls into the arrayOf function.

```
[[1, 2], [3, 4], [5, 6, 7]]
```

Declare and initialize a 2-dimensional array in Kotlin

1. Using arrayOf() function. The most common way to declare and initialize arrays in Kotlin is the arrayOf() function. ...

2. Array constructor. ...

3. Using arrayOfNulls() function. ...

4. Jagged Array.

Using arrayOf() Function

```
fun main() {
    var arr = arrayOf(int ArrayOf(1, 2, 3), int ArrayOf(4, 5, 6), int ArrayOf(7, 8, 9))

    for (row in arr) {
        println(row.contentToString())
    }
}
```

The above code will result in the following matrix:

[1, 2, 3]
[4, 5, 6]
[7, 8, 9]

Array constructor

```
fun main() {
    val rows = 3
    val cols = 4

    val arr = Array(rows) { IntArray(cols) }
    for (row in arr) {
        println(row.contentToString())
    }
}
```

This will result in a 3×4 matrix with a default value of 0 assigned to each element since no initializer is supplied.

```
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

```
val array2dInts = Array<T>[n][m] //with optional initialization for obvious types
```

So if you have Numeric types or Strings it will initialize it with 0 and empty string respectively.

If my array has a type that requires a call to a constructor with some parameters then it can be this:

```
val array2dChessBoard = Array<ChessPiece>[8][8] { ChessPiece(name = "Bishop") }
```

To initialize the matrix with a fixed value, you can do like:

```
fun main() {
    val rows = 3
    val cols = 4

    val arr = Array(rows) { IntArray(cols) { 1 } }
    for (row in arr) {
        println(row.contentToString())
    }
}
```

This will result in a 3×4 matrix with all elements initialized with value 1.

```
[1, 1, 1, 1]
[1, 1, 1, 1]
[1, 1, 1, 1]
```

You can also initialize your matrix with a lambda, as shown below.
This is often useful to fill the matrix with some dynamic value based on the x and y index

```
fun main() {
    val rows = 3
    val cols = 4

    val arr = Array(rows) { r -> IntArray(cols) { c -> r + c } }
    for (row in arr) {
        println(row.contentToString())
    }
}
```

```
[0, 1, 2, 3]
[1, 2, 3, 4]
[2, 3, 4, 5]
```

IntArray is used for integers, DoubleArray for double, LongArray for long, CharArray for Chars

Using arrayOfNulls() function

```
fun main() {
    val arr = arrayOfNulls<IntArray>(3)

    arr[0] = intArrayOf(1, 2, 3)
    arr[1] = intArrayOf(4, 5, 6)
    arr[2] = intArrayOf(7, 8, 9)

    for (row in arr) {
        println(row?.contentToString())
    }
}
```

arrayOfNulls() function is used declare a two-dimensional array by only specifying the first dimension:

Jagged Array

Finally, you can also initialize the columns with different lengths.
The resultant array is called a **jagged array**, whose elements can have different dimensions and sizes. Following is a simple example demonstrating the jagged arrays:

```
fun main() {
    val arr = arrayOf(intArrayOf(1, 2, 3), intArrayOf(4, 5),
intArrayOf(6, 7, 8, 9))

    for (row in arr) {
        println(row.contentToString())
    }
}
```

This will result in the following two-dimensional ragged array:

```
[1, 2, 3]
[4, 5]
[6, 7, 8, 9]
```

Copy a two-dimensional array in Kotlin

The idea is to iterate over each row of the original array and then call the clone() function to clone each row.

```
fun clone(array: Array<IntArray>?): Array<IntArray?>? {
    if (array == null) {
        return null
    }
    val clone = arrayOfNulls<IntArray>(array.size)
    for (i in array.indices) {
        clone[i] = array[i].clone()
    }
    return clone
}

fun main() {
    val array = arrayOf(intArrayOf(1, 2, 3, 4, 5), intArrayOf(6, 7, 8, 9),
            intArrayOf(10, 11, 12))

    val clone = clone(array)
    clone?.forEach { println(it?.contentToString()) }
}
```

The Nullability Problem

Null represents the absence of a value in a programming language. A null reference typically means that a variable or object doesn't point to any valid data in memory, and it's often used to represent missing or undefined values. Incorrectly handling null references can lead to crashes, runtime errors, and unexpected behavior in a program. Kotlin addresses this problem by distinguishing between nullable and non-nullable types.

In Kotlin, we can explicitly mark a variable or a property as nullable by appending the nullable operator to its type:

