UNIT 6

INTRODUCTION TO KOTLIN

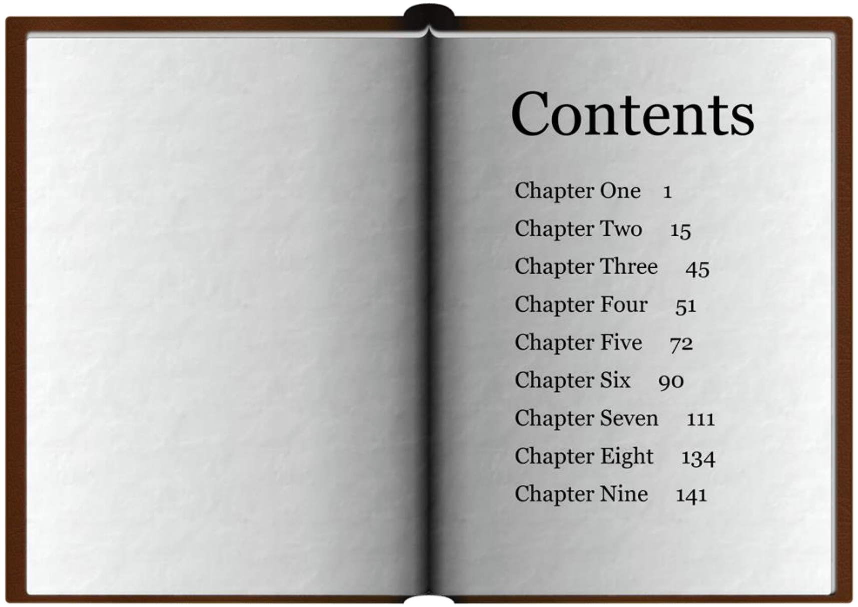
PMDM - 2DAM

Àngel Olmos (a.olmosginer@edu.gva.es)

Jose Pascual Rocher (jp.rochercamps@edu.gva.es)

Content

1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE



Contents

INTRODUCTION

- Developed by JetBrains (Russia) in 2011
- Popular programming language especially in **Android development**
- Designed to be **interoperable with Java** → one can use it in projects that are already written in Java and vice versa

Key features

Security (null safety)	**Conciseness** (fewer lines of code)
Interoperability (Kotlin <-> Java)	**Functional programming** (higher-order functions, lambdas ...)
Object-oriented	Cross-platform support

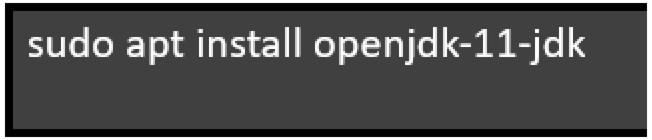
INTRODUCTION

Installation

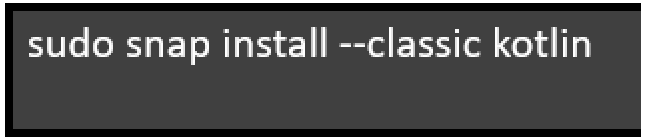
- Install the desired version of java
- Install Kotlin
- Create a “Hello World” file *hello.kt*
- Compile and Run
- One can also start some Kotlin coding and testing using a Kotlin playground

https://play.kotlinlang.org

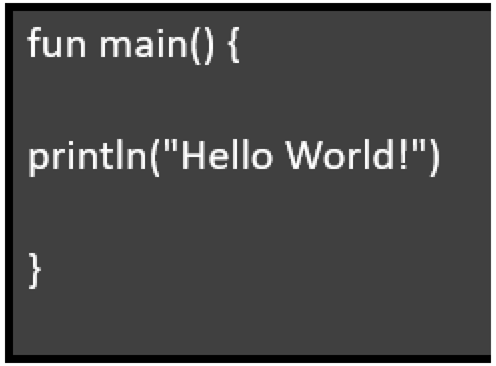
https://developer.android.com/training/kotlinplayground



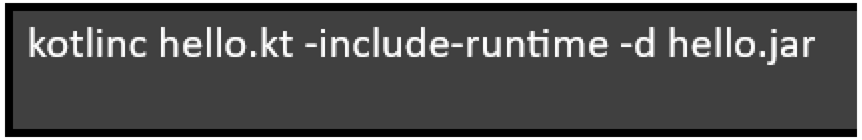
```
sudo apt install openjdk-11-jdk
```



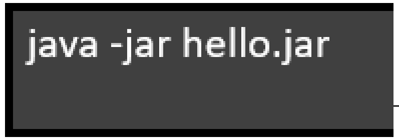
```
sudo snap install --classic kotlin
```



```
fun main() {

println("Hello World!")

}
```



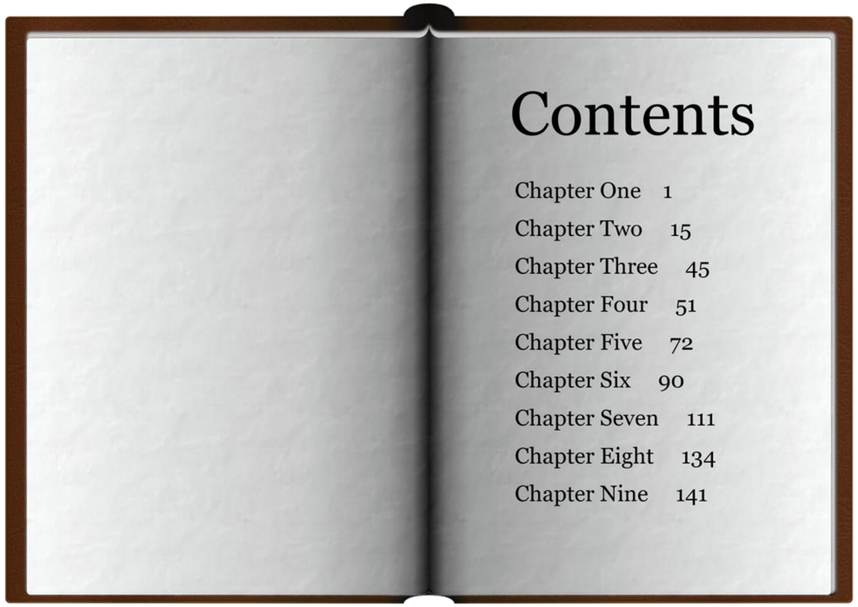
```
kotlinc hello.kt -include-runtime -d hello.jar
```



```
java -jar hello.jar
```

Content

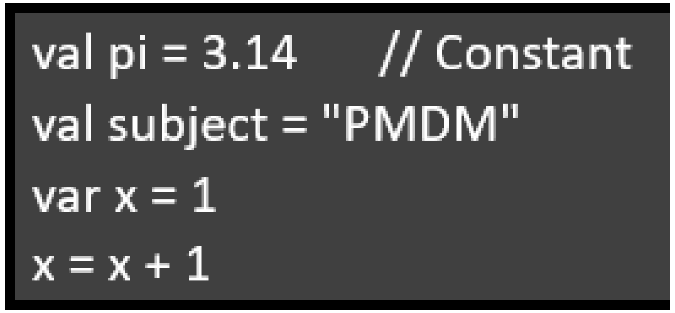
1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE



Contents

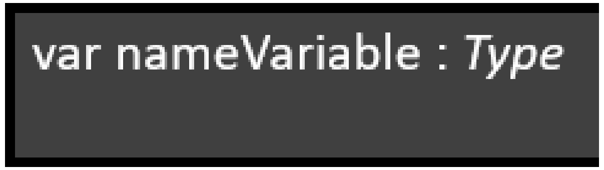
VARIABLES AND DATA TYPES

- In Kotlin data types are classes, so we can access their properties and member functions
- Use *var* to define mutable variables
- Use *val* to define immutable variables (constant values)
- It is recommended to use constant values if we know that they won’t be modified



```
val pi = 3.14      // Constant
val subject = "PMDM"
var x = 1
x = x + 1
```

Kotlin can infer the type of variables from the values with which we initialize them

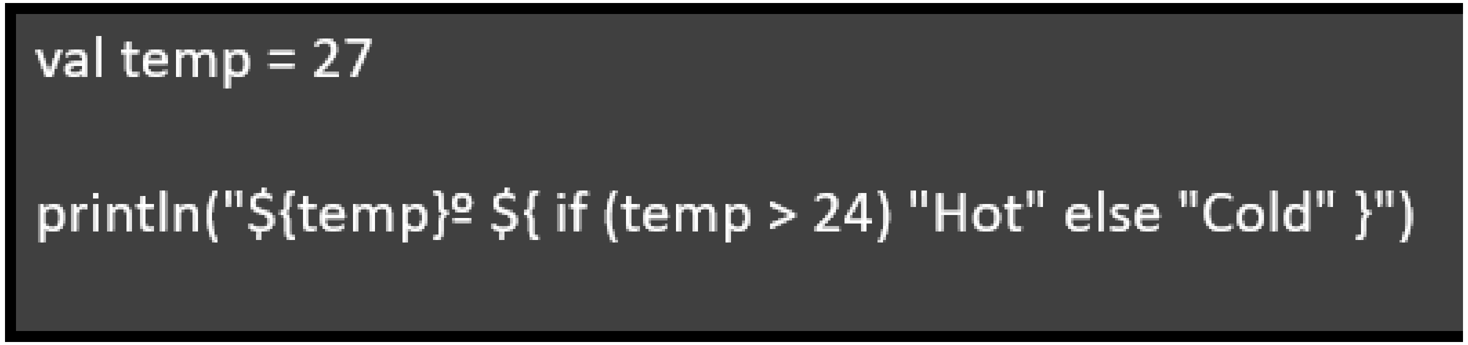


```
var nameVariable : Type
```

VARIABLES AND DATA TYPES

String Templates

- Fragments of code that will be evaluated and their result concatenated into the string
- Begin with the dollar sign *$* and consist of a **variable name or** an **expression** between keys *{}*

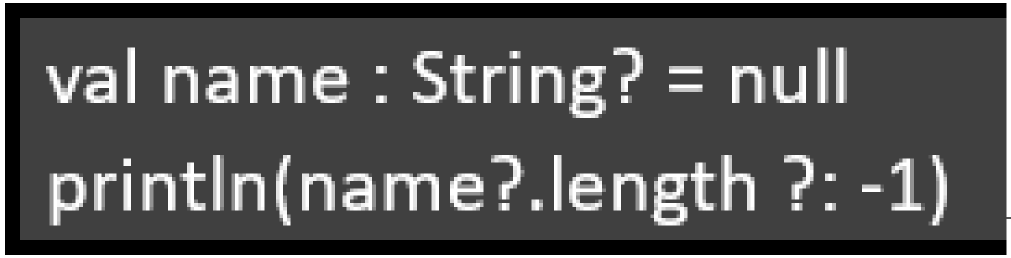


```
val temp = 27

println("${temp}º ${ if (temp > 24) "Hot" else "Cold" }")
```

VARIABLES AND DATA TYPES

Nullable Types and Elvis Operator

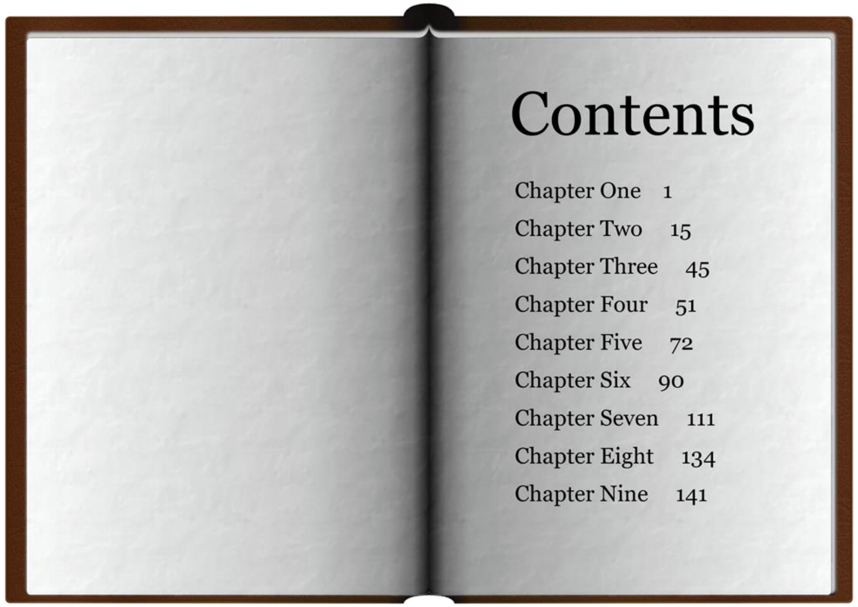
- Kotlin does not allow variable values to be *null* by default
- Prevents us from programming errors such as *NullPointerException*
- If we want to specify that a variable can contain a *null* value, it is necessary to **explicitly define it as *nullable***
- Kotlin provides the "**?:**" operator (Elvis), to specify an alternative value when the variable is *null*



```
val name : String? = null
println(name?.length ?: -1)
```

