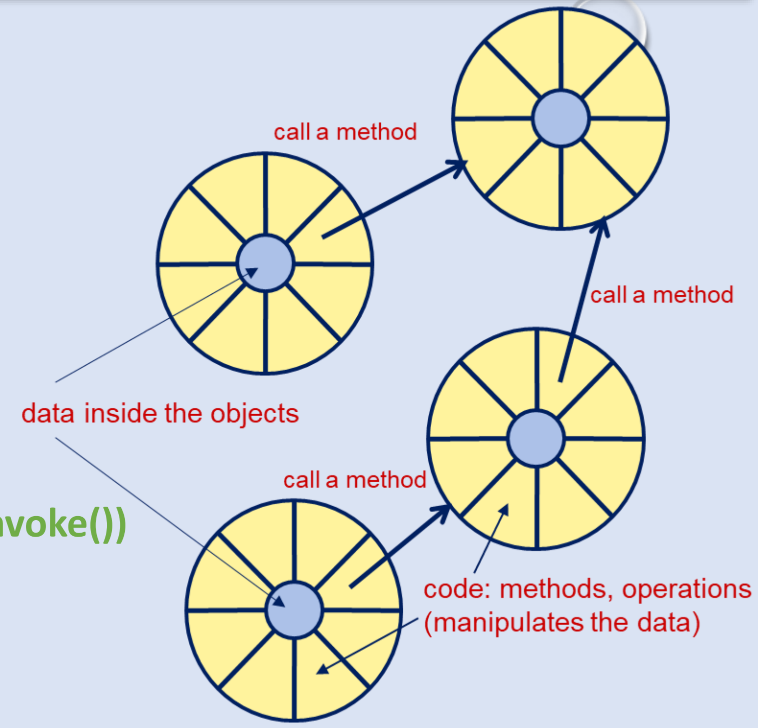
KOTLIN

An overview of the Kotlin programming language

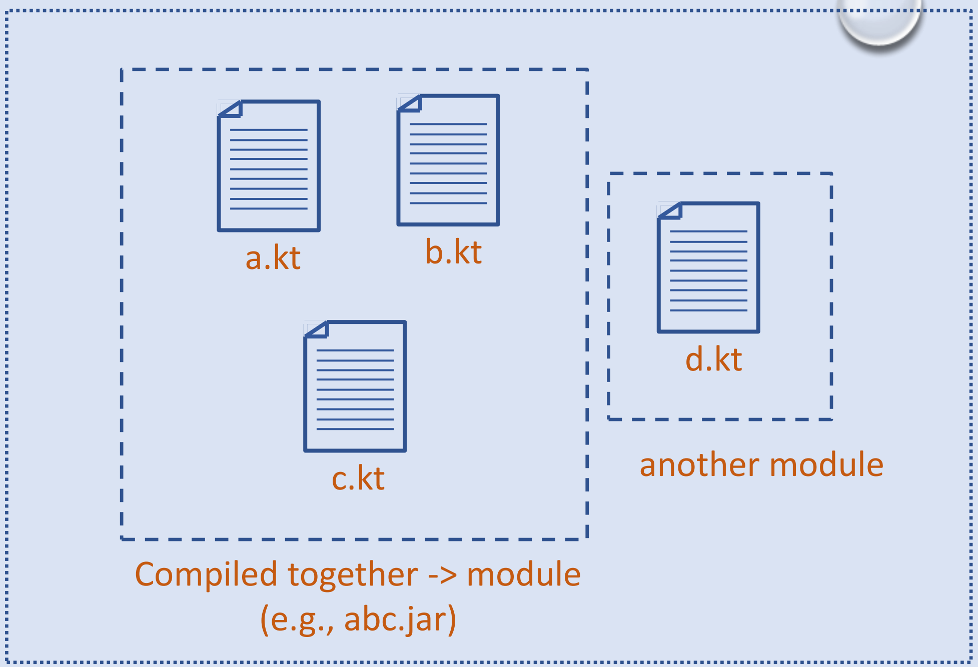
Kotlin language

- Kotlin is object-oriented with many functional features
  - A program contains several entities like
    - Properties – memory positions with a name, containing objects
    - Objects and their templates (classes)
      - Objects contain properties (representing the object <u>state</u>)
      - Objects contain also methods (functions representing the object <u>behavior</u>)
    - Functions – containing code, but also seen as objects with a type and methods (like invoke())
      - They are the methods of objects
  - Properties, Objects, and Functions can be top-level or inside Classes
  - All entities have a Type, which is the name of a class. It is strongly and statically-typed
  - A Kotlin program is made of declarations and definitions (Classes for objects, Properties, Functions, …)
  - Kotlin had the influence of many other programming languages: C/C++, Java, Scala, C#, …
  - Initially it was designed to produce code to the JVM (bytecode)
    - It can interoperate with Java compiled classes (calling them) and compiled Kotlin can be called by Java code
  - It can be very compact, but it has also many features that make it big and complex …