```
val age: Int
val name: String?
```

In this example, name is a nullable variable, while age is a non-nullable variable. This distinction helps the compiler catch potential nullability issues at compile time.

The Double-Bang (!!) Operator

The double-bang (!!) operator, also referred to as the not-null assertion operator, is used to tell the compiler that we are certain a nullable type is not null at a given point in the code.
When we use the double-bang, we're telling the compiler to suppress its null-safety checks:

```
val name: String? = null
val length = name!!length
```

We’re using the double-bang operator to assert that name is not null. However, since name is explicitly initialized as null, this code will throw a NullPointerException at runtime because we’re attempting to access the length property on a null reference.

Unit testing this code will involve checking whether the code behaves as expected, which, in this case, would be to verify that it indeed throws a NullPointerException when name is null:

```
@Test
fun testNullPointerException() {
    val name: String? = null
    assertFailsWith<NullPointerException> {
        val length = name!!.length
    }
}
```

We’ve intentionally set name to null, and then we use assertFailsWith() to confirm a NullPointerException is thrown when trying to access the length property with !!.

Kotlin filtering arrays

Filter.kt

```
package com.zetcode

fun main() {

    val nums = arrayOf(1, -2, 3, 4, -5, 7)

    nums.filter { e -> e > 0 }.forEach { e -> print("$e ") }
}
```

The example creates an array of positive and negative integers. The filter method is used to pick up only positive values.

```
1 3 4 7
```

Filter2.kt

```
package com.zetcode

data class User(val fname: String, val lname: String, val salary: Int)

fun main() {

    val users = arrayOf(

        User("John", "Doe", 1230),
        User("Lucy", "Novak", 670),
        User("Ben", "Walter", 2050),
        User("Robin", "Brown", 2300),
        User("Amy", "Doe", 1250),
        User("Joe", "Draker", 1190),
        User("Janet", "Doe", 980),
        User("Albert", "Novak", 1930)
    )

    users.filter { it.salary < 1000 }.forEach { e -> println(e) }
}
```

With filter, we find out all users whose salary is below 1000.

We have an array of users having firstname, lastname, and salary properties.

Kotlin array finding elements

We can find elements with find and findLast.

ArrayFind.kt

```
package com.zetcode

fun main() {

    val nums = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

    val firstEven = nums.find { it % 2 == 0 }
    println("The first even value is: $firstEven")

    val lastEven = nums.findLast { it % 2 == 0 }
    println("The last even value is: $lastEven")
}
```

```
The first even value is: 2
The last even value is: 8
```

The example looks for the first and last even values in the array.

Kotlin array reduction

Reduction is a terminal operation that aggregates array values into a single value.
The reduce method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

ArrayReduce.kt

```
package com.zetcode

fun main() {

    val nums = intArrayOf(2, 3, 4, 5, 6, 7)

    val total = nums.reduce { product, next -> product * next }

    println(total)
}
```

The product is the accumulator, the next is the next value in the array.

5040

We use the reduce method to calculate a product from array elements.

```
val total = nums.reduce { product, next -> product * next }
```

Kotlin array all

The all method returns true if all elements match the given predicate.

ArrayAll.kt

```
package com.zetcode

fun main() {

    val nums = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

    val hasAllEvens = nums.all { it % 2 == 0 }

    if (hasAllEvens) {

        println("The array contains only even values")
    } else {

        println("The array contains odd values")
    }
}
```

The example checks if the array elements are all even values.

```
The array contains odd values
```

Kotlin array any

The `any` method returns `true` if at least one of the elements matches the given predicate.

ArrayAny.kt

```
package com.zetcode

fun main() {

    val nums = intArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

    val hasEvenVals = nums.any { it % 2 == 0 }

    if (hasEvenVals) {

        println("The array contains even values")
    } else {

        println("The array does contain even values")
    }
}
```

The example checks if the array elements contains any even values.

```
The array contains even values
```

Kotlin lambda expression

A *lambda expression* is an anonymous function which is treated as a value.
It can be bound to a variable, passed to a function as a parameter, or returned from a function.

val square: (Int) -> Int = { e: Int -> e * e }
In Kotlin, a lambda expression is always delimited by curly braces.

Kotlin anonymous function

Main.kt

```
package com.zetcode

fun main() {

    val vals = intArrayOf(-2, -1, 0, 1, 2, 3, 4)

    val filtered = vals.filter(fun(e) = e > 0)
    println(filtered)
}
```

We define an array of integers.
The array is filtered with the filter function.
The filter function takes an anonymous function as a parameter.

