KOTLIN

JETBRAINS

- Software company, founded 2000 in Prague.
- Known for IntelliJ IDEA and PyCharm
- Developer of Kotlin

TEST KOTLIN CODE

Unfortunately, you cannot create a terminal Kotlin app in Android Studio. However, two solutions exist for testing Kotlin code:

- Use https://play.kotlinlang.org
- Generate a new Android Studio project and implement your code in an Android Testfile.

OVERVIEW

- statically typed, compiles into Java bytecode.
- can be compiled into JavaScript!
- Developed by the St. Petersburg team of JetBrains.
- Kotlin: Island 32 km west of St. Petersburg in the Baltic Sea.
- Kotlin was first presented in 2011, version 1.0 was released in 2016.

- Packages, Import, Comments: see Java
- One addition: block comments can be nested

VARIABLES

```
//define a variable using var in Pascal style (with colon)
var x : Int = 1
var y = 2   //gets the type Int indirectly
var z1 : Int

//define a constant using val
val w : Int = 1

x += 3 // x is now 4
w += 5 // this will fail

print(x) //use print for console output
```

NUMBER DATA TYPES

Data types for Numbers:

- Double (64 Bit)
- Float (32 Bit)
- Long (64 Bit)
- Int (32 Bit)
- Short (16 Bit)
- Byte (8 Bit)

NUMBER CONVERSION I

Every number type supports the following conversions:

- toByte(): Byte
- toShort(): Short
- toInt(): Int
- toLong(): Long
- toFloat(): Float
- toDouble(): Double
- toChar(): Char (see below)

NUMBER CONVERSION II

```
var x = 10
var y = 6
var z = x.toDouble() //z is now 10.0
x = x / y //x is now 1 -> no implicit conversion to double!
```

STRINGS I

- Defined by type String
- Immutable
- Characters are stored as type char (similar to Java)
- A lot of api methods return a CharSequence instead of a String. Basic idea: You can use other CharSequence objects like StringBuffer or StringBuilder.

STRINGS II

```
var s : String = "Hallo"
var l : Int = s.length //=5
var c : Char = s.get(0) //='H'
var sc : String = s + s //= HalloHallo
var su : CharSequence = s.subSequence(0,2) //=Ha
//note that subSequence returns a CharSequence
//use .toString() to convert a CharSequence to a String

val t = "Hallo"
var b = s == t //=true (no isEqual)
```

STRING III

```
//You can use templates in Kotlin strings
val i = 99
var s : String = "Johnny ${i}" //or short $i
var t = s //=Johnny 99
//mask the $ if you need one:
print ("${'$'}123") //=$123

//templates also work with expressions
print("${s.length}") //=9

//Note that the template is compiled when the string
//is created
```

STRINGS IV

```
//There are also triple quoted strings that
//can contain newlines and any other characters
val tripleString = """
Hallo Welt
"""
//contains two newlines
```

NULL SAFETY (OPTIONALS) I

```
//Kotlin's type system distinguishes between references
//that can become null and those that can not.
//a nullable reference must end with a question mark
var s : String = null //won't compile
var s : String? = null //will compile

//if you have a nullable reference, use
//the (?.) to savely access properties
var l1 : Int? = s?.length //returns null if s is null

//the elvis operator ?: can be used to define a default
//value if the value is null
var j : Int? = 2
var i : Int = j ?: 0 //notice that i is not optional!
```

NULL SAFETY (OPTIONALS) II

```
//use the !! operator if you want an Int
var l2 : Int = s!!.length
//Note that this operator will throw an Exception
//if s is null

//save downcasting with as?
s = "10"
var i = s as? CharSequence
//will return null if downcasting fails
```

TYPE CHECKING

```
//is-operator will check the type of an instance
//and automatically cast to String (smart casting!)
if (obj is String) {
    print(obj.length) // x is automatically cast to String
    return
}

//(similar to) let in Swift
//the casting works even in a condition
if (obj is String && obj.length == 0) return
```

KOTLIN AND JAVA TYPES

Kotlin	Java
Int, Int?	int, Integer
Boolean, Boolean?	boolean, Boolean
Double, Double?	double, Double
String	String

IF AND ELSE

```
//if and else is similar to Java, but
//"if" is an expression (last expression of a block is returned)
var minimum = if (a < b) a else b
```

COMPARISON OPERATORS

- ==, != - equality operators (translated to calls of equals() for non-primitive types)
- <, >, >= and <= (translated to compareTo for non primitive types)
- [,] - indexed access operator (translated to calls of get and set)
- ===, !== - referential equality operators

LOOPS: FOR

```
//iterator based
//similar to C# or Python
for (item in collection) print(item)

//direct string iteration
for (c in "abc") print(c) //prints abc

//create your own range if you need an index
for (i in 0..5) {
 print(i)
}
```

RANGE EXPRESSION

```
//use a range expression to iterate
//over integers (start..stop)
for (i in range 1..3) {print(i)}
//will print 123
//if you want a negative step you need to use downTo
for (i in 3 downTo 1) {print(i)}
//will print 321
//add both with a step if needed
for (i in 3 downTo 1 step 2) {print(i)}
//will print 31
```