Kotlin source files

- Kotlin source files have the extension .kt
  - A program can be composed by several source files
    - A set of source files compiled together form a <u>module</u>
      - A program is composed by one or more modules
      - Must contain a top-level function **main()**



a program –> must contain a top-level function *main()*

- A Kotlin file is contained by the following
  - KotlinFile : [<package header>] (**package <id>** like Java)
    <import list>* (**import <id>**[**.***] | **as <id>**])
    <declaration>*
  - Where
    declaration: <class declaration>
    <property declaration>
    <function declaration>
    <object declaration> (**object <id> <class body>**)
    <type alias> (**typealias <id>**[**<type parameters>**] **= <type>**)

[…] optional
<…> some grammar rule
* zero or more times
| alternative (or)
text actual example or definition

Example programs

A simple 'Hello World'

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

Contains only a **function declaration** – the mandatory *main()* function (without parameters)

- *println()* is a **library function** defined in the automatically imported kotlin.io package
- There are a number of automatically imported standard libraries from Kotlin, and related to Java: java.lang.* and kotlin.jvm.*
- We can import any other of the packages defined in Kotlin and Java.

A more elaborated example

```
package org.feup.apm.example
import kotlin.math.*

class A() {
  var v = 2.0
  fun process(): Double {
    return sqrt(v)
  }
}

val another = 81.0

fun main() {
   val a = A()
   println("Value from A: ${a.process()}")
   a.v = another
   println("Another value from A: ${a.process()}")
}
```

Result:
Value from A: 1.4142135623730951
Another value from A: 9.0

- Instructions don't need to be terminated with **;** (they are optional)
- We can use **packages** to group declarations (like Java), but we don't need to put them on a similar directory structure (is mandatory in Java)
- We don't need to put just one top-level declaration on a single file (is mandatory in Java)
- **String literals** can contain property or expression values ($name or ${<expression>})
- The square root function *sqrt()* is already defined in the package kotlin.math
- **identifiers** can start with a letter or _ and contain more letters or digits (like Java)
- **convention** class names and types (which are classes) should **start** with a capital letter
  other names should use **camel case** (start with a small letter and if there are more words those are capitalized without _'s) example: **aCamelCaseName**

KOTLIN BASIC TYPES

Types and literals (values)

Basic types and literals

- **Kotlin basic types**
  - In Java we have primitive (directly a value) and reference (an address of a block of memory where values are) types
  - Kotlin does not make that differentiation, although the compiler can optimize the representation
- **Kotlin considers as basic types the following, which are also considered classes**
  - Numbers, Booleans, Characters, Strings, Arrays, the base type Any, and special return types (Unit and Nothing)
  - Also, the nullable extensions of these types (except Unit and Nothing)
- **Number types**

Type (class)	size (bits)	literal (value)	examples
Byte	8	[-]<digits>	123, -41
Short	16	[-]<digits>	1054, -1234
Int	32	[-]<digits>	54467, -23
Long	64	[-]<digits>L	569L, -1L
Float	32	[-]<digits>.<digits>F	2.456F
Double	64	[-]<digits>.<digits>	46.785
UByte	8	<digits>u	123u
UShort	16	<digits>u	1054u
UInt	32	<digits>u	54467u
ULong	64	<digits>uL	569uL

- **Integer** types can be expressed in **decimal**, **hexadecimal** (prefix 0x example 0xAB) or **binary** (prefix 0b example 0b00001111 (=15))
- Can be **signed** or **unsigned** (added in Kotlin 1.3)
- **Long** values should have the suffix L
- **Unsigned** values should have the suffix u
- **Floating point** types can be expressed with the standard dot notation (integer and fractional part) or the scientific notation (with an exponent)
  examples: 23.678 -4.8347 1e-8 34.67E3

Non number basic types (1)

- **Booleans**
  - Type: **Boolean** representing the logic values with literals **true** and **false**
- **Characters**
  - Type: **Char** represents a single character in Unicode (16 bit)
  - Is not treated as a number Literals between single quotes (‘) ‘a’ ‘+’ ‘K’ ‘\u1234’ ‘\t’ ‘\n’ ‘\r’ ‘\b’ ‘\’’ ‘\\’
- **Strings**
  - Type: **String** represents a sequence or array of characters (in Unicode) with a length of 0 or more
  - String literals are written between quotes (“)
    - The same escape sequences (with a \) as for characters can be used, plus the escape \$ for the dollar character
    - Properties (single name) and expressions can be embedded into a string using $<property name> or ${<expression>}
      - These are called templated strings: “The name is: $name” “The value of 4.5^2 is ${4.5*4.5}”
  - Verbatim String literals are written between triple quotes (“””) - in these escapes do not take effect
    - Example: “””This is a \verbatim string writing money like $100.00”””
  - The String class defines a lot of properties (like **length**) operators ([] and +) and methods to work with Strings

Non number basic types (2)

Arrays

- Arrays belong to the generic class **Array<T>** where **T** is the type of each element
- There is not a notation for representing literals of the Array type
  - But there is a top-level library function for building an Array from their elements: **arrayOf()**
    - Example: `val a = arrayOf(1, 2, 3)`
  - Also, the Array class has constructors for building an initialized Array
    - One of them allows to specify the size and function to calculate each element:

```
val perfectSquares = Array(10, { k -> k*k })
for (i in 0..9) print("{perfectSquares[i]} ")
```