```
val filtered = vals.filter(fun(e) = e > 0)
```

The anonymous function is used to filter the array.

We rewrite of the previous one using a lambda expression.

Main.kt

```
package com.zetcode

fun main() {

    val vals = intArrayOf(-2, -1, 0, 1, 2, 3, 4)

    val filtered = vals.filter { e -> e > 0 }
    println(filtered)
}
```

Type declaration

In a lambda expression type declaration, we have a list of parameters in square brackets followed by the arrow -> operator followed by the return type.

Main.kt

```
package com.zetcode

fun main() {

    val square: (Int) -> Int = { e: Int -> e * e }

    val r1 = square(5)
    val r2 = square(3)

    println(r1)
    println(r2)
}
```

we bind a lambda expression to a value; full type declarations are provided.

```
val square: (Int) -> Int = { e: Int -> e * e }
```

The square value is bound to a lambda expression, which accepts one integer and returns an integer.

Type inference

Kotlin can infer the data types of values and therefore, can omit some declarations.

Main.kt

```
package com.zetcode

fun main() {

    val square1: (Int) -> Int = { e: Int -> e * e }
    val square2 = { e: Int -> e * e }
    val square3: (Int) -> Int = { e -> e * e }
//    val square4 = { e -> e * e }

    val r1 = square1(5)
    val r2 = square2(3)
    val r3 = square3(6)

    println(r1)
    println(r2)
    println(r3)
}
```

In the example, the square function is defined multiple times.

```
val square1: (Int) -> Int = { e: Int -> e * e }
```

This is the full type declaration.

```
val square2 = { e: Int -> e * e }
```

Here the lambda declaration is omitted for the square2 name.

```
val square3: (Int) -> Int = { e -> e * e }
```

In this case, we omit the declaration of the element inside the lambda expression.

```
// val square4 = { e -> e * e }
```

However, we cannot omit both declarations. This code does not compile.

The Unit type is used for an expression that does not return a value.

Main.kt

```
package com.zetcode

fun main() {

    val l1 = { println("Hello there!") }
    val l2: (String) -> Unit = { name: String ->
        println("Hello $name!")
    }

    l1()
    l2("Lucia")
}
```

If we print something to the console, we do not return anything.
For such cases, we specify Unit.

Kotlin lambda expression it

The ***it*** is a special keyword that represents a single parameter passed to the lambda expression.

Main.kt

```
package com.zetcode

fun main() {

    val nums = listOf(1, 2, 3, 4, 5, 6)
    nums.forEach { println(it * 2) }
}
```

We have a list of integers.
With forEach, we go through the list of elements and multiply them by two.

```
nums.forEach { println(it * 2) }
```

The ***it*** represents the currently processed item.

Passing lambdas as function arguments

Main.kt

```
package com.zetcode

val inc = { e: Int -> e + 1 }
val dec = { e: Int -> e - 1 }
val square = { e: Int -> e * e }
val triple = { e: Int -> e * e * e }

fun doProcess(vals: List<Int>, f: (Int) -> Int) {

    val processed = vals.map { e -> f(e) }
    println(processed)
}

fun main() {

    val vals = listOf(1, 2, 3, 4, 5, 6)

    doProcess(vals, inc)
    doProcess(vals, dec)
    doProcess(vals, square)
    doProcess(vals, triple)
}
```

We define four lambdas: inc, dec, square, triple.
We pass the lambdas to the doProcess function.

```
fun doProcess(vals: List<Int>, f: (Int) -> Int) {

    val processed = vals.map { e -> f(e) }
    println(processed)
}
```

We provide the type declaration for the second parameter: (Int) -> Int.
Each lambda takes an integer and returns an integer.

```
val processed = vals.map { e -> f(e) }
```

With map, we apply the lambda expression on each element of the list.

Returning from lambda

Main.kt

```
package com.zetcode

val check = { u:Pair<String, Int> ->

    when (u.second) {
        in 0..75 -> "failed"
        else -> "passed"
    }
}

fun main() {

    val students =  listOf(

        Pair("Maria", 98),
        Pair("Pablo", 81),
        Pair("Lucia", 45),
        Pair("Peter", 98),
        Pair("Simon", 73),
    )

    students.forEach {

        val res = check(it)
        println("${it.first} has $res")
    }
}
```

The last expression in the lambda is returned

In the example, we have a list of students. We check which students have passed the exam.