LOOPS: WHILE

```
//while and do ... while also as usual
var i : Int = 5
while (i > 0) {
 print(i)
}
```

EXERCISE: COUNT THE LETTER 'E'

Write three Kotlin programs that count the number of 'e's in a string. Try all three presented loops:

- while
- for direct
- for range

WHEN (SWITCHES)

```
when (api) {
    in 1..13 -> print("Probably not used anymore (0.3%)")
    24,25 -> print("Nougat")
    26 -> print("Oreo 8.0")
    27 -> print("Oreo 8.1")
    28 -> print("P")
    else -> { print("api level not found") }
}
```

COLLECTIONS

Kotlin provides 3 powerful collection classes:

- List: ordered collection with access to elements by indices
- Set: collection of unique elements
- Map: set of key-value pairs

All three can be used as readonly (default) or mutable (MutableList, MutableSet, MutableMap)

COLLECTION: LIST

```
//always use mutableList/listOf when creating a new list
var stringList : MutableList<String> = mutableListOf("a", "b", "c")
//create an empty mutableList with mutableListOf<String>()
stringList.add("d")
stringList.removeAt(1)
print(stringList) // will print ["a", "c", "d"]

var immutableList = listOf("Hallo", "Welt")
immutableList.add("xxx") //won't compile
```

COLLECTION: SET

```
//mutableSetOf and setOf also exist
var stringSet : MutableSet<String> = mutableSetOf("a", "b", "c")
print(stringSet.contains("d")) // will print false
stringSet.remove("c")
print(stringSet) // will print [a, b]
```

COLLECTION: MAP

```
var dict : MutableMap<String,Int> = mutableMapOf("A" to 1, "B" to 2)
dict["C"] = 3
dict.remove("A")
print(dict) // will print {B=2, C=3}
print(dict.keys) // will print [B, C]
print(dict.size) // will print 2
print(dict.containsKey("B")) //will print true
```

FUNCTIONS I

```
//use keyword fun to define a function
fun printString(message : String, tag : String) {
   print("$tag : $message")
}
//you have named parameters (works both):
printString(message = "Hallo Welt", tag = "APP")
printString(tag = "APP", message = "Hallo Welt")
```

FUNCTIONS II

```
//the default return type is Unit. You can define another
fun concatStrings(s1 : String, s2  : String) : String {
 return s1 + s2
}
```

EXCEPTIONS I

Work the same way as in Java (same syntax).

```
try {
  //your code that may throw an exception
  //comes here
}
catch(e: Exception) {
  //do something in case of an exception. As in Java,
  //all exceptions are descendants of Throwable
}
finally {
  //an optional final block that is executed in
  //any case
}
//However, there is one addition: try returns the last exception
//or expression. You can use it in the following way:
val a: Int? = try { parseInt(input) }
  catch (e: NumberFormatException) { null }
```

EXCEPTIONS II

```
try {
//Kotlin does not have checked exceptions like Java:
public static int parseInt(String s)
  throws NumberFormatException

//this will cause each usage of parseInt to either
//catch the exception or rethrow it
```

EXERCISE: CHAR COUNT

Write a Kotlin function charCount that can parse a text and print out how often each letter appears within the text.

```
val text = "Kotlin is a cross-platform, statically typed,
general-purpose programming language with type inference.
Kotlin is designed to interoperate fully with Java, and
the JVM version of Kotlin's standard library depends on
the Java Class Library, but type inference allows its
syntax to be more concise. On 7 May 2019, Google announced
that the Kotlin programming language is now its preferred
language for Android app developers."
val cc = charCount(text)
for (key in cc.keys) {
    print("\"${key}\" = ${dict[key]}, ")
}
// "K" = 4, "L" = 1, "M" = 2, "O" = 1, "V" = 1, "a" = 30,
// "b" = 4, "c" = 7, "d" = 13, "e" = 37, ...
```

CLASSES I

```
//start with keyword class and add a primary constructor. You can
//add multiple secondary constructors.
class Product constructor (name : String, price : Double) {
  //now build an init block that does the initialization of the objec
  //put everything in the init block except property initialization
  val name = name
  val price = price
  init {
  //put your initialization code here. All this stuff
  //is executed when the class is initialized
  }
}
```

CLASSES II

```
//You can add a secondary constructor if you like
class Product constructor (name : String, price : Double) {
  val name = name
  val price = price
  //we add a third property "description". However, we need to
  //initialize it.
  var description = ""
  constructor(name: String, price: Double, description : String)
    : this(name, price) {
  //as we now change the value of "description", the
  //property must not be constant (val)
    this.description = description
  }
}
```

PROPERTIES

```
//If you define your class with val parameters, you will
//get properties automatically.
class Product constructor (val name : String, val price : Double) {}
val p1 = Product("Notepad 1A", 1200)
print(p1.name)
//if you want to define your own getters and setters, use so called
//get or set accessors: (Note that visitCounter does not have
//var / val and is therefore not a property but a local parameter)
class Customer constructor(var name: String,  visitCounter : Int) {

 var visitCounter = visitCounter //define a property visitCounter
 set(value) { //restrict visitCounter to positive values
 if (value >= 0) field = value //field references the backing field
  //(writing visitCounter = value would result in calling the setter
  //recursively and resulting in StackOverflowError)
 }}
```