Result: 0 1 4 9 16 25 36 49 64 81

  - Array elements are accessed with the usual **[ ]** syntax
  - Indexes start at **0**
  - Also, the Array class has many defined properties and methods
  - For the Boolean and numeric types, the Kotlin library also defines specific Array types
    - Like **IntArray**, **BooleanArray**, **UShortArray**, …
    - This allows the compiler to optimize the internal representation as arrays of values (not boxed)

The base type and nullable types

The Kotlin base type

- All types derive from a single type, considered the root or ancestor of all classes: **Any**
  - All other types and classes derive directly or indirectly from **Any**
  - It implements three base methods, that can (and should) be overridden in derived classes:
    - operator fun equals(other: Any?): Boolean (defines the operator == to compare two objects, true if they have the same value)
    - fun hashCode(): Int (returns an integer value unique with the object value)
    - fun toString(): String (returns the object representation as a string; by default, the class name, except for basic types)

Nullable types

- Any type can add the **null** value, declaring properties, parameters, and return types with the nullable value
  - The syntax is: <Type>?
- Nullable entities cannot be stored in non nullable from the same type, even if the value is not **null**
  - Is an error to assign a value from a nullable type to a property of the same non nullable type
  - var abc: Int? = 123
    val cde: Int = abc // this produces an error
- The **null** value <u>cannot</u> be stored on a <u>non nullable type</u>

Special return types

- A function that does not return any value (produces only side effects, aka procedure) can be declared as returning the type **Unit**
  - This type is used as Void in Java, and functions returning it can be declared with or without it
    - fun f(): Unit { … }
    - fun f() { … } is equivalent
  - As it is a class, it can be used as a generic type parameter; functions returning Unit don't need a return statement
    - The compiler adds a '**return Unit**' statement at the end of the function
    - **Unit** is also a predefined object, as a singleton of the Unit class, with no state inside it
    - Example:

```
interface Processor<T> {
    fun process(): T
}
class NoResultProcessor: Processor<Unit> {
    override fun process() {   …   }    // should return Unit, but we don't need to declare or use a return statement
}
```

- type **Nothing** is only used for functions that never terminate normally
  - Like: `fun fail(message: String): Nothing  { throw IllegalStateException(message) }`

KOTLIN PROPERTIES

Read-only, read-write, backing storage

Properties (aka variables)

- Kotlin state is stored in properties, at top-level, belonging to objects, and as local state in functions
  - Usually there is a memory backing store for property values, but not always
  - They can be seen as objects with two methods: **get()** and **set()**
    - By default, **get()** returns the property value, and **set()** assigns a value to the property
    - It’s possible to modify the default behavior of **get()** and **set()**
    - **get()** is invoked when the property is used in an expression, and **set()** when is used in the left side of an assignment (=)
  - There are two kinds of properties: read-only and read-write
    - <u>Read-only</u> have only the get() method and are declared as: **val <name>[: <Type>] = <value>**
      - Usually, they need to be initialized when declared, unless we redefine **get()** without using its own store
      - A **function local property** (with its type) can be initialized later
      - It can only be used after initialization
    - <u>Read-write</u> properties have the **get()** and **set()** methods and are declared as: **var <name>[: <Type>] = value**
    - Usually, the type is inferred from the value and needs not to be declared
      - It never changes, and is checked by the compiler on every use
    - Examples: **val myName: String = “John Doe”** **var myAge = 31** **val mySSN = “12345678910”**

Property storage (value or reference)
get() { … }
set() { … }

A property as an object

Redefining get() and set()

- The get() method of a property is seen as a function: fun get(): <property type>
- The set() method is implicitly declared as: fun set(val value: <property type): Unit
- Redefinition
  - Example:

```
var storeHalf: Int = 100
    get() { return field * 2 }
    set(value) { field = value / 2 }
println(storeHalf)    // prints 50
storeHalf = 1000      // stores 500 in the memory store
println(storeHalf)    // prints 1000
```

- field (keyword) refers to the backing storage of the property
- The parameter of set() (by convention should be named value) is the value to be assigned and does not need a Type (is the property type)
- The default get() and set() are equivalent to 'return field' and 'field = value' respectively
- If the actual implementations of get() and set() don't refer field the property does not have a backing storage in memory

KOTLIN FUNCTIONS

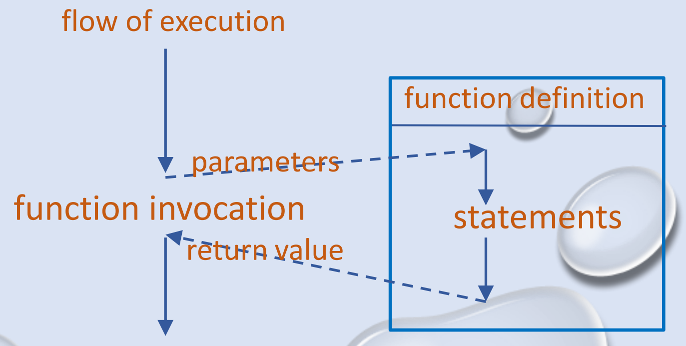
Functions, function types, function literals