Content

1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE



Contents

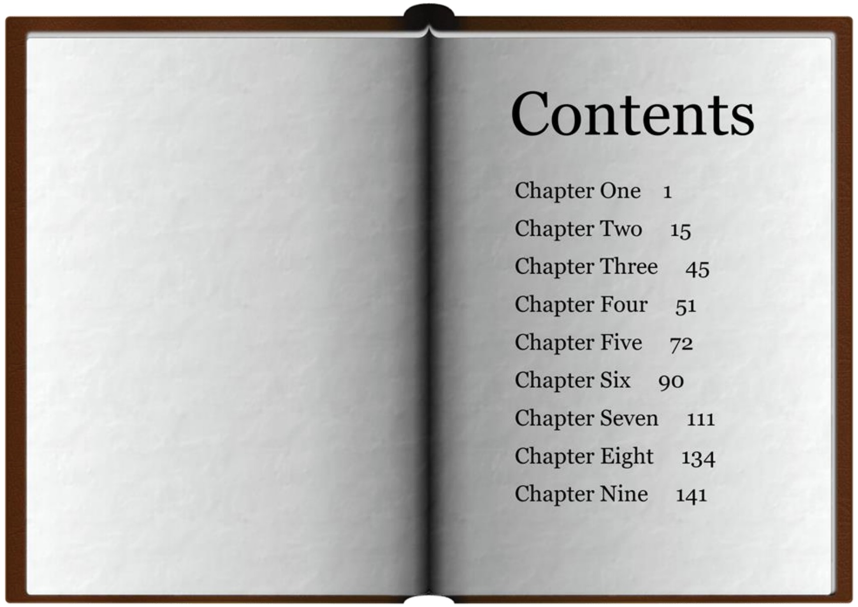
Chapter One 1
Chapter Two 15
Chapter Three 45
Chapter Four 51
Chapter Five 72
Chapter Six 90
Chapter Seven 111
Chapter Eight 134
Chapter Nine 141

OPERATORS

ARITHMETIC OPERATORS	RELATIONAL OPERATORS	LOGICAL OPERATORS	TERNARY CONDITIONAL OPERATOR
+ - % * / ++ - -	== != <, >, <=, >= === same object !== not same object	! Negation || OR && AND	variable = condition ? expression_1 : expression_2

Content

1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE



Contents

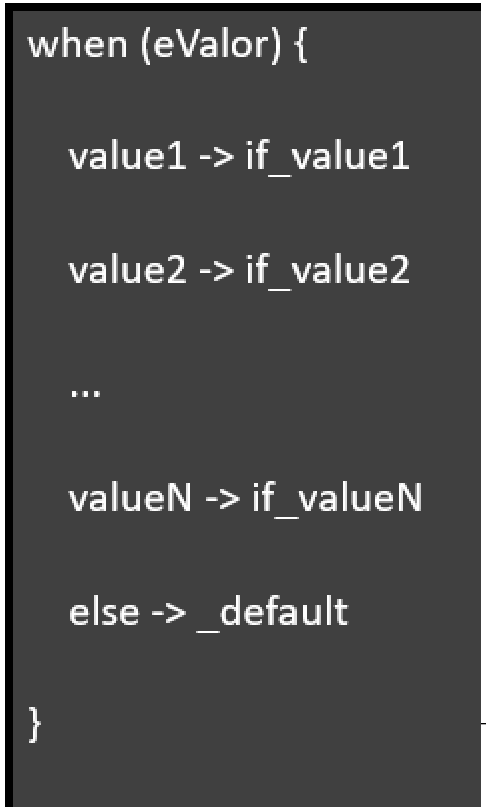
Chapter One 1
Chapter Two 15
Chapter Three 45
Chapter Four 51
Chapter Five 72
Chapter Six 90
Chapter Seven 111
Chapter Eight 134
Chapter Nine 141

CONTROL STRUCTURE

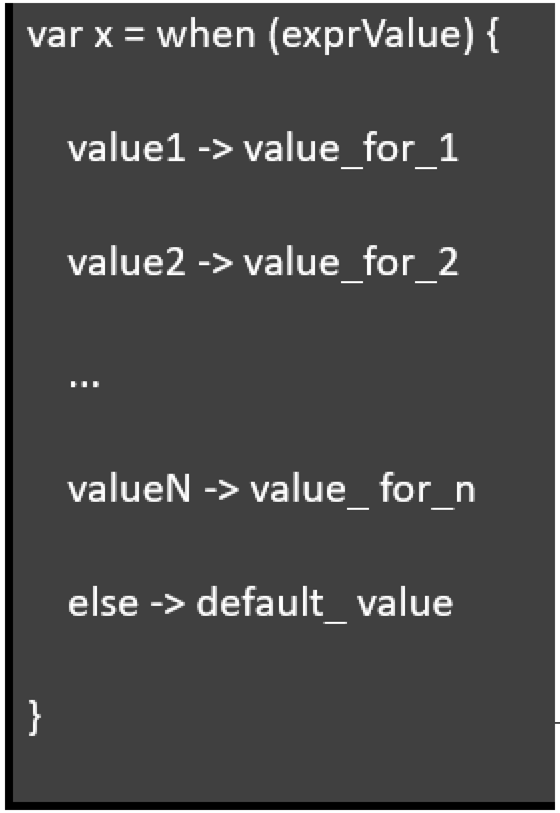
If - else → Like in Java

When

It can be expressed as a statement or as an expression



```
when (eValor) {

   value1 -> if_value1

   value2 -> if_value2

   ...

   valueN -> if_valueN

   else -> _default

}
```

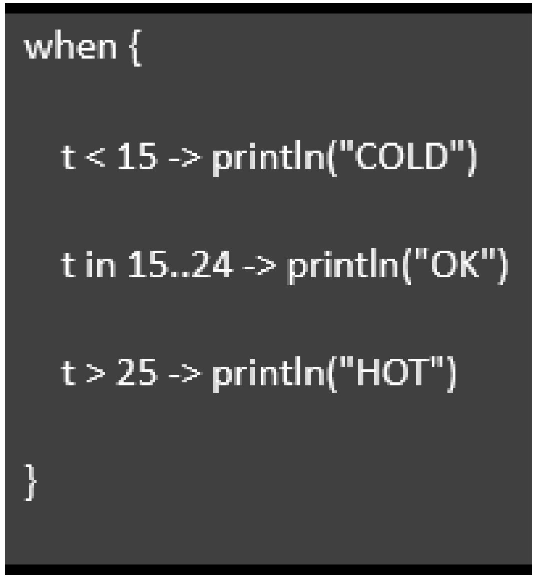


```
var x = when (exprValue) {

   value1 -> value_for_1

   value2 -> value_for_2

   ...

   valueN -> value_ for_n

   else -> default_ value

}
```

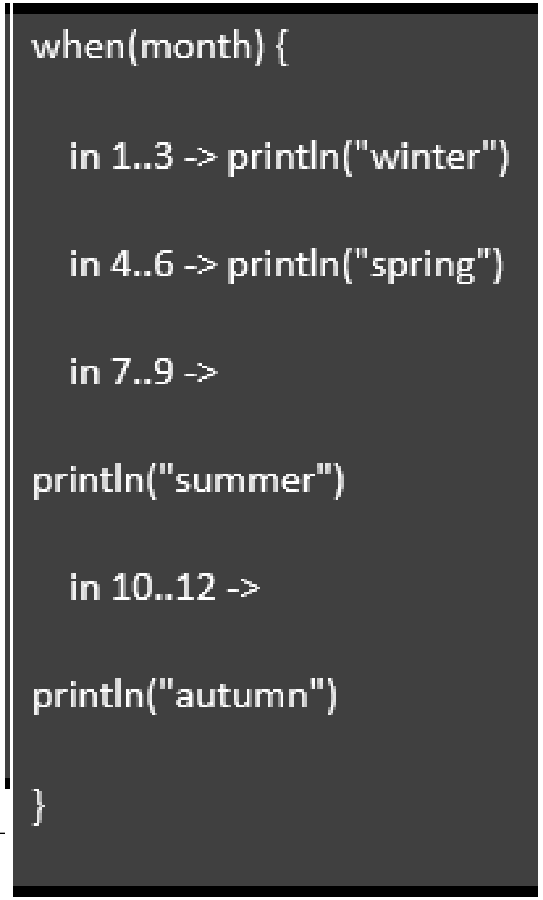
CONTROL STRUCTURE

When

Also use it without arguments (as if-then-else) and with the “is” and “in” operators



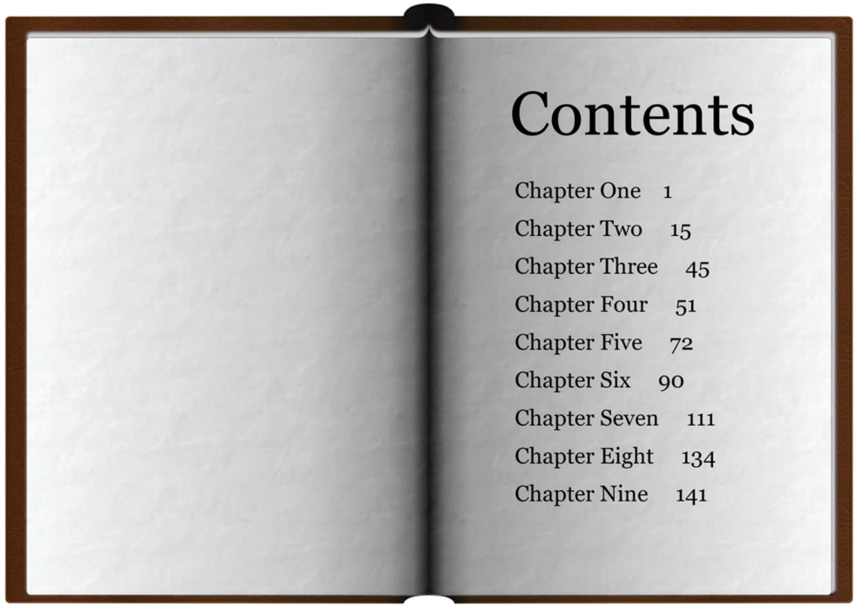
```
when {
   t < 15 -> println("COLD")
   t in 15..24 -> println("OK")
   t > 25 -> println("HOT")
}
```



```
when(month) {
   in 1..3 -> println("winter")
   in 4..6 -> println("spring")
   in 7..9 ->
println("summer")
   in 10..12 ->
println("autumn")
}
```

Content

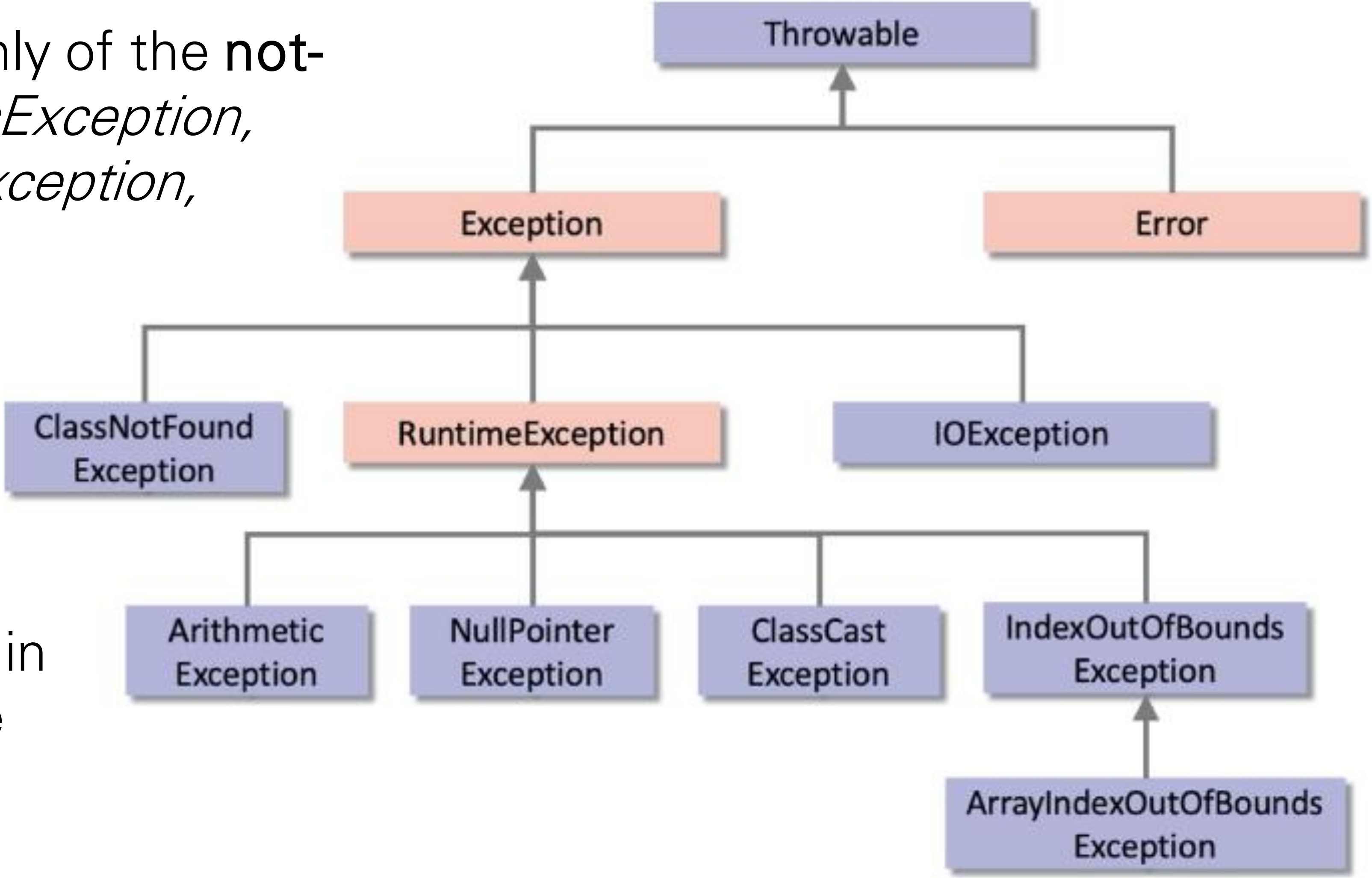
1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE



Contents

EXCEPTIONS

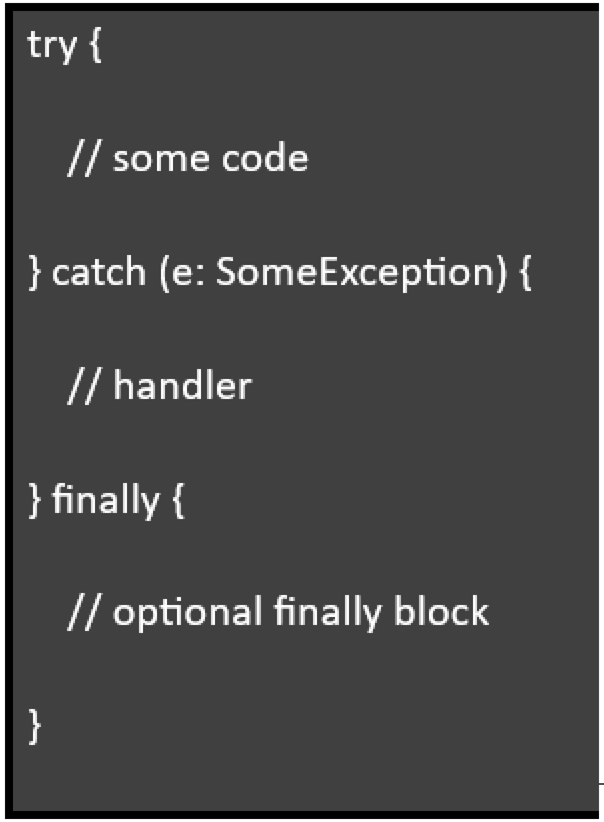
Exceptions in Kotlin are only of the **not-reviewed** type: *ArithmeticException, ArrayIndexOutOfBoundException, NullPointerException...*



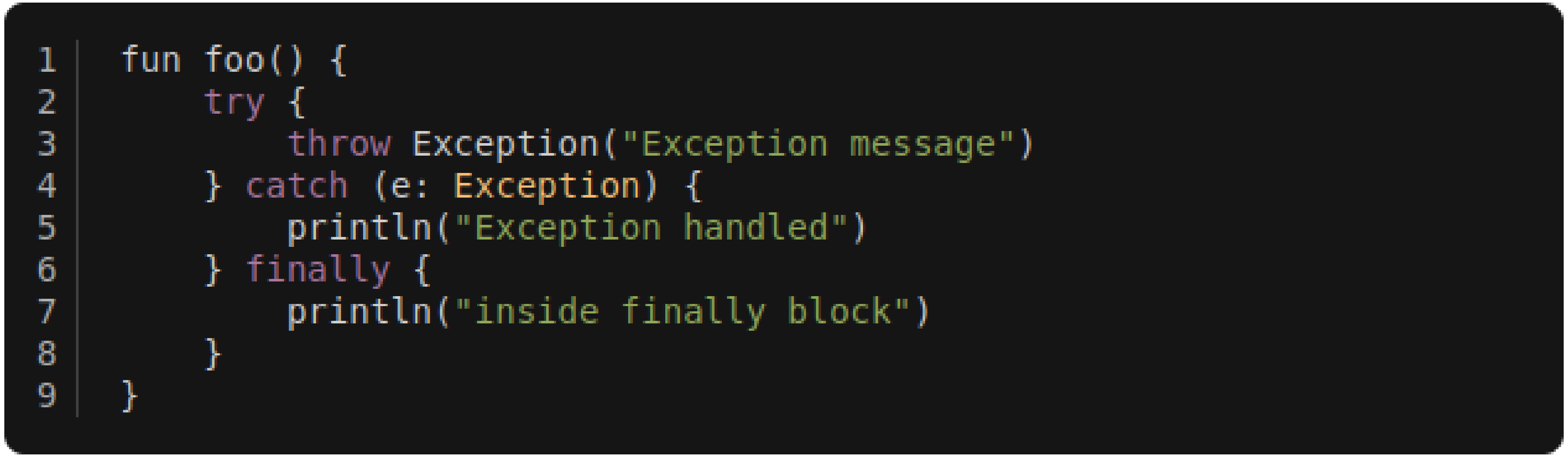
throws can not be included in function declarations where they may occur

EXCEPTIONS

This does not mean that we cannot handle exceptions or throw exceptions in our code



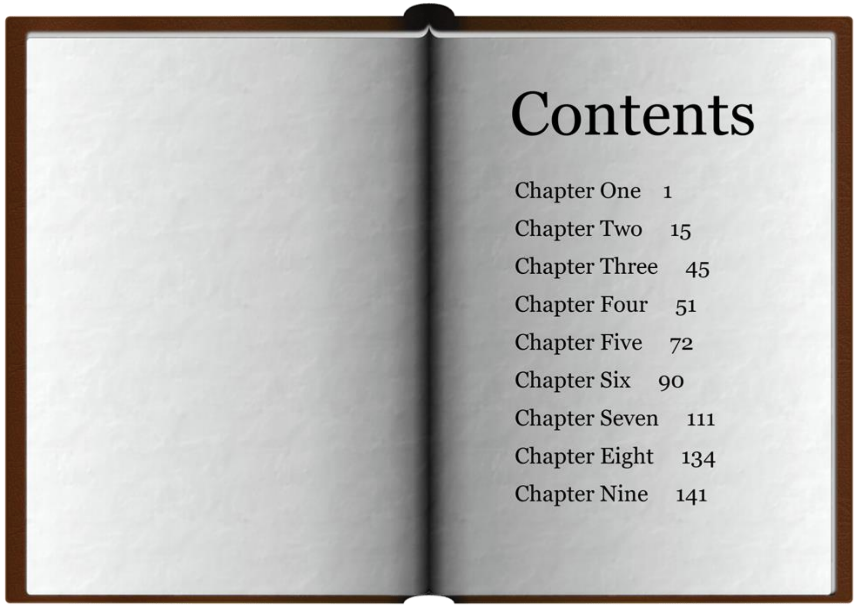
```
try {

   // some code

} catch (e: SomeException) {

   // handler

} finally {

   // optional finally block

}
```



```
fun foo() {
    try {
        throw Exception("Exception message")
    } catch (e: Exception) {
        println("Exception handled")
    } finally {
        println("inside finally block")
    }
}
```

Content

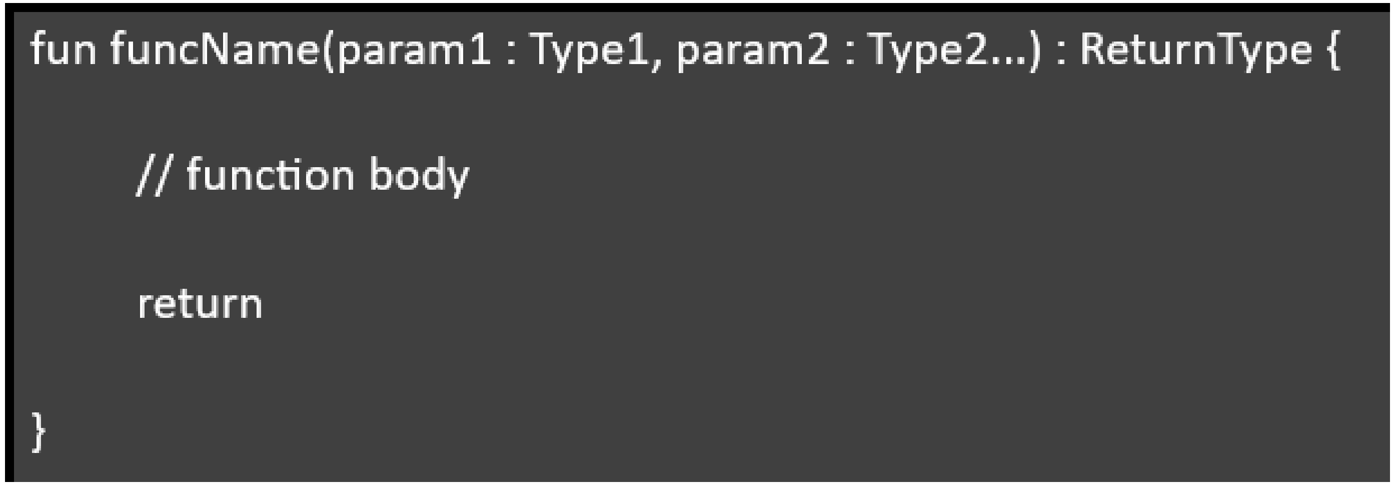
1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE



Contents

FUNCTIONS

Definition and Invocation



```
fun funcName(param1 : Type1, param2 : Type2...) : ReturnType {

    // function body

    return

}
```

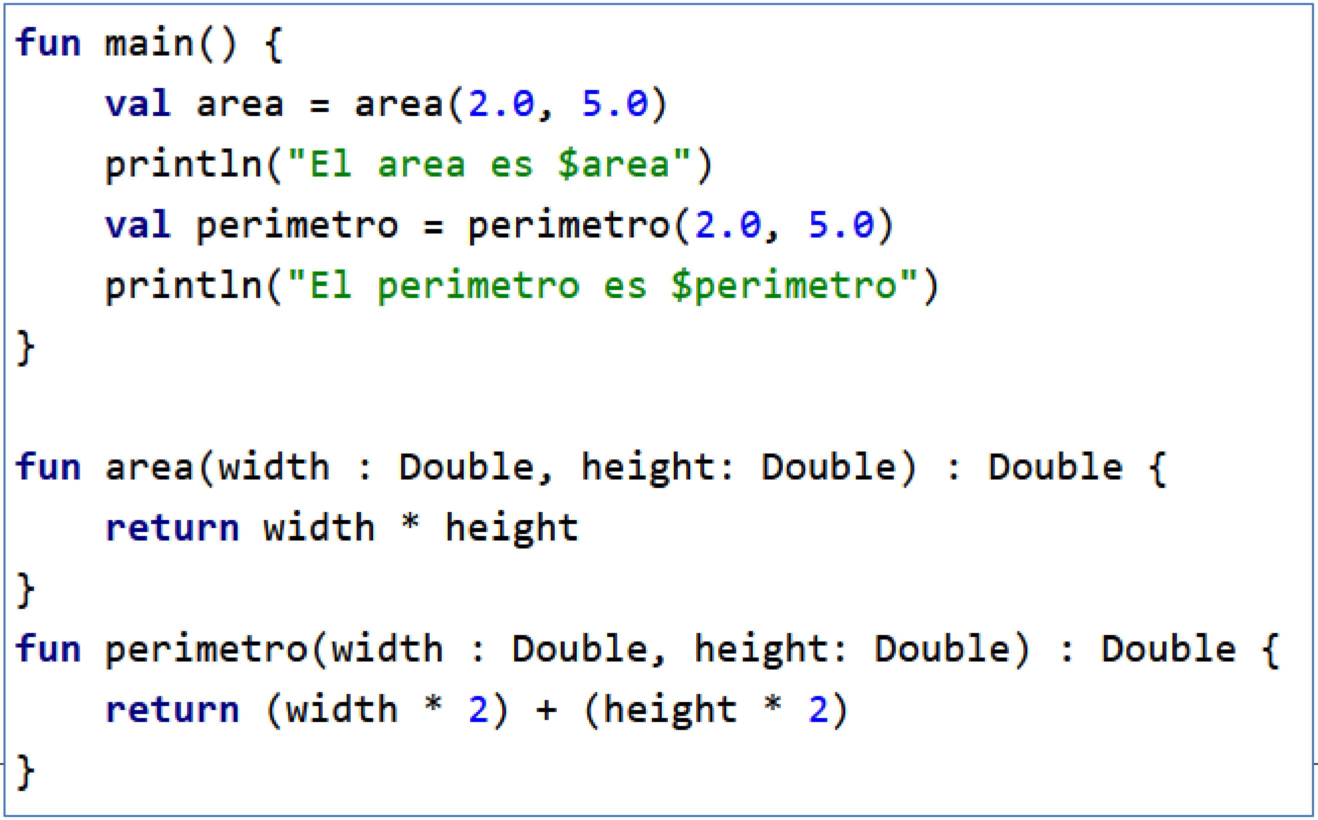
How did we call that in Java?