DATA CLASSES I

```
//if you define a class as data class,
//you will get toString, equals, copy and hashCode automatically
data class Customer (val name: String, val age: Int)
//some requirements:
//the primary constructor needs to have at least one parameter
//all primary constructor parameters need to be marked as val or var
//no abstract, open, sealed or inner data classes
//classes -> no inheritance
//
//Pair and Tuple are Kotlin standard data classes
```

DATA CLASSES II

```
//Within data classes, custom getter/setters are
//not directly possible. Generally, Kotlin recommends
//to switch back to regular class if you need it.
//Using a hack with a backing field works (but is ugly):
data class Customer(val name: String, private val _age: Int) {
 val age: Int
 get() = _age
 set(value) {
   _age = value
 }
}
```

EXERCISE: CHAR COUNT REFACTORING

Write a class TextStore that stores a text as attribute and provides a method charCount (see last exercise).

INHERITANCE

```
//by default, all classes are final. If you want to inherit use
//the open keyword (or sealed if all childs are within the same file
open class Product (val name : String, val price : Double) {
  open fun getCompletePrice() : Double {return this.price;}
}
//you can access the parent class methods and properties by using
//the "super" keyword. Note that we do not use val/var in
//this constructor as this would declare (and override) the existing
//properties from the parent class.
open class ExpensiveProduct(name : String, price : Double) :
 Product(name, price) {

 override fun getCompletePrice() : Double {
  return super.getCompletePrice() * 1.2
 }
}
```

OBJECTS

```
//Kotlin supports Singletons by defining an object
object SingletonObject {
 fun dothis() {
  print("Hello World")
 }
}

//use it
SingletonObject.dothis()
```

EXERCISE: TEXTSTORE LETTERS

Extend the TextStore class with another method letterCount. This method should only count the letters from A-Z (uppercase or lowercase) and return this result as dictionary. If a letter X is not in the text, the dictionary should contain an entry X with 0.

```
// You can iterate over the alphabet in the same way as in Java, ie.

var c = 'a'
while (c <= 'z') {
  print(c)
  c += 1
}
// The uppercase of a char can be retrieved with uppercaseChar():
val cu = "c".uppercaseChar() // cu is now "C"
```

EXTENSION FUNCTIONS

```
//we extend Int:
infix fun Int.percentOf(Int i) = (this * 100 / i)
//now available in the whole package

//and use it
val p = 10 percentOf 80 // p==8

//we can also extend a class with properties
private val Int.bd : BigDecimal
 get() {
  return BigDecimal(this)
 }

//and use it (convert an Int to BigDecimal)
val b : BigDecimal = p.bd
```

HIGHER ORDER FUNCTIONS I

```
//we define a function as parameter
fun filterEMails(emails : List<String>,
  predicate : (String) -> (Boolean)) : List<String>
  {...}
//and use it by providing a lambda
val listOfUsers = listOf("peter.muster@zhaw.ch",
  "paul.muster@google.com")

val filteredMails = filterEMails(users, {
 value -> value.endswith("@zhaw.ch")
})
```

HIGHER ORDER FUNCTIONS II

```
//when the last parameter in a function accepts a function, a
//lambda expression can be put outside the parantheses:
val filteredMails = filterEMails(users)
 {value -> value.endswith("@zhaw.ch")}
```

EXERCISE: LANGUAGE DETECTION I

Extend TextStore with a method "language" that tries to decide if the text was written in German or English. Use the letter frequency to determine the language by calculating the difference to the expected values.

```
val german = mapOf('e' to 17.40,'n' to 9.78, 'i' to 7.55,
's' to 7.27, 'r' to 7.00, 'a' to 6.51, 't' to 6.15, 'd' to 5.08,
'h' to 4.76, 'u' to 4.35, 'l' to 3.44, 'c' to 3.06, 'g' to 3.01,
'm' to 2.53, 'o' to 2.51, 'b' to 1.89, 'w' to 1.89, 'f' to 1.66,
'k' to 1.21, 'z' to 1.13, 'p' to 0.79, 'v' to 0.67, 'j' to 0.27,
'y' to 0.04, 'x' to 0.03, 'q' to 0.02)
val english = mapOf('a' to 8.167, 'b' to 1.492, 'c' to 2.782,
'd' to 4.253, 'e' to 12.702, 'f' to 2.228, 'g' to 2.015,
'h' to 6.094, 'i' to 6.966, 'j' to 0.153, 'k' to 0.772,
'l' to 4.025, 'm' to 2.406, 'n' to 6.749, 'o' to 7.507,
'p' to 1.929, 'q' to 0.095, 'r' to 5.987, 's' to 6.327,
't' to 9.056, 'u' to 2.758, 'v' to 0.978, 'w' to 2.360,
'x' to 0.150, 'y' to 1.974, 'z' to 0.074)
```

EXERCISE: LANGUAGE DETECTION II

Improve the letter detection with the following steps:

- Try to check for Umlauts
- Try to check for common words