Functions (1)

- Functions perform actions (aka execute code), have parameters, and a return type
  - A simplified syntax is:

```
fun <id>(<parameters>): <return type> {
      <declarations>
      <statements>
}
```

  - Functions can be top-level, inside classes (named as methods), or local to other functions (in **<declarations>**)
  - **<parameters>** are zero or more pairs of **<name>: <type>**, eventually with a default value (**= <value>**)
    - When invoking a function, the **function name** is followed by the **parameters values** between **(** and **)** in the **same order**
    - Parameters with **default values don’t need to be supplied** (parameters with default values are usually the last)
      - Supplied parameters can have their name with the value (**<name>=<value>**); these are called named parameters
      - For these, the **order doesn´t matter**, but **after one is named**, the **following must** also **be named**
  - A function invocation is an expression producing a value of the return type
    - The statements should calculate and **return that value** (unless the return type is **Unit**)
    - Example: **fun divide(divisor: Big, scale: Int = 0, round: RMode = RMode.NO): Big { … }**
    - Invoking: **divide(Big(12.65))**   **divide(Big(12.65), 4)**   **divide(Big(12.65), 4, RMode.HD)**
      divide(Big(12.65), round=Rmode.HD)



Functions (2)

- **Single expression functions**
  - Functions whose result is calculated with a single expression can be simplified:

    fun <id>(<parameters>) = <expression>

    - The return type is not needed to be declared, if it can be inferred from <expression>
    - The result is the value from the <expression> evaluation

```
fun square(k: Int) = k * k
fun concat(a: String, b: String) = a + b
```

- **Local functions can be declared inside other functions, and only used by the enclosing function statements**
  - Example:

```
fun printArea(width: Double, height: Double): Unit {
    fun calculateArea() = width * height
    val area = calculateArea()
    println("The area is $area")
}
```

- **Function types: functions can be seen as objects with a Type (for parameter or return type or for properties)**
  - (<list of parameters types> -> <return type>) Examples: (Int, Int) -> Int () -> Unit () -> () -> Int (right associative)
  - Technically they are interfaces that define an invoke() function, so we can define classes that implement them

Functions (3)

➢ Function literals

- We can define a property with a function type, and also parameters and the return type of another function
  - The initialization or parameter of a property or function that is a function, can be a function literal
  - A function literal can be defined as a lambda: [<label>] { (<parameters>) -> <statements> }
    - The return value is the result of the last <statement> in a lambda
    - To return earlier we need a <label> at the beginning of the function literal

```
val getMessage = lambda@ { response: Response -> if (response.code !in 200..299)
                                                      return@lambda "Error"
                                                  return response.message }
```

    - Examples:

```
val square: (Int) -> Int = { x: Int -> x * x }
val printer: () -> () -> Unit = { { println("I am printing") } }
```

      - If there no arguments on a function type the literal omits () ->
      - If there is only one: omit the parenthesis or omit the left-hand side and refer to the parameter with the keyword it

```
val printInt: (Int) -> Unit = { print($it) }
```

- Invocation of something that is a function is done using the usual (<parameter values>)
- Another way to define a function literal is using an anonymous function: the fun(…): … { … }, without a name
- A reference to a named function can also be used in the place of a function, as: ::<function name>

Functions (4)

- **When last parameter is a function**
  - It can be passed (usually as a function literal) detached from the previous parameters
    - Example: fun abc(a: String, b: () -> Int) { … } Invocation: abc(“John”) { return 41 } (= abc(“John”) { 41 })
    - When the last statement of a function returns the value of an expression, the return keyword is optional
- **Function with variable number of arguments: use keyword varargs**
  - Example: fun multiprint(prefix: String, varargs strings: String) { println(prefix) for (str in strings) println(str) }
    - Invocation must have separated arguments: multiprint(“Start”, “a”, “b”, “c”)
  - if the varargs is not the last parameter, those must be named
    - fun multiprint(prefix: String, varargs strs: String, suffix: String) { println(prefix) for (str in strs) println(str) println(suffix) }
      Invocation: multiprint(“Start”, “a”, “b”, “c”, suffix = “End”)
  - An Array<T> can be separated in its elements using the spread operator (prefix *)
    For example, this array: val strs = arrayOf(“a”, “b”, “c”) can be passed as a varargs parameter with *strs
    Thus, the invocation of the first *multiprint()* function can be:
    multiprint(“Start”, *strs)

Tail recursive functions

- **Tail recursive functions – last operation of a function is the recursive call, and it returns this call result**
  - This functions can be very efficient (no need to keep stack frames)

    the compiler can optimize the code marking those functions with **tailrec** keyword
  - Example: Making a function tail recursive usually needs some refactoring of the function code
  - Start with the classic factorial calculation in a recursive function

```
fun factorial(k: Int): Int {
    if (k == 0) return 1
    return k * factorial(k – 1)
}
```