- Function parameters are specified in the form *parameter : Type*
- These types must necessarily be specified
- Return type may be specified after the parenthesis
- When the function does not return a value, its default return type is *Unit*

FUNCTIONS

Definition and Invocation

- Write a program that asks for the width and height of a rectangle and displays its area and perimeter. Implement a function for each thing

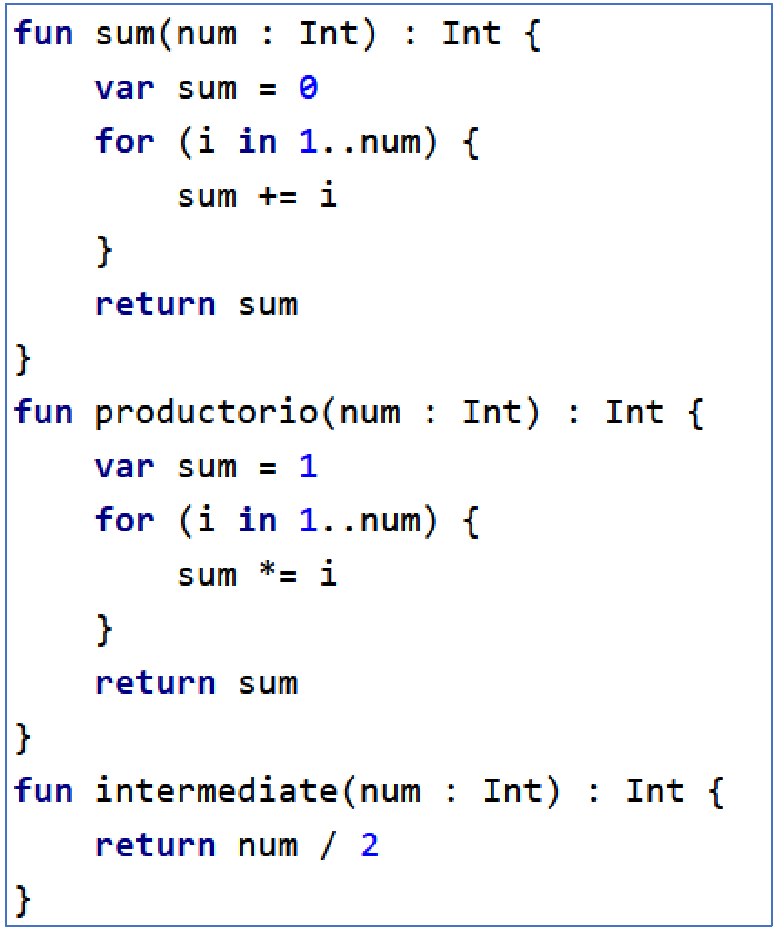


```
fun main() {
    val area = area(2.0, 5.0)
    println("El area es $area")
    val perimetro = perimetro(2.0, 5.0)
    println("El perimetro es $perimetro")
}

fun area(width : Double, height: Double) : Double {
    return width * height
}
fun perimetro(width : Double, height: Double) : Double {
    return (width * 2) + (height * 2)
}
```

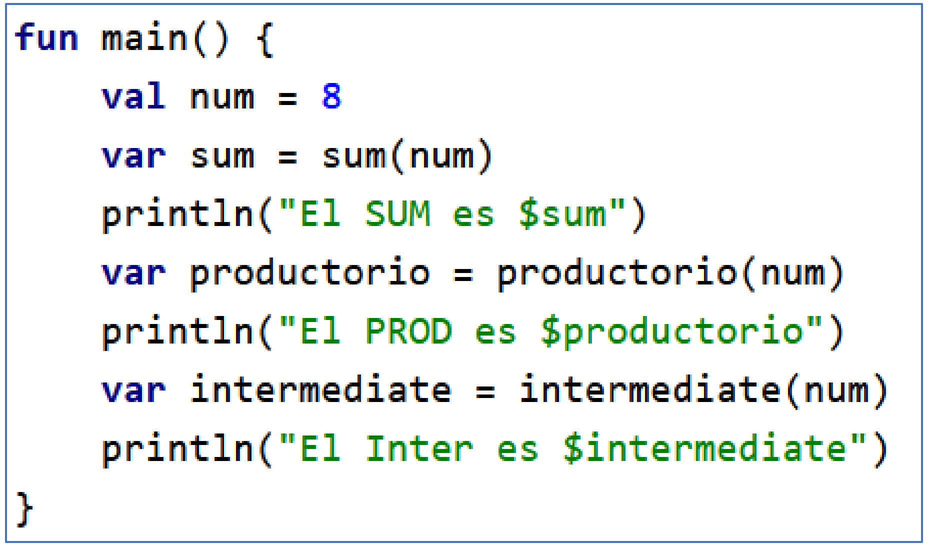
FUNCTIONS

Definition and Invocation

- Write a program that asks for an integer value N and then show: the SUM from 1 to N, the "*productorio*" from 1 to N and the intermediate value between 1 and N. Implement a function for each thing



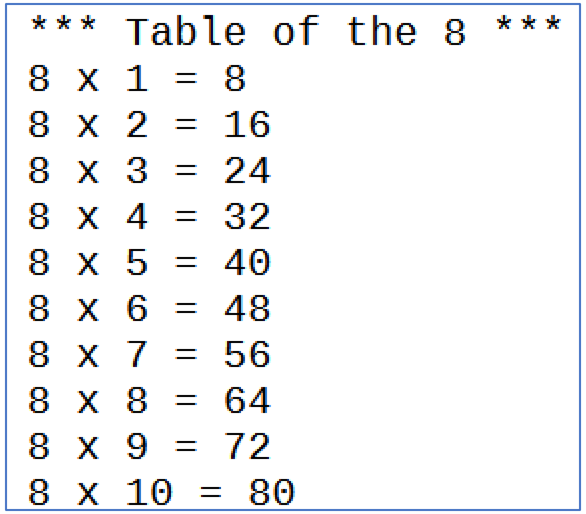
```
fun sum(num : Int) : Int {
    var sum = 0
    for (i in 1..num) {
        sum += i
    }
    return sum
}
fun productorio(num : Int) : Int {
    var sum = 1
    for (i in 1..num) {
        sum *= i
    }
    return sum
}
fun intermediate(num : Int) : Int {
    return num / 2
}
```



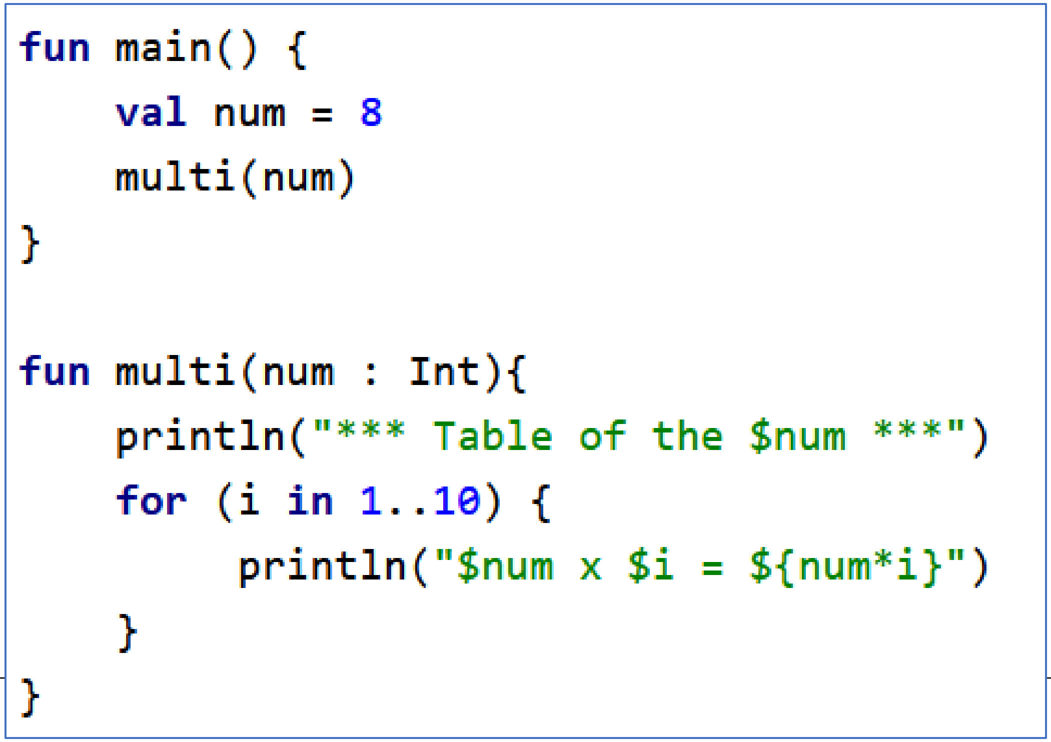
```
fun main() {
    val num = 8
    var sum = sum(num)
    println("El SUM es $sum")
    var productorio = productorio(num)
    println("El PROD es $productorio")
    var intermediate = intermediate(num)
    println("El Inter es $intermediate")
}
```

Definition and Invocation

- Make a program that writes the multiplication table of an integer. Implement a function that receives as parameter a number and displays on the screen the multiplication table of this number



```
*** Table of the 8 ***
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80
```

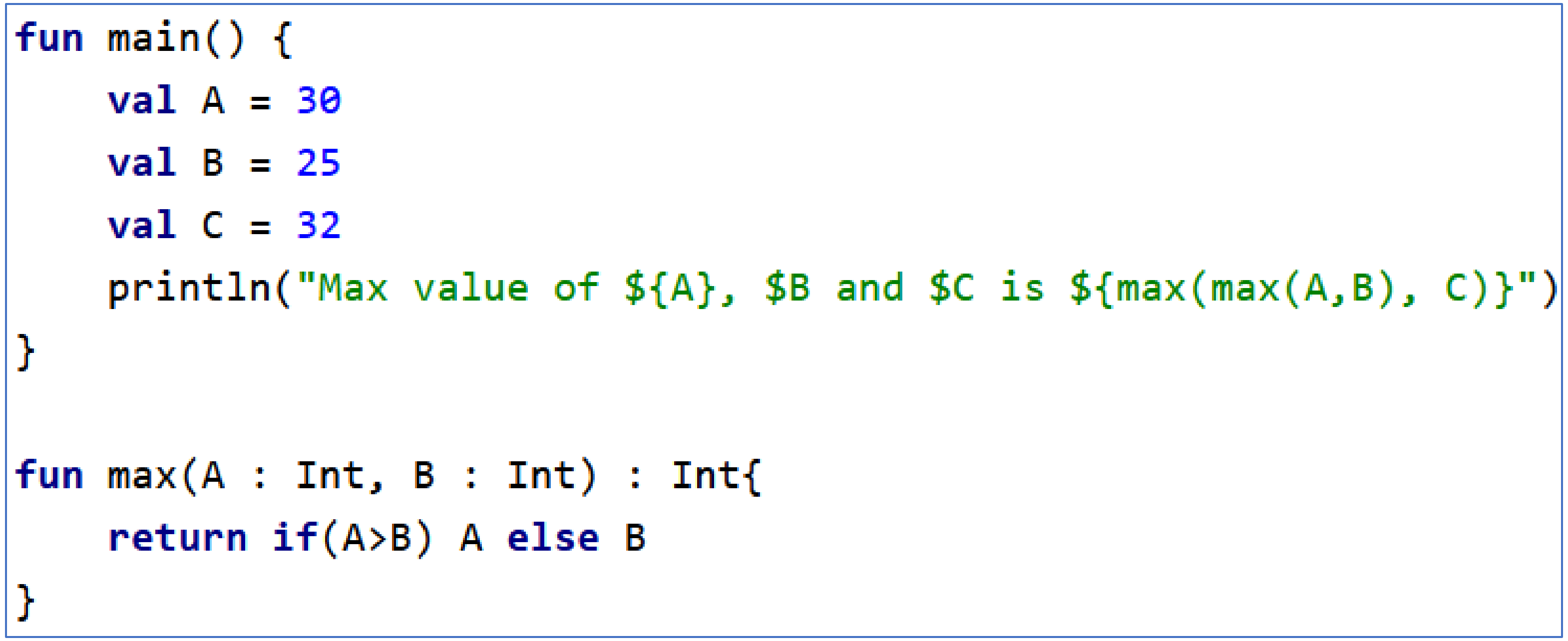


```
fun main() {
    val num = 8
    multi(num)
}

fun multi(num : Int){
    println("*** Table of the $num ***")
    for (i in 1..10) {
        println("$num x $i = ${num*i}")
    }
}
```

FUNCTIONS

Definition and Invocation

- Make a program that tells which of a given set of three integer values is the highest. Implement it by creating only one function to which **we pass two values (not three)** and return the maximum of the two values.



```
fun main() {
    val A = 30
    val B = 25
    val C = 32
    println("Max value of ${A}, $B and $C is ${max(max(A,B), C)}")
}

fun max(A : Int, B : Int) : Int{
    return if(A>B) A else B
}
```

FUNCTIONS

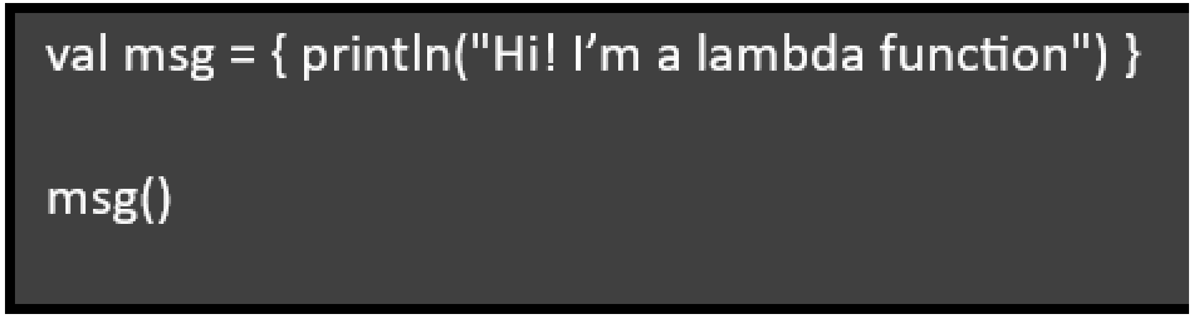
Lambda Expressions

- Represents the block of a function and simplifies the code
- Characteristics:
  - It has no *fun* keyword and access modifiers (private, public or protected)
  - It is an anonymous function (no name)
  - Return type is inferred by the compiler
  - The last expression is considered the return value

FUNCTIONS

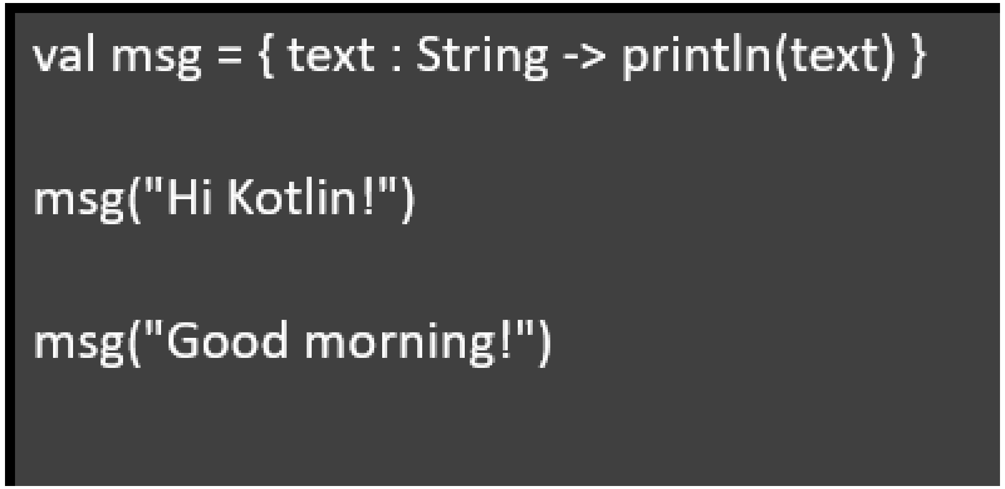
Lambda Expressions

Without parameters and assigned to a variable



```
val msg = { println("Hi! I’m a lambda function") }

msg()
```