```
val check = { u:Pair<String, Int> ->

    when (u.second) {
        in 0..75 -> "failed"
        else -> "passed"
    }
}
```

The lambda contains a when expression. The matched arm's value is returned from the lambda.

```
val res = check(it)
println("${it.first} has $res")
```

The returned value is used to display a message.

Trailing lambdas

Main.kt

```
package com.zetcode

data class User(val fname: String, val lname: String, val salary: Int)

fun main() {

    val users = listOf(
        User("John", "Doe", 1230),
        User("Lucy", "Novak", 670),
        User("Ben", "Walter", 2050),
        User("Robin", "Brown", 2300),
        User("Amy", "Doe", 1250),
        User("Joe", "Draker", 1190),
        User("Janet", "Doe", 980),
        User("Albert", "Novak", 1930)
    )

    val r1 = users.maxBy({ u: User -> u.salary })
    println(r1)

    val r2 = users.maxBy() { u: User -> u.salary }
    println(r2)

    val r3 = users.maxBy { u: User -> u.salary }
    println(r3)
}
```

If the last parameter of a function is a function, then a lambda expression can be placed outside the parentheses.

If the lambda is the only argument, the parentheses can be omitted entirely.

In the example, the maximum salary of all users are found .

```
val r1 = users.maxBy({ u: User -> u.salary })
println(r1)
```

In the first case, we pass the lambda expression to the maxBy function as a parameter.

```
val r2 = users.maxBy() { u: User -> u.salary }
println(r2)
```

Since the lambda is the last parameter, we can take it out of the parentheses.

```
val r3 = users.maxBy { u: User -> u.salary }
println(r3)
```

Since the lambda is the only parameter, we can omit the parentheses.

Kotlin chaining functions with lambdas

We can chain function calls with lambda expressions, creating succinct code.

Main.kt

```
package com.zetcode

fun main() {

    val words = listOf("sky", "cup", "water", "den",
        "knife", "earth", "falcon")

    val res = words.filter { it.length == 5 }.sortedBy { it }
        .map { it.replaceFirstChar(Char::titlecase) }
    println(res)
}
```

We filter a list of words, sort it, and capitalize its elements.

All is done in a chain of three function calls. The functions have trailing lambdas.

Destructuring in lambdas

Main.kt

```
package com.zetcode

fun main() {

    val words = mapOf(
        1 to "sky", 2 to "cup", 3 to "water", 4 to "den",
        5 to "knife", 6 to "earth", 7 to "falcon"
    )

    words.forEach { (_, v) -> println(v) }
}
```

We have a map of words. We go through the map with forEach. Each element of the map is destructured into a key/value pair. Since we do not use the key, we use the underscore character.

Parameters can be destructured in lambda expressions. For unused variables, we can use the underscore _ character.

Underscore for unused variables
Sometimes, it becomes necessary to ignore a variable in a destructuring declaration.

To do so, an **underscore** is put in place of its name.
In this case, the component function for the given variable isn't invoked.

Following the arrival of **Kotlin 1.1**, the destructing declaration syntax can be used for **lambda parameters** also.

If a lambda parameter has a parameter of **Pair** type or some other type that declares component functions,

then new parameters can be introduced by putting them in parenthesis. The rules are the same as defined above.

```
fun main(){
    val map = mutableMapOf<Int,String>()
    map.put(1,"Ishita")
    map.put(2,"Kamal")
    map.put(3,"Kanika")
    map.put(4,"Minal")
    map.put(5,"Neha")
    map.put(6,"Pratyush")
    map.put(7,"Shagun")
    map.put(8,"Shashank")
    map.put(9,"Uday")
    map.put(10,"Vandit")
    println("Initial map is")
    println(map)
    // Destructuring a map entry into key and values
    val newmap = map.mapValues { (key,value) -> "Hello ${value}" }
    println("Map after appending Hello")
    println(newmap)
}
```

Output:

```
Initial map is
{1=Ishita, 2=Kamal, 3=Kanika, 4=Minal, 5=Neha, 6=Pratyush, 7=Shagun, 8=Shashank,
9=Uday, 10=Vandit}
Map after appending Hello
{1=Hello Ishita, 2=Hello Kamal, 3=Hello Kanika, 4=Hello Minal, 5=Hello Neha, 6=Hello
Pratyush, 7=Hello Shagun, 8=Hello Shashank, 9=Hello Uday, 10=Hello Vandit}
```

If a component of the destructured parameter is not in use, we can replace it with the underscore to avoid calling component function:

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