  - This is not a tail recursive function because it returns the result of the recursive call multiplied by k
  - But here the local function *factTail()* is (*factorial()* is not recursive anymore, it just calls a recursive one):

```
fun factorial(k: Int): Int {
    tailrec fun factTail(m: Int, n: Int): Int {
        if (m == 0) return n
        return factTail(m-1, m*n)
    }
    return factTail(k, 1)
}
```

KOTLIN STATEMENTS

Statements, Expressions, Operators

Statements / assignment

- Statements perform some action and generate low-level instructions
  - Included are assignments, control-flow, expressions, exceptions
- Assignments modify program state and are performed with one of the assignment operators
  - <assignable expression> <assignment operator> <expression>
    - The assignable expression is a property, a member of an object, an element of an array, or a suffixed result of a call; so, anything that can receive a value
      - abc, abc.cde, abc[k], abc().cde, abc
    - The assignment operator is =, but also +=, -=, *=, /=, %=, which also perform the operation (a += 20 ➔ a = a + 20)
    - An expression calculates a values from a large set of operands and operators (unary or binary)
    - An assignment statement does not represent a value in Kotlin, so you cannot write a = b = 10 ✘
- Expressions compute a value of some Type resulting usually from the application of an operator to operands
  - The value of an operand can come from literals, or other constructs that represent a value
  - Operators are defined for precise operand Types, have priorities and associativity, and a syntactic rule for use
  - Many operators can be defined in classes through operator functions

Operators

➢ Most Kotlin operators and their properties are shown on the following table

Kotlin Operators		
Precedence	**Name**	**Operator**
Highest precedence	postfix	++ -- . ?. ? [] ()
	prefix	- + ++ -- !
	type	: as as?
	multiplicative	* / %
	additive	+ -
	range	..
	infix function (bitwise)	shl shr ushr and or xor inv
	Elvis	?:
	checks	in !in is !is
	comparison	< <= > >=
	equality	== != === !==
	conjunction	&&
	disjunction	||
	spread	*
Lowest precedence	assignment	= += -= *= /= %=

All operators are usually left associative,
except the prefix operators,
and the lambda function type indicator (**->**)

Types and casts

- **Operators require well defined operand types**
  - Kotlin usually does not automatically convert between types (some exceptions with numeric literals)
    - For numeric types, there are conversion functions defined on the type classes (e.g., the **Float** type has the **.toInt()** method)
    - Objects can be cast to super (ancestor) or sub (descendent) classes
    - Nullable classes need special care
  - Casts
    - Are performed with the **as** operator

      ```
      fun casting(any: Any) { val num = any as Int }        // OK if the passed parameter was an Int
      ```

      - When a cast fails an **Exception** is thrown
      - We can test before for a successful cast with the **is** or **!is** operators: `if (any is Int)  num = any     // in this case the cast is not needed`
  - Safe casts
    - To avoid the Exception, we can use the safe cast operator: **as?**

      ```
      fun safeCasting(any: Any) { val num: Int? = any as? Int }
      ```

      - If the cast fails **null** is the result
  - Elvis operator: Allows to test is something is null, if not evaluates to the value, else evaluates to something specified
    - Example: `fun safeCasting(any: Any)  { val num: Int  =  any as? Int  ?:  0 }`

Null handling

- Trying to use a **null** to access something or assign it to a non-nullable receiver generates an **Exception**
  - We can test against **null**

```
fun nullCheck(str: String?) {
    val upperCase: String? = if (str != null)
                               str.toUpperCase()
                             else
                               null
}
```

  - Or we can use the safe call (or safe access) operator, which is **?.**

```
fun safeCall(str: String?) {
    val upperCase: String? = str?.toUpperCase()       // evaluates to null if str is null
}
```

  - It's possible to chain safe calls if those can also return **null**

```
val firstLetterCapitalized: String?  =  str?.firstOrNull()?.titlecase()
```

  - Using the non-null assertion (**!!**) where the programmer vouches for a non-null value (converts to non-nullable without testing)

```
fun nonNullAssertion(str: String?) { val upper: String = str!!.toUpperCase() }
```

Control flow statements

- **Contain the If, When, Loops, and Return statements**
  - **If instruction is similar to any other language**
    - Evaluate a Boolean expression (condition) and if true execute the embedded statement or statements (between { and })
    - Can have an optional else with a one statement or statements (between { and })
    - An if statement can be used as an expression, having as result the last expressions of the then or else parts
      - In this case the else part must be present
    - Example:

```
val x = if (a <= 10)
            2 * a
        else
            a / 2
```

      - In this way it is equivalent to the C or Java ternary operator (Boolean ? Value1 : Value2)
  - **When is like the switch statement of C or Java but more flexible and powerful (it can be used also as expression)**

```
when (num) {
  1 -> println(“1”)
  2, 3, 4, 5 -> println(“Range 2 to 5”)
}
```