With parameters

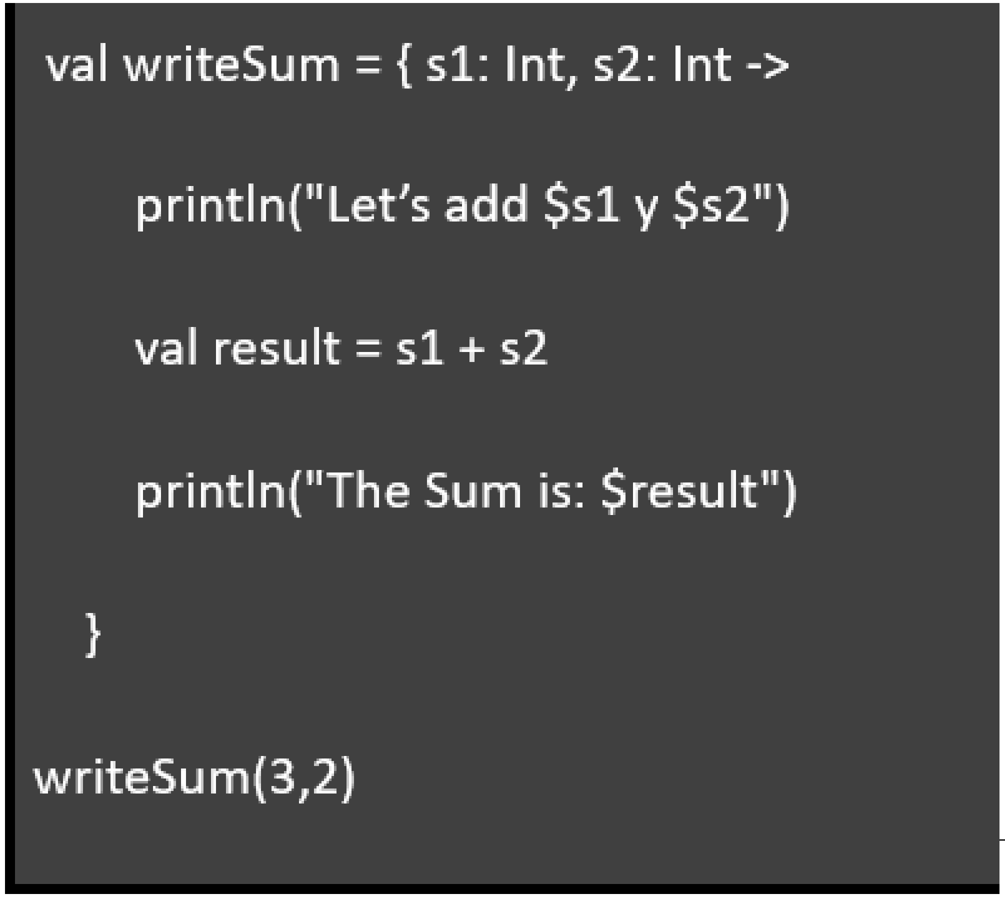


```
val msg = { text : String -> println(text) }

msg("Hi Kotlin!")

msg("Good morning!")
```

FUNCTIONS

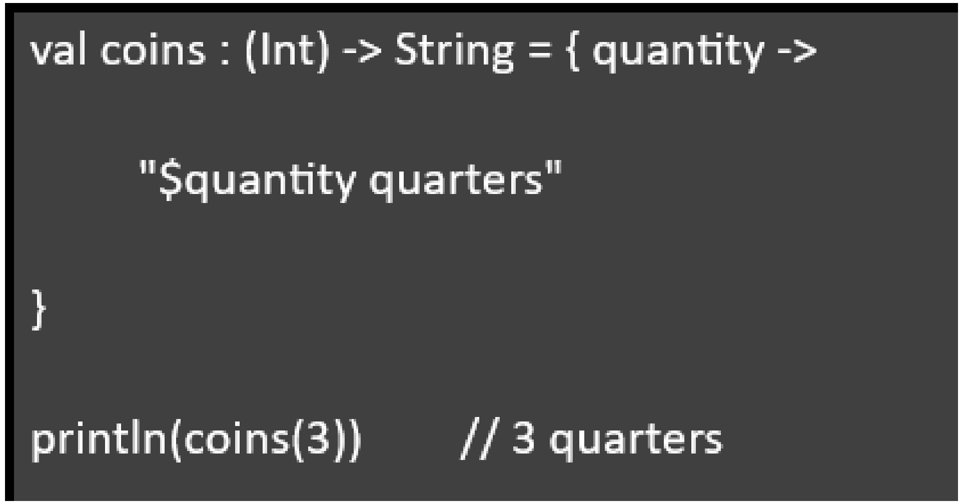
Lambda Expressions

With N parameters

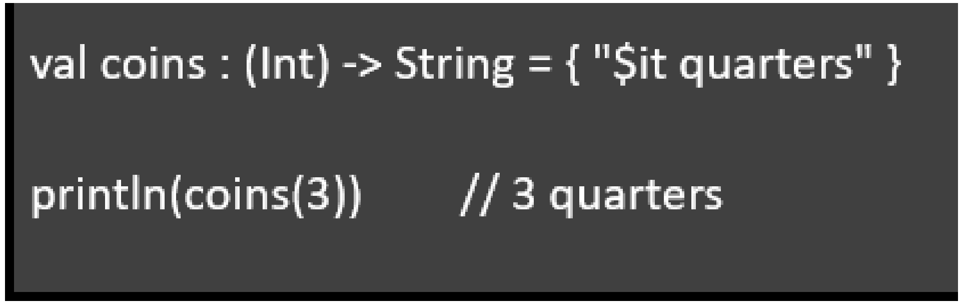


```
val writeSum = { s1: Int, s2: Int ->

      println("Let’s add $s1 y $s2")

      val result = s1 + s2

      println("The Sum is: $result")

  }

writeSum(3,2)
```

Omitting parameters



```
val coins : (Int) -> String = { quantity ->

       "$quantity quarters"

}

println(coins(3))      // 3 quarters
```



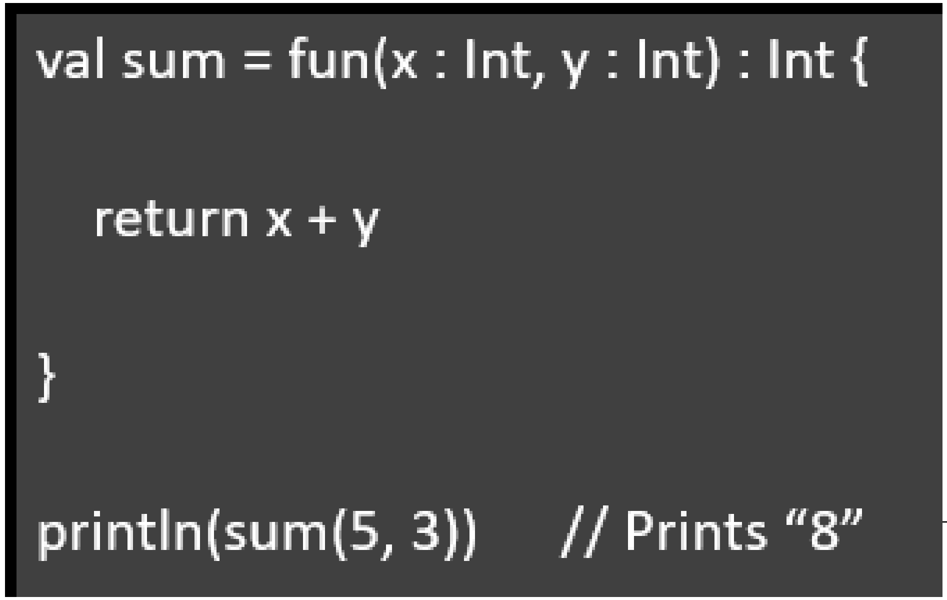
```
val coins : (Int) -> String = { "$it quarters" }

println(coins(3))      // 3 quarters
```

use of *it* when only one param

FUNCTIONS

Anonymous Functions

- These functions can be assigned to variables or passed as arguments to other functions
- They are often used to implement functional interfaces, such as *Runnable* or *OnClickListener*

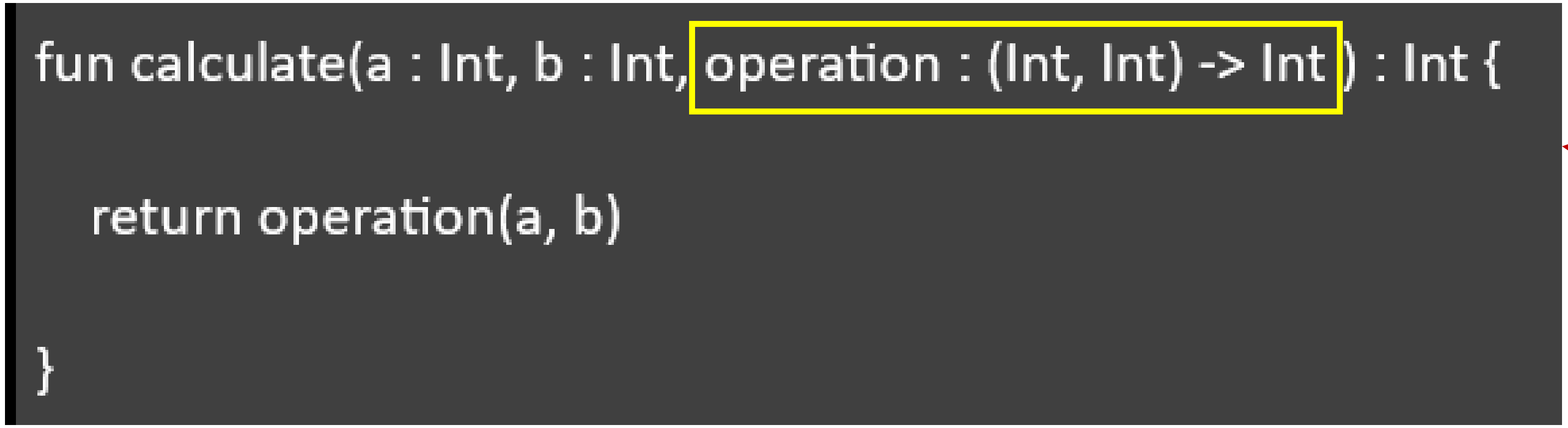


```
val sum = fun(x : Int, y : Int) : Int {

   return x + y

}

println(sum(5, 3))     // Prints “8”
```

FUNCTIONS

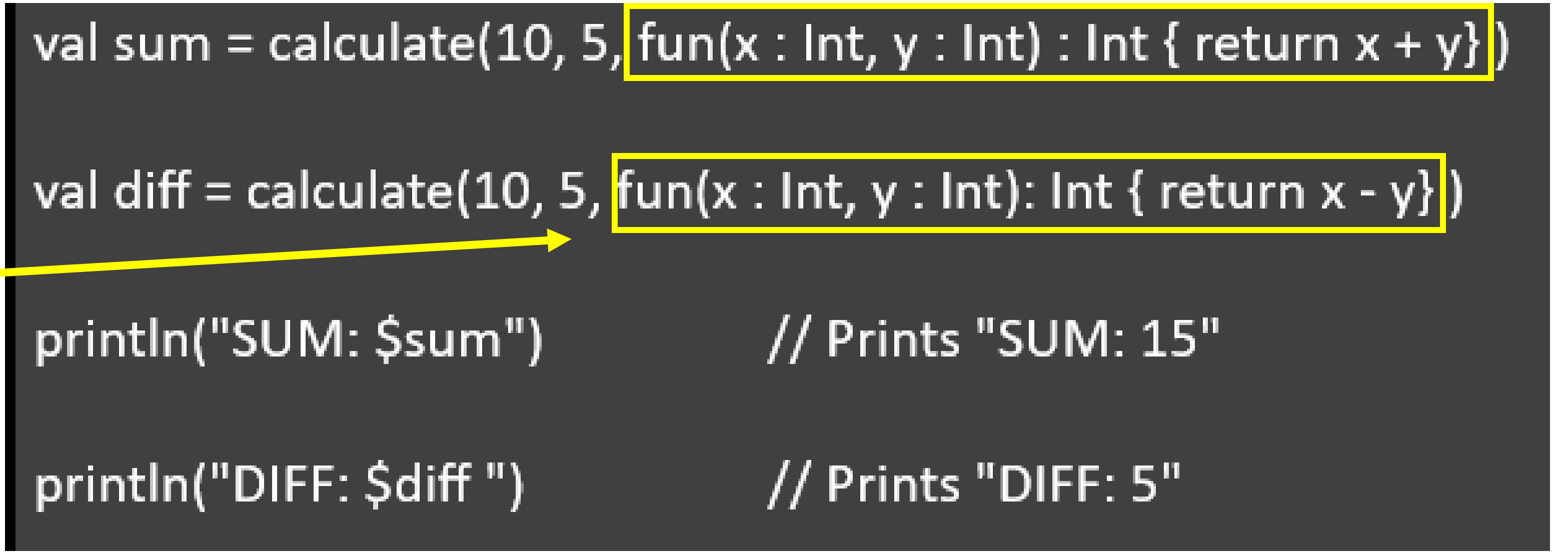
Anonymous Functions

“Normal” function



```
fun calculate(a : Int, b : Int, operation : (Int, Int) -> Int ) : Int {

    return operation(a, b)

}
```

Which are the parameters and the return types?

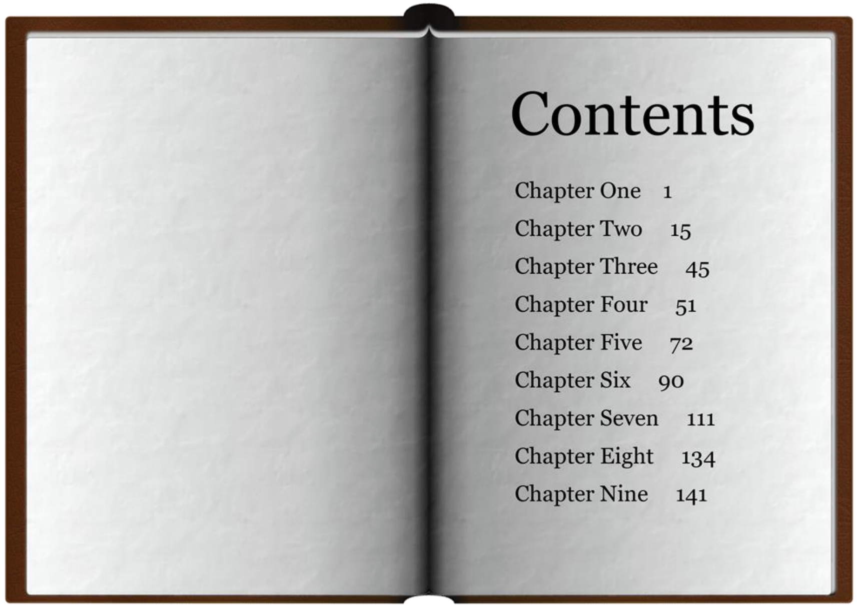


```
val sum = calculate(10, 5, fun(x : Int, y : Int) : Int { return x + y})

val diff = calculate(10, 5, fun(x : Int, y : Int): Int { return x - y})

println("SUM: $sum")          // Prints "SUM: 15"

println("DIFF: $diff ")       // Prints "DIFF: 5"
```

Anonymous functions as parameters

Content

1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPE

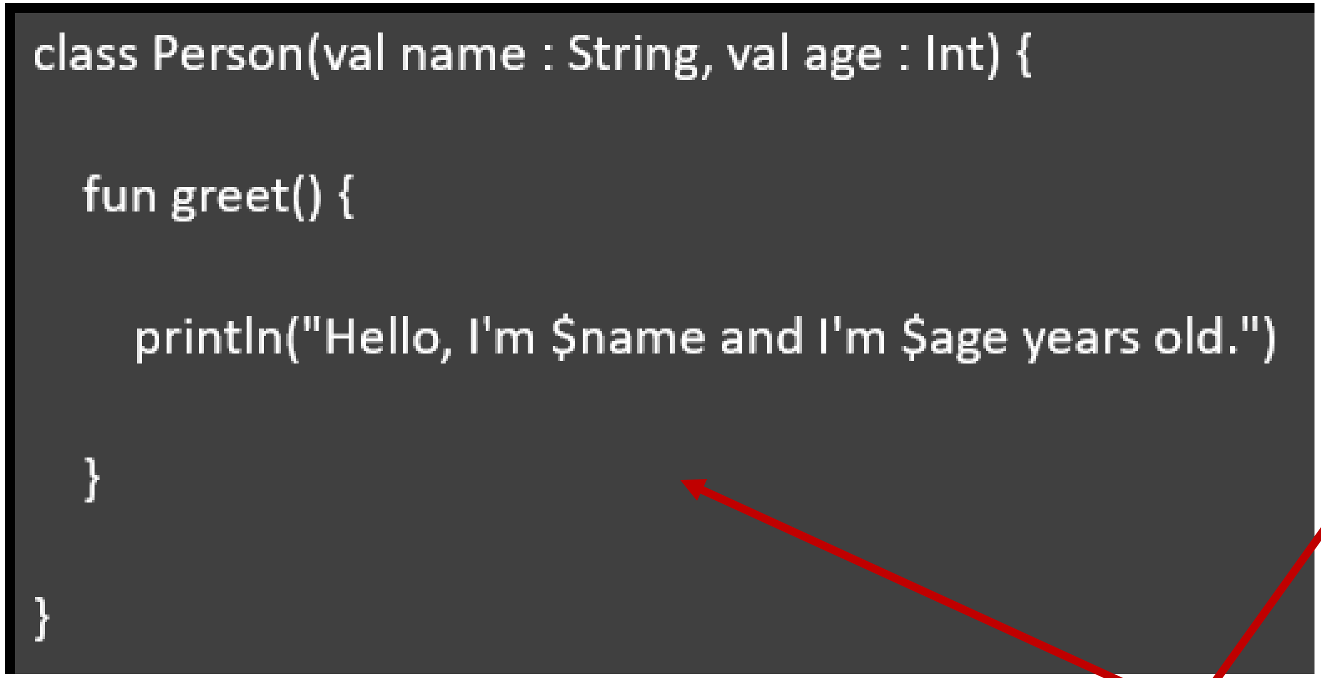


Contents

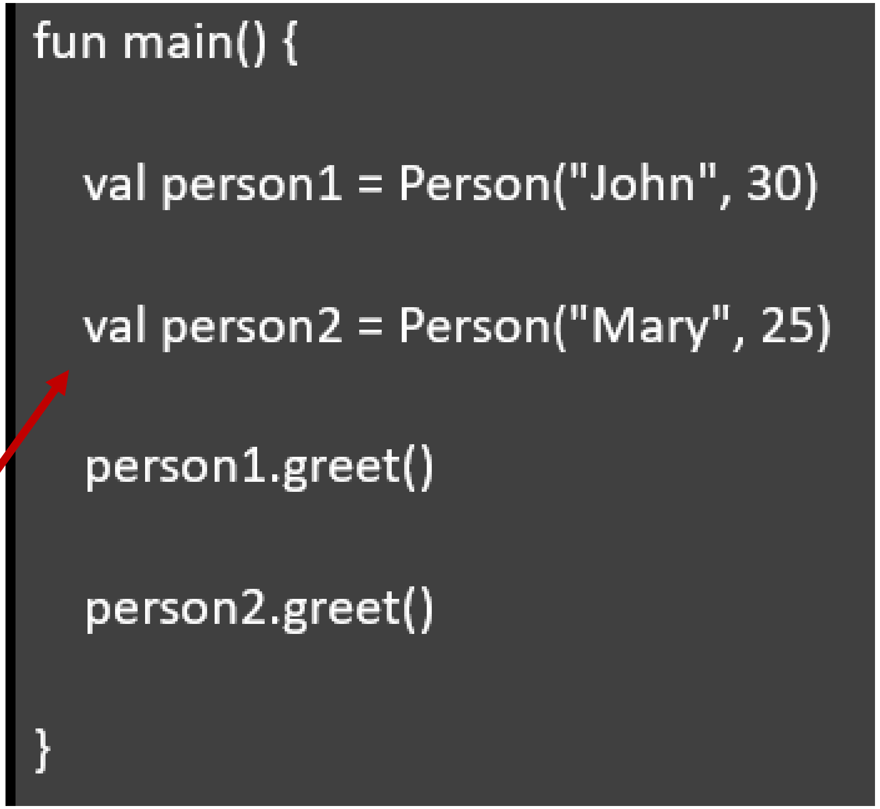
Chapter One 1
Chapter Two 15
Chapter Three 45
Chapter Four 51
Chapter Five 72
Chapter Six 90
Chapter Seven 111
Chapter Eight 134
Chapter Nine 141

OBJECT ORIENTED PROGRAMMING (OOP)

Classes and Objects



```
class Person(val name : String, val age : Int) {
    fun greet() {
        println("Hello, I'm $name and I'm $age years old.")
    }
}
```



```
fun main() {
    val person1 = Person("John", 30)
    val person2 = Person("Mary", 25)
    person1.greet()
    person2.greet()
}
```

How did we code that in Java?

OBJECT ORIENTED PROGRAMMING (OOP)

Classes and Objects

DIY

Create a class called Person that will represent the main data of a person: ID, name, surname and age.
Add the following methods to the class:

- toString: Returns the information of the object: “ID:... Name:... etc.".
- isFullAge: Returns true if over the age of 18
- isRetired: Returns true if 65 years of age or older
- ageDiff: Returns the age difference between the person and another person

Instantiate two objects of the class Person and:

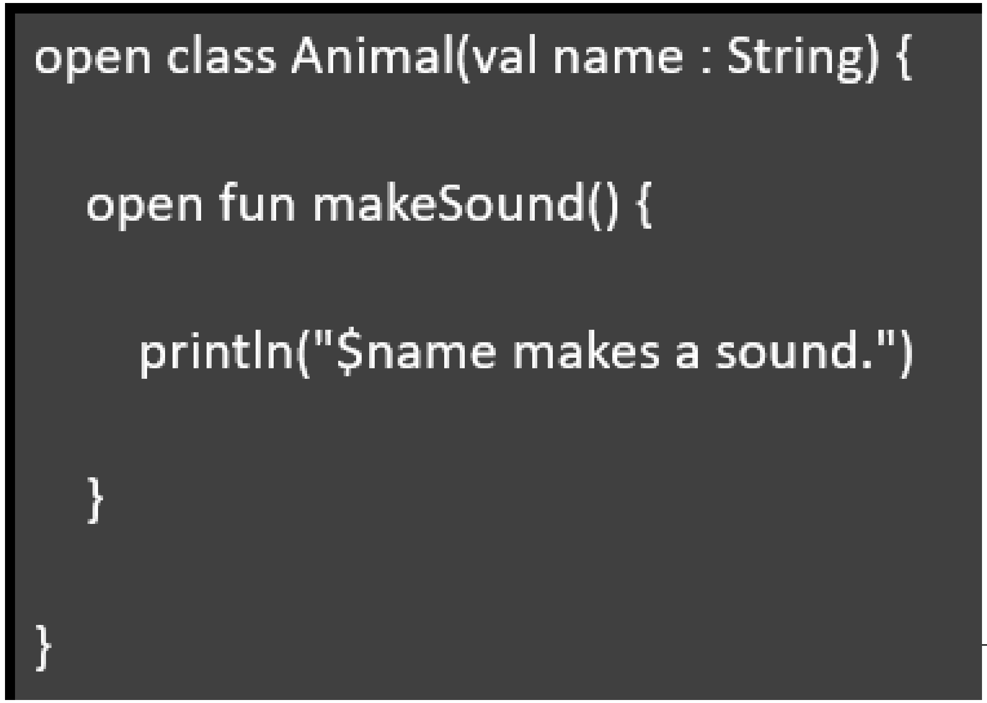
- Print their characteristics on the screen, showing whether or not they are above 18 and/or retired
- Displays a message with the age difference between them

OBJECT ORIENTED PROGRAMMING (OOP)

Inheritance

What’s that?

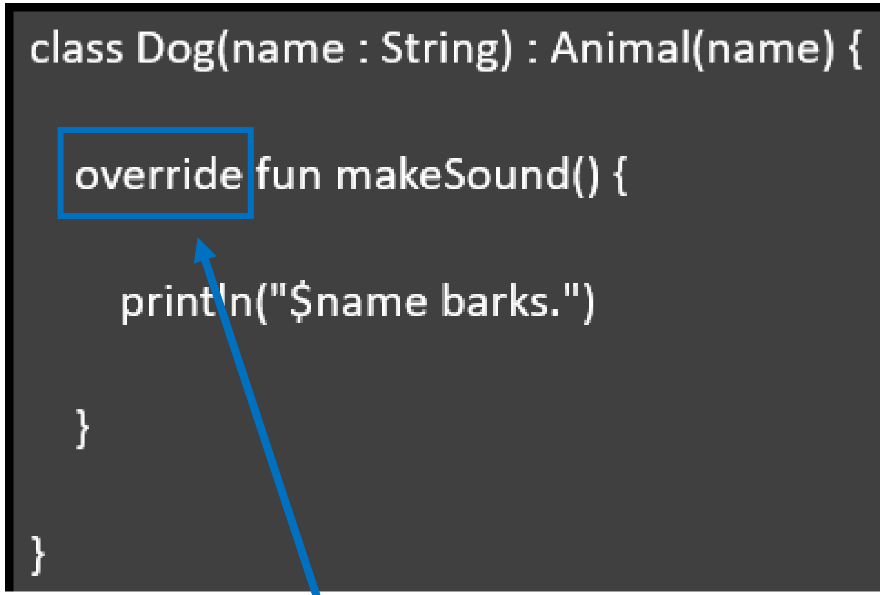
- Kotlin classes and their functions are *final* **by default**
- To allow a class to be extended it must be marked *open*
- To allow class functions and fields to be overridden, they must also be marked *open*