Normal (similar to switch)

```
val s = when (num) {
  1 -> “Number is 1”
  2, 3, 4, 5 -> “Number in range 2 to 5”
  else -> “Number higher than 5”
}
```

Expression (exaustive)

```
when (any) {
  is Int -> print(“Int”)
  is Double -> print(“Double”)
  is String -> print(“String”)
}
```

Any object and conditions

```
when {
  a*b > 100 -> print(“* >”)
  a+b > 100 -> print(“+ >”)
  a<b -> print(“<”)
}
```

Without a selector

Loops

- Kotlin has the While and Do-while loops that are identical with the ones of C and Java
- Also, the **break** and **continue** statements, inside loops, have similar behavior
- The For loop is different (identical to the foreach in C#)
  - The for loop iterates a property in a collection, like a **Range** (elements here must be comparable)
    - Range is a type, with literals built with the **..** operator, and functions like **downTo()**, **step()**, **reversed()**, **rangeTo()**, and **until()**
    - It contains in sequence a start value, an end value, and all values in between with a certain increment or decrement
      - Usually built from integer types, characters, Booleans, or enumerations
      - Examples: 1..10 ‘a’..’h’ 100.downTo(0) 10.rangeTo(20) (1..50).step(2) (2..100).step(2).reversed() 0.until(10)
      - The functions **downTo**, **rangeTo**, **step** and **until** can also be used in infix call format: 2 until 5 4..20 step 2
    - A value can be tested if it is contained in collection (or not) with the containment operators: in or !in
  - Examples:

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

```
val a = arrayOf(4, 10, 25, 50)
for (item in a) {
  println(item)
}
```

```
val set = setOf(1, 15, 25, 30)
for (element in set) {
  println(element)
}
```

```
val array = arrayOf(4, 10, 25, 50)
for (k in array.indices) {
  println(“value at position $k: ${array[k]})
}
```

Return and Exceptions

- Return statement terminates a function with a possible result which is the expression after it
  - We don't need a **return** for functions returning **Unit** – it is automatically inserted at the end of the statements
  - **Return** applies only to the scope where it executes, unless it is a closure lambda literal (applies to the outer scope)

```
fun largestOfThree(a: Int, b:Int, c: Int): Int {
   fun largest(a: Int, b: Int): Int {
      if (a > b) return a
      else return b
   }
   return largest(largest(a, b), largest(b, c))
}
```

```
fun printWithoutStop() {
   val list = listOf("a", "b", "stop", "c")
   list.forEach {
      if (it == "stop") return
      else println(it)
   }
}
```

Prints a and b only

or the implicit forEach@

```
fun printWithoutStop() {
   val list = listOf("a", "b", "stop", "c")
   list.forEach stop@ {
      if (it == "stop") return@stop
      else println(it)
   }
}
```

Prints a , b, and c only

- Exceptions – any fault situation throws an Exception
  - Exceptions are handled like Java, with **try**, **catch** and **finally** (**catch** and **finally** are optional, but one must be present)
    - Unlike Java there are no checked Exceptions (must be declared in functions signatures)
    - There are many Exception classes ready to be used
    - They are generated with the **throw <exception>** statement in some function
    - Unhandled exceptions terminate the program

KOTLIN CLASSES

Object-oriented features

Class general format

- In Kotlin, as other OO languages, classes act as templates to build objects (which are class instances)
- The general format of a class declaration and definition comprises:

```
class <name>[<primary_constructor>][<derivations>] [ {
        [<init_blocks>]
        [<secondary_constructors>]
        [<companion_objects>]
        <declaration>*                    // mostly properties, functions, and other classes
 } ]
```

- Classes obey the four main principles of OO
  - Encapsulation, Abstraction, Inheritance, and Polymorphism
  - Classes have names, by convention starting with a capital letter, and using Camel Case
  - There are also specialized constructs for special kinds of classes, like interfaces, and enumerations
  - A class in Kotlin can inherit all the functionality of only another class, but implement several interfaces (like Java)
  - Unlike Java, and other languages, by default classes cannot be sub-classed (inherited from, aka final), but require a prefix modifier to allow that (the open keyword)

Class and modifiers

- The default behavior for a class declaration is
  - Usable by any other construct on the program (in any file), and not inheritable
  - Those behaviors can be changed by optional modifiers (keywords in front of the class keyword)
  - For class visibility we have the visibility modifiers
    - public (the default) – visible anywhere
    - internal – visible only on the same module
    - private – visible only on the same file
  - These visibility modifiers can also be applied to class members with similar meaning, except for
    - private – only accessible from within the same class
    - protected – only accessible from the same class or derived classes
  - Inheritance is controlled by inheritance modifiers, which are
    - final (the default) – the class is not inheritable; for a member it means it is not overridable (this is the opposite of Java)
    - open – the class is inheritable, and a member is overridable (it should be explicitly added if we desire inheritance)
    - sealed – a memberless class that can only be extended by sub-classes defined as nested classes
  - For nested classes we can add the modifier inner – adds a reference to the Outer class (this@Outer)

Constructors

- When an object is created from a class (the template), a constructor should be executed
  - Classes can declare a simple primary constructor that only initializes the main properties of the class
    - The **primary constructor** (if present) appear immediately after the **class name** between **(** and **)**, and a **list of properties**
    - When an object is created, the values for properties must be supplied
    - We can define default values for the constructor properties

```
class User(val name: String, var age: Int = 20)
val u = User("John", 41)            // the User class has two properties
                                    // initialized when an object is created
```

    - Kotlin classes support additional constructors, as functions, inside the class body (**{ … }**), and with the **constructor** keyword
      - These are called **secondary constructors**
      - They must differ in the parameters, and contain code (the function body)
      - A class without any constructor, will receive a parameter less, do nothing, constructor (the default constructor)
      - A secondary constructor must call the primary, if present (using **: this(…)**)
      - Class properties must be initialized if there is a primary constructor, otherwise they can be initialized in a secondary constructor

```
class User(val name: String) {
  var address: String = ""

  constructor(name: String, address: String)
                                : this(name) {
    this.address = address
  }
```

Init blocks in a class

- In the body of a class, we can declare one or more init blocks: init { … }
  - The init blocks execute their instructions during object creation, after the primary constructor
    - If there is more than one, they execute in sequence, by the definition order (together with other class properties initialization)

```
class Person(val name: String, val age: Int) {
   var isOlderThanMe = false
   val myAge = 25

   init {
      isOlderThanMe = age > myAge
   }
}
```

    - They have access to the class members, can modify them, and call methods

- Classes with only secondary constructors should not have init blocks

```
class User {
  var name: String
  var address: String
  val all
    get() = "$name, $address"

  constructor(name: String, address: String) {
    this.name = name
    this.address = address
  }

  constructor(name: String): this(name, "none") {
  }
}
```

Inheritance / derivation

- A class definition can be made to use all the functionality of another class and add some new functionality
  - The class from where we derive the new one is called the super-class or the base class
  - The derived class is called a sub-class or child class
  - Kotlin supports only single derivation, but multiple inheritance from interfaces (special classes with restrictions)
  - If a new class declaration does not specify a base class, implicitly is Any
- Syntax for derivation
  - It should include a call to a constructor of the base class with : super(<parameters>) appended to a constructor
  - The use of any constructor of the sub-class must include that call in its way
    - If the sub-class has a primary constructor, it must include the call to the super-class constructor immediately

```
open class Base(val p: Int)
class Derived(val p: Int) : super(p)
```

    - If the sub-class defines secondary constructors, all should conduct also to such a call

```
class MyView : View {
  constructor(ctx: Context) : super(ctx)
  constructor(ctx: Context, attrs: AttributeSet) : this(ctx) {
     // do something with attrs
}
```

Overriding and polymorphism

- When a class derives from other, it can replace some members, with same name and type
  - For that to be possible the member in the super-class must have the modifier **open** (default is **final**)
  - On the sub-class, that member (same name and type) must have the prefix override
  - These members implement polymorphism
    - That means that a parameter or property with a super-class type, uses the actual members, when assigned with a sub-class object (without a cast)

```
open class A {
   open fun identify() { println("I'm A") }
}

class B: A() {
   override fun identify() { println("I'm B") }
}
```

method overriding

```
fun some(a: A) {
   a.identify()
}
```

calling some() with a B object
will use the B identify()

```
fun main() {
   val a1: A = B()
   val a2: A = A()
   some(a1)
   some(a2)
}
```

This prints:
I'm B
I'm A
(polymorphism in action)

To access a member on the parent class we use the qualifier **super**

```
class B: A() {
   override fun identify() { println("I'm B") }
   fun identifyParent() { super.identify() }
}
```

On an open class overridden methods are also **open** by default.
To prevent further replacement on children,
we can make them **final**

Interfaces

- Interfaces are classes with some restrictions
  - They do not have constructors
  - They can have properties and methods, are open by default, and can inherit from other interfaces
  - Members can be abstract (only declaration, but no implementation; for properties this means no accessors and no backing storage)
  - Or they can have an implementation (called the default implementation, properties cannot have backing storage)
- Other classes can inherit from any number of interfaces
  - Must override the interface abstract members

```
interface Document {
   val version: String
   var size: Long
   val name: String
      get() = “No name”
   fun save(stream: InputStream)
   fun load(stream: OutputStream)
   fun description(): String {
      return “Document $name has $size bytes”
   }
}
```

```
class MyDoc: Document {
   override var size = 0
   override val version = “1.0”
   override fun save(stream: InputStream) { … }
   override fun load(stream: OutputStream) { … }
}
```

```
fun main() {
   val doc = MyDoc()
   println(doc.description())
}
```

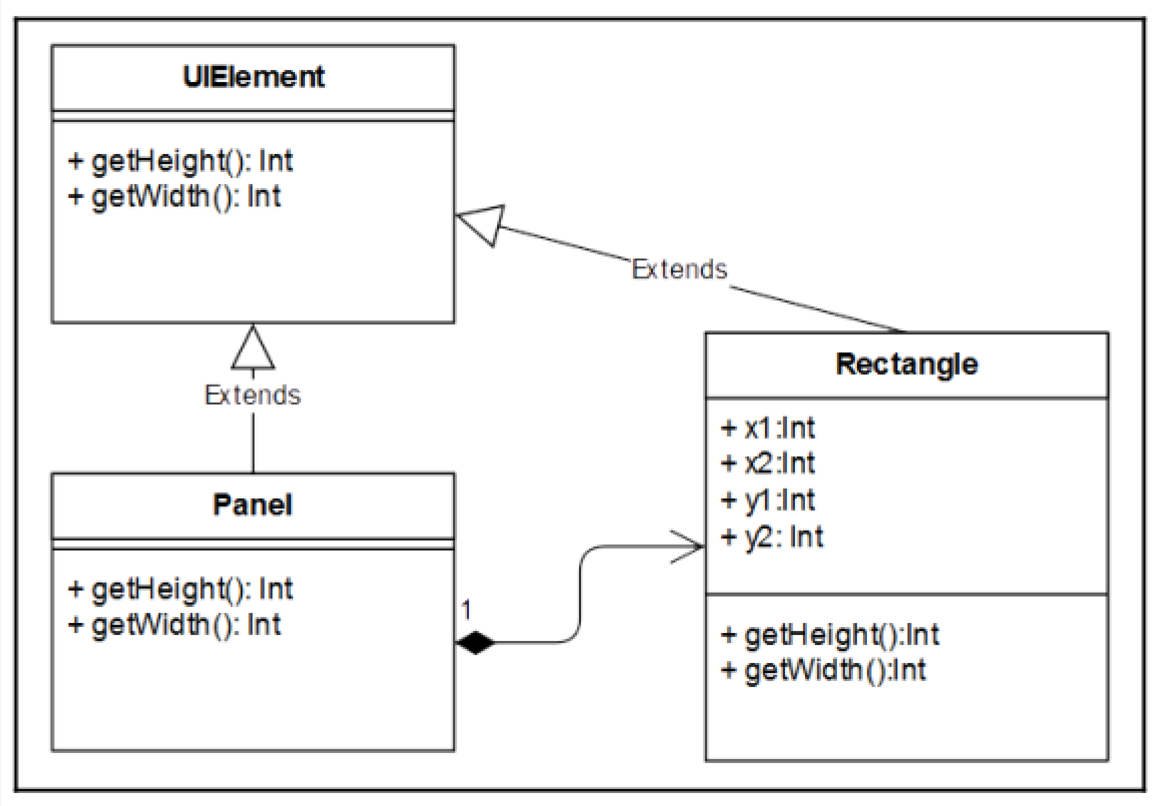
This will print:
Document No name has 0 bytes

Interfaces or abstract classes

- Kotlin supports abstract classes (with the abstract keyword) the same way as Java
  - Which are partially implemented, with not implemented properties and methods also marked with abstract
  - Derived classes must implement them, or be marked with abstract also
- Interfaces and abstract classes are very similar
  - Abstract classes are full-fledged classes
  - Derivation from an abstract class should be the way, if the better description of a child is “is a …”
  - Interfaces should be used if the better description of the child is “can do …”
  - Abstract classes promote “code reuse”, and also “versioning”
    - We can add a method to an abstract parent and all the children inherit it
  - Interfaces define a “contract” and all classes implementing it can be used in that way (even if they do very different things)

Class delegation

- In class hierarchies, derived classes sometimes have common code
  - Kotlin allows its reuse by **delegation** (use implementations of a “sister” included property) using the **by** keyword




```
interface UIElement {
  fun getHeight(): Int
  fun getWidth(): Int
}

class Rectangle(val x1: Int, val x2: Int, val y1: Int, val y2: Int): UIElemnt {
  override fun getHeight() = y2-y1
  override fun getWith() = x2-x1
}

class Panel(val rect: Rectangle): UIElement by rect

fun main() {
  val panel = Panel(Rectangle(10, 100, 30, 100))
  println(“W: ${panel.getWidth()}”)
  println(“H: ${panel.getHeight()})
}
```

Prints:
W: 90
H: 70

Extension functions

- Sometimes we have Classes (e.g., from a library) that we would like to had some methods
  - One way to do that is by deriving a child class, and add the methods
  - But what if the class is not open? Well, in Kotlin we can **extend it** with any functions indicating the **receiver type**
    - Suppose we have a List type for a collection of any objects, and it misses a drop() method, that produces a new List without the first k elements
    - We can add it

```
fun List.drop(k: Int): List {
  val resultSize: size – k
  when {
    resultSize <= 0 -> return emptyList()
    else -> {
      val list = ArrayList(resultSize)
      for (index in k..size-1) {
         list.add(this[index])
      }
      return list
    }
  }
}
```

We can use it as any other method of List

```
val list = listOf(1, 2, 3)
val dropped = list.drop(1)
```

Or even together with other functions of List (could be extensions)

```
val r = list.take(2).reverse().drop(1)
```

Which is much more readable than if the functions were defined top-level with a List parameter

```
var r = drop(reverse(take(list, 2)), 1)
```