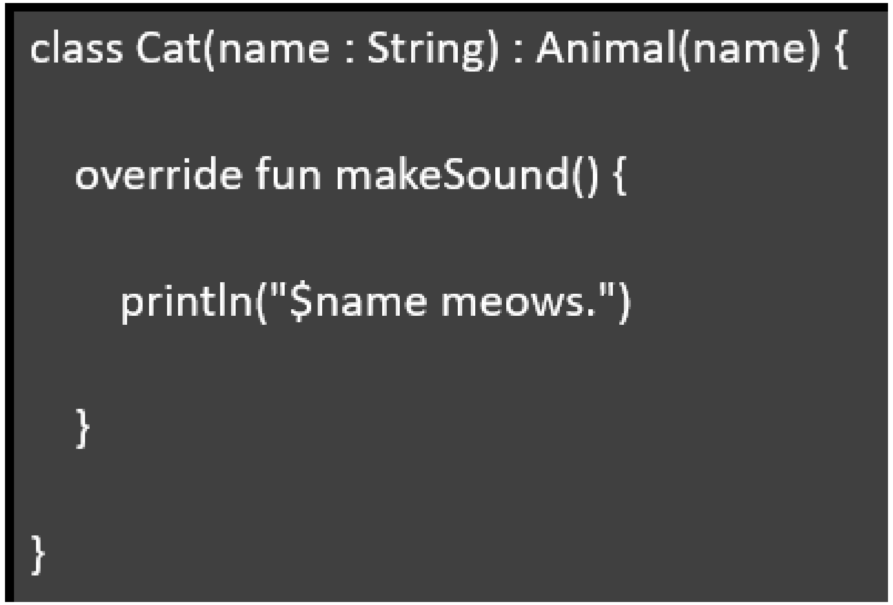
```
open class Animal(val name : String) {

    open fun makeSound() {

        println("$name makes a sound.")

    }

}
```

OBJECT ORIENTED PROGRAMMING (OOP)

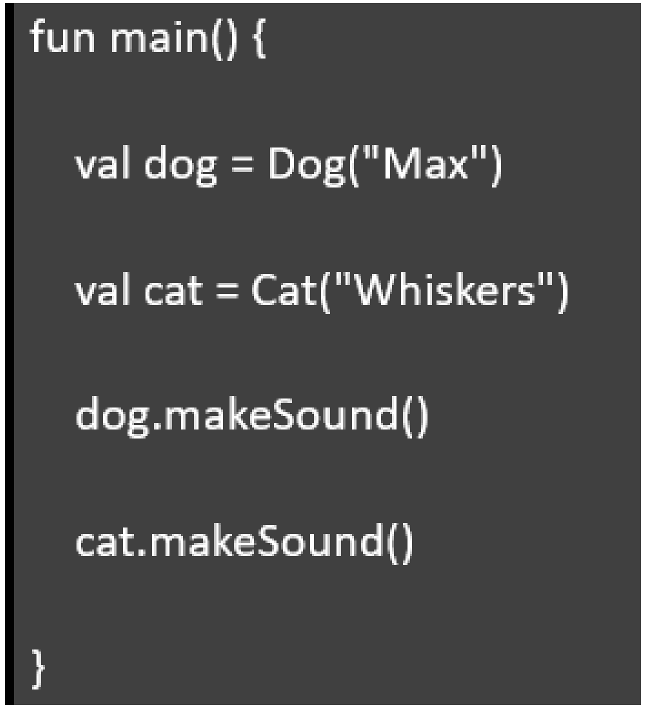
Inheritance



```
class Dog(name : String) : Animal(name) {
    override fun makeSound() {
        println("$name barks.")
    }
}
```



```
class Cat(name : String) : Animal(name) {
    override fun makeSound() {
        println("$name meows.")
    }
}
```



```
fun main() {
    val dog = Dog("Max")
    val cat = Cat("Whiskers")
    dog.makeSound()
    cat.makeSound()
}
```

One has to define that the parent function will be overridden

Inheritance

DIY

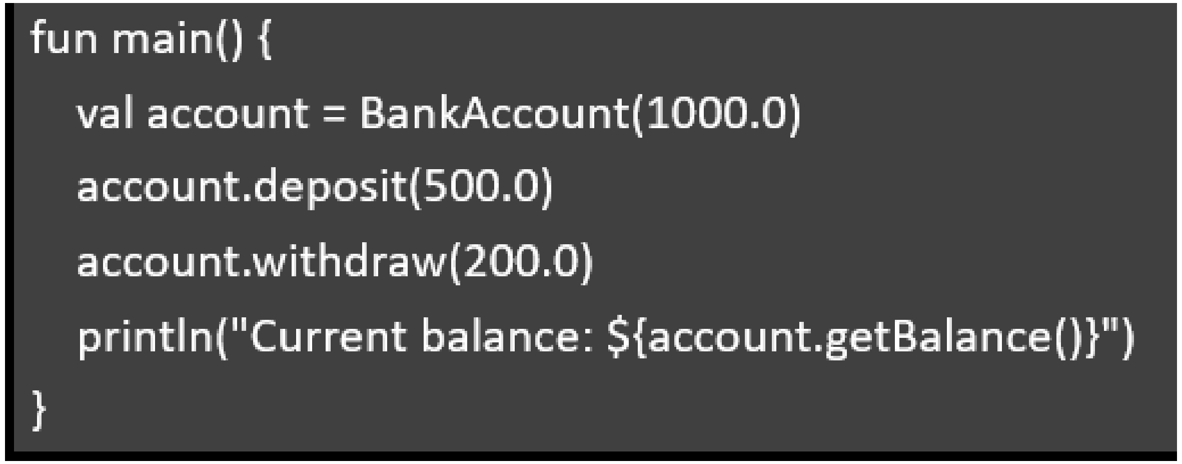
Create subclasses Student and Teacher from Person:

- Student will have a ‘level‘ attribute and Teacher a ‘center’ one.
- Update the toString() method to include the new attributes
- Instantiate a person of each type, show their attributes and the age difference between them

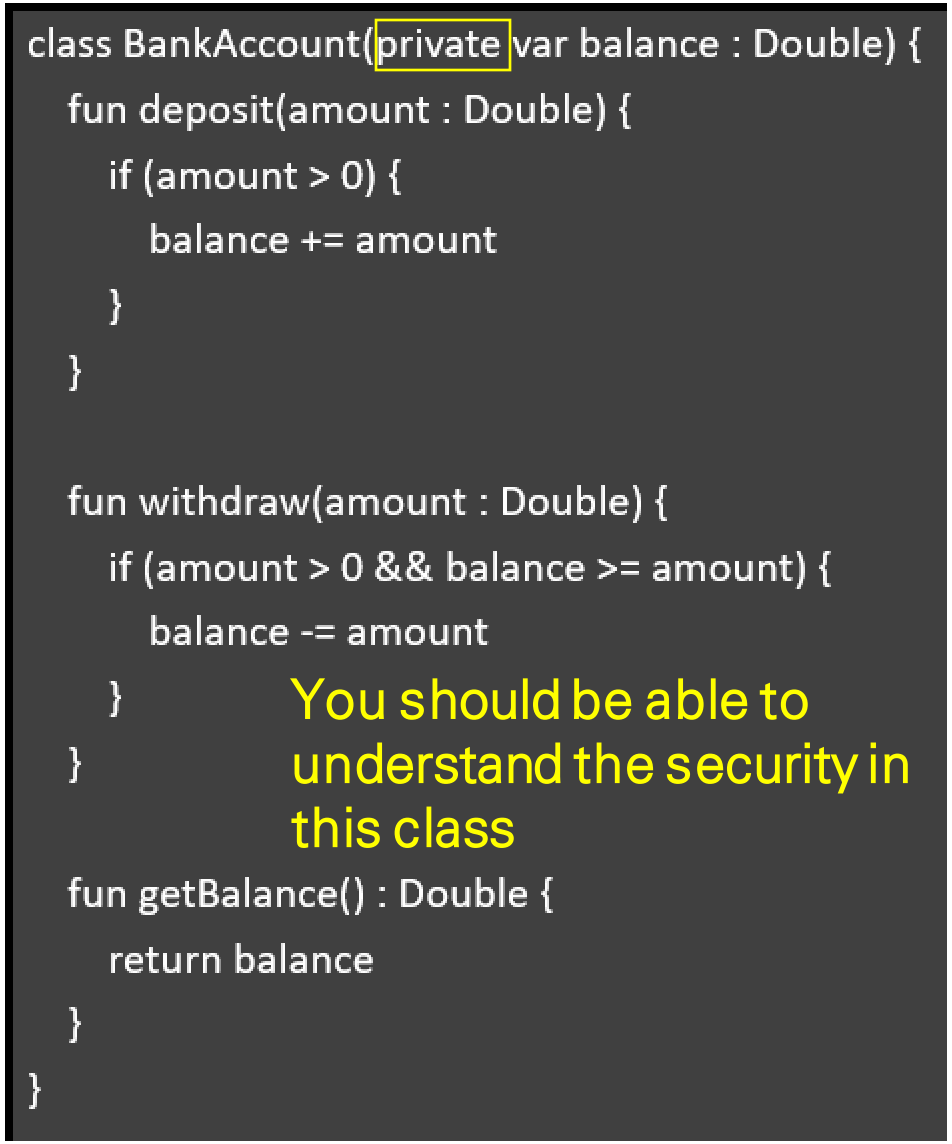
OBJECT ORIENTED PROGRAMMING (OOP)

What's that? Encapsulation

Kotlin allows you to control access to a class's properties and methods using access modifiers like *private, protected, internal* (module) and *public* (default)



```
fun main() {
    val account = BankAccount(1000.0)
    account.deposit(500.0)
    account.withdraw(200.0)
    println("Current balance: ${account.getBalance()}")
}
```



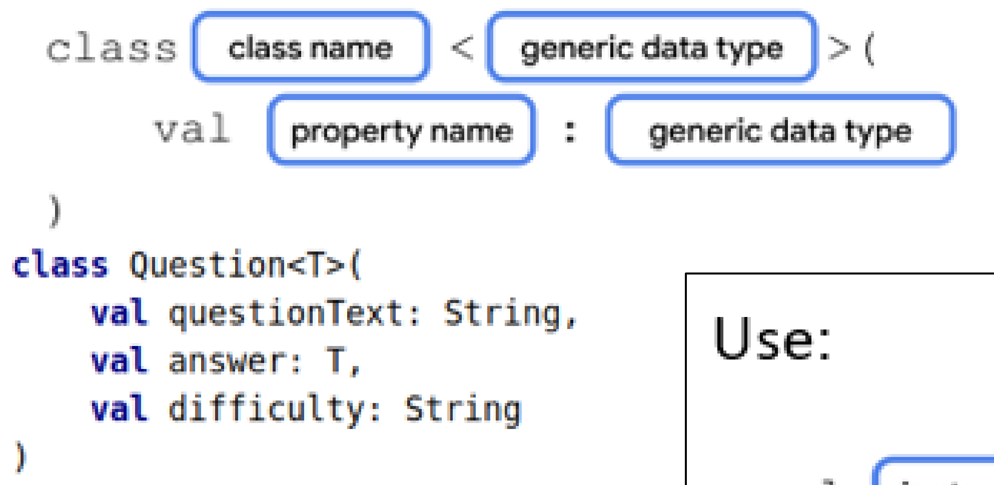
```
class BankAccount(private var balance : Double) {
    fun deposit(amount : Double) {
        if (amount > 0) {
            balance += amount
        }
    }

    fun withdraw(amount : Double) {
        if (amount > 0 && balance >= amount) {
            balance -= amount
        }
    }

    fun getBalance() : Double {
        return balance
    }
}
```

You should be able to understand the security in this class

OBJECT ORIENTED PROGRAMMING (OOP)

What's that? **Use of Generics**

Definition:



```
class  class name  <  generic data type  > (
     val  property name  :  generic data type
)
```

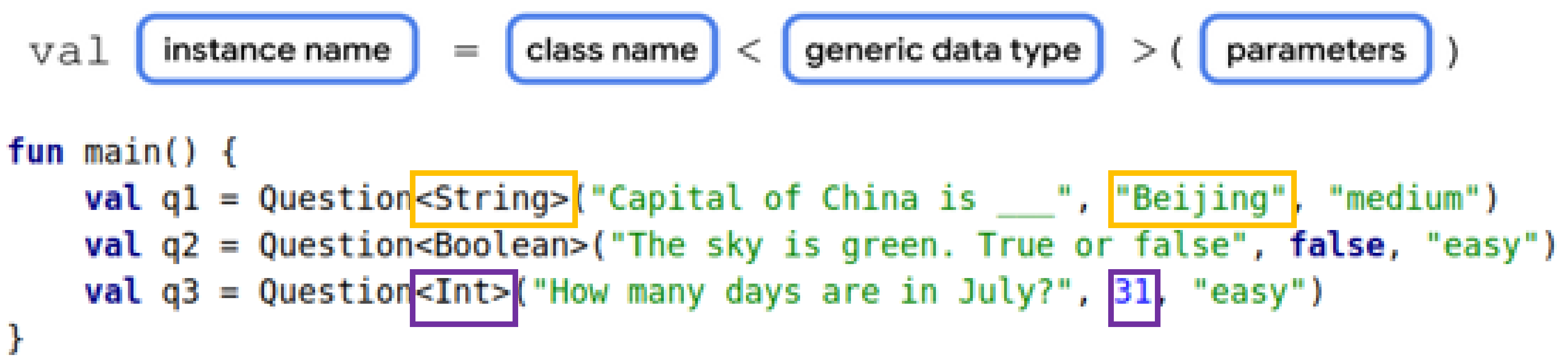
```
class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: String
)
```

DIY

Can a class have more than one generic?

```
class Question <T, Q> (
    val questionText : String,
    val answer : T,
    val difficulty: Q
)
```

Use:



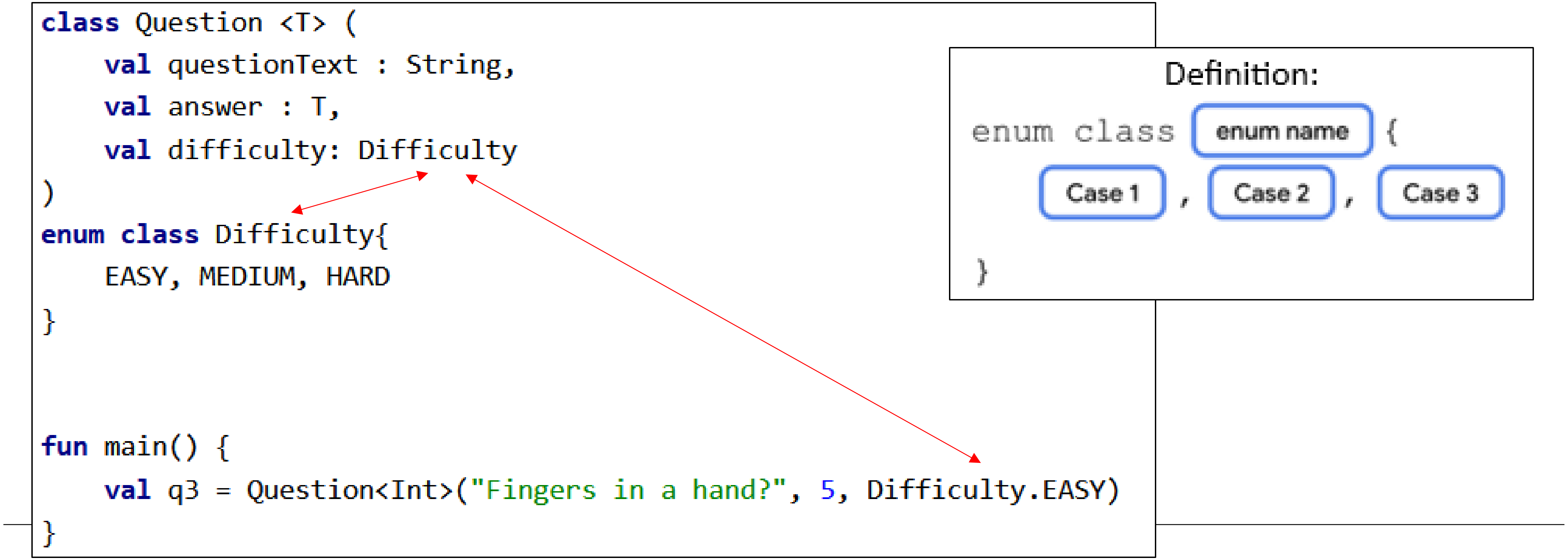
```
val  instance name  =  class name  <  generic data type  > (  parameters  )
```

```
fun main() {
    val q1 = Question<String>("Capital of China is ___", "Beijing", "medium")
    val q2 = Question<Boolean>("The sky is green. True or false", false, "easy")
    val q3 = Question<Int>("How many days are in July?", 31, "easy")
}
```

OBJECT ORIENTED PROGRAMMING (OOP)

ENUM classes

- Used to prevent programmers and users wrong typing
- Force a given set of values to be the only accepted ones (type-safe)



```
class Question <T> (
    val questionText : String,
    val answer : T,
    val difficulty: Difficulty
)
enum class Difficulty{
    EASY, MEDIUM, HARD
}


fun main() {
    val q3 = Question<Int>("Fingers in a hand?", 5, Difficulty.EASY)
}
```

Definition:

```
enum class  enum name  {
      Case 1 ,  Case 2 ,  Case 3
}
```

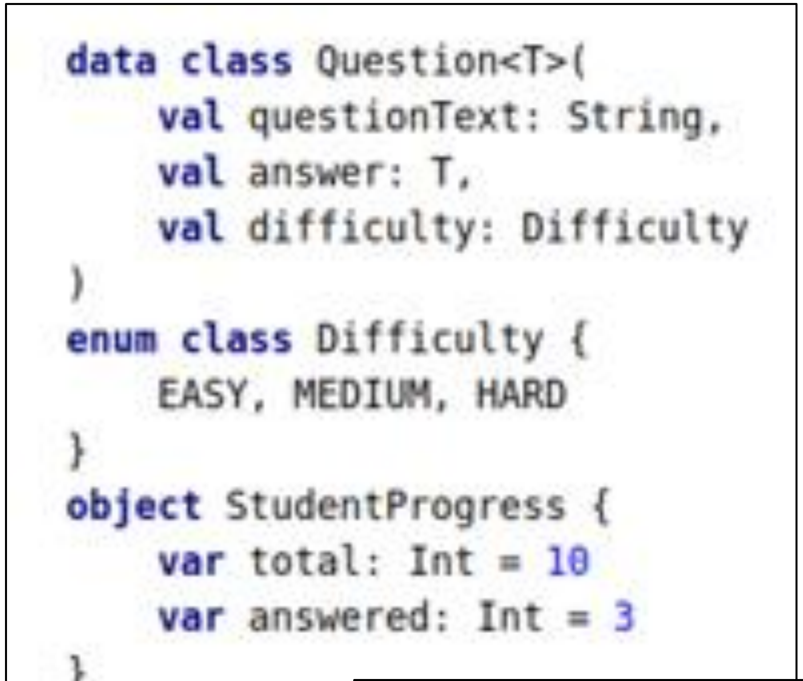
OBJECT ORIENTED PROGRAMMING (OOP)

Singleton Objects

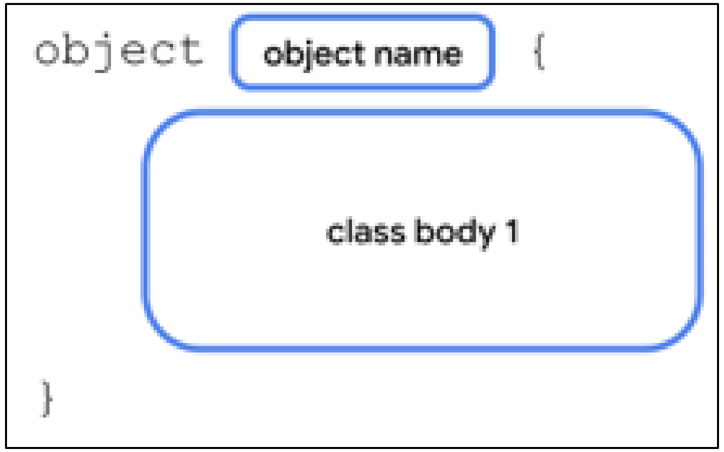
- There are cases where you want a class to **only have one instance**. For example:
  - Player stats in a mobile game for the current user
  - Interacting with a single hardware device, like sending audio through a speaker
  - Authentication, where only one user should be logged in at a time
- In the above scenarios, you'd probably need to use a class but only one instance of that class --> **Singleton Object**
- A singleton can't have a constructor as you can't create instances directly. Instead, all the properties are defined within the curly braces and are given an initial value

OBJECT ORIENTED PROGRAMMING (OOP)

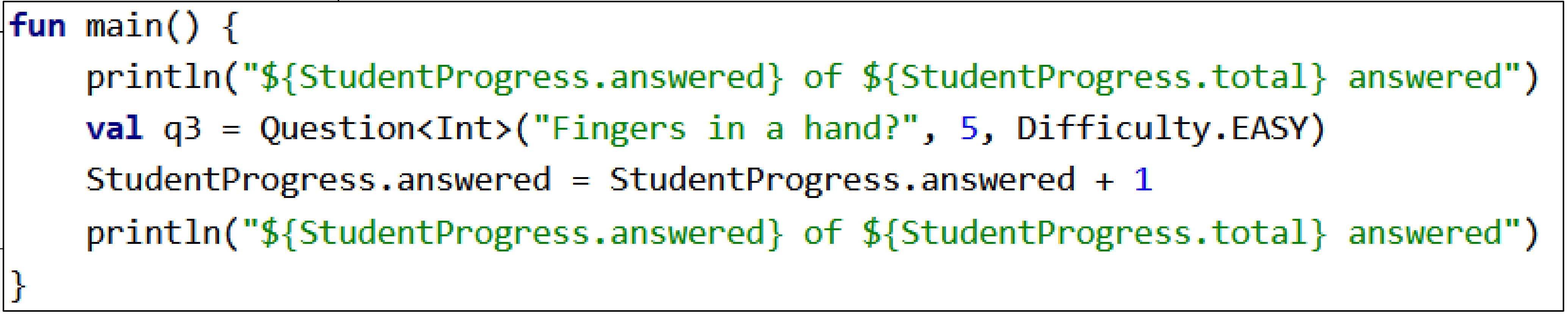
Singleton Objects



```
data class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)
enum class Difficulty {
    EASY, MEDIUM, HARD
}
object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
}
```



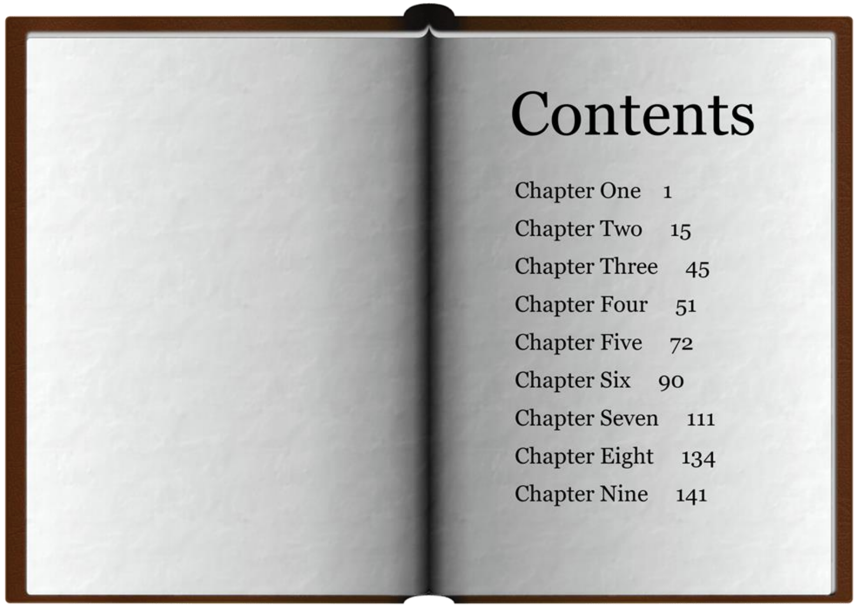
```
object object name {
        class body 1
}
```



```
fun main() {
    println("${StudentProgress.answered} of ${StudentProgress.total} answered")
    val q3 = Question<Int>("Fingers in a hand?", 5, Difficulty.EASY)
    StudentProgress.answered = StudentProgress.answered + 1
    println("${StudentProgress.answered} of ${StudentProgress.total} answered")
}
```

Content

1. INTRODUCTION
2. VARIABLES AND DATA TYPES
3. OPERATORS
4. CONTROL STRUCTURE
5. EXCEPTIONS
6. FUNCTIONS
7. OBJECT ORIENTED PROGRAMMING
8. COMPLEX DATA TYPES



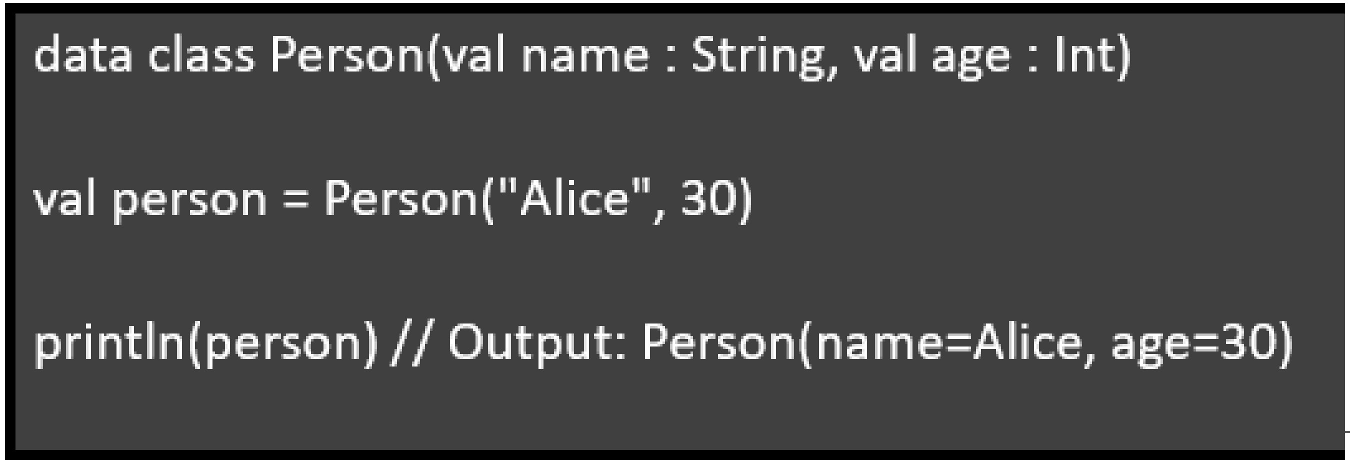
Contents

Chapter One	1
Chapter Two	15
Chapter Three	45
Chapter Four	51
Chapter Five	72
Chapter Six	90
Chapter Seven	111
Chapter Eight	134
Chapter Nine	141

COMPLEX DATA TYPES

Classes and Data Classes

- Data Classes are a special type of class designed primarily for holding data
- They automatically provide useful functions like `toString()`, `equals()`, and `hashCode()` based on their properties
- A data class needs to have at least one parameter in its constructor



```
data class Person(val name : String, val age : Int)

val person = Person("Alice", 30)

println(person) // Output: Person(name=Alice, age=30)
```

COMPLEX DATA TYPES

Classes and Data Classes

- Create and instantiate a HighSchool data class to keep track of the number of students and teachers in a given high-school (include a name parameter)
- Modify the classes to use the new HighSchool data class
- Increase the high-school parameters on every Student/Teacher instantiation
- Show center people totals on screen at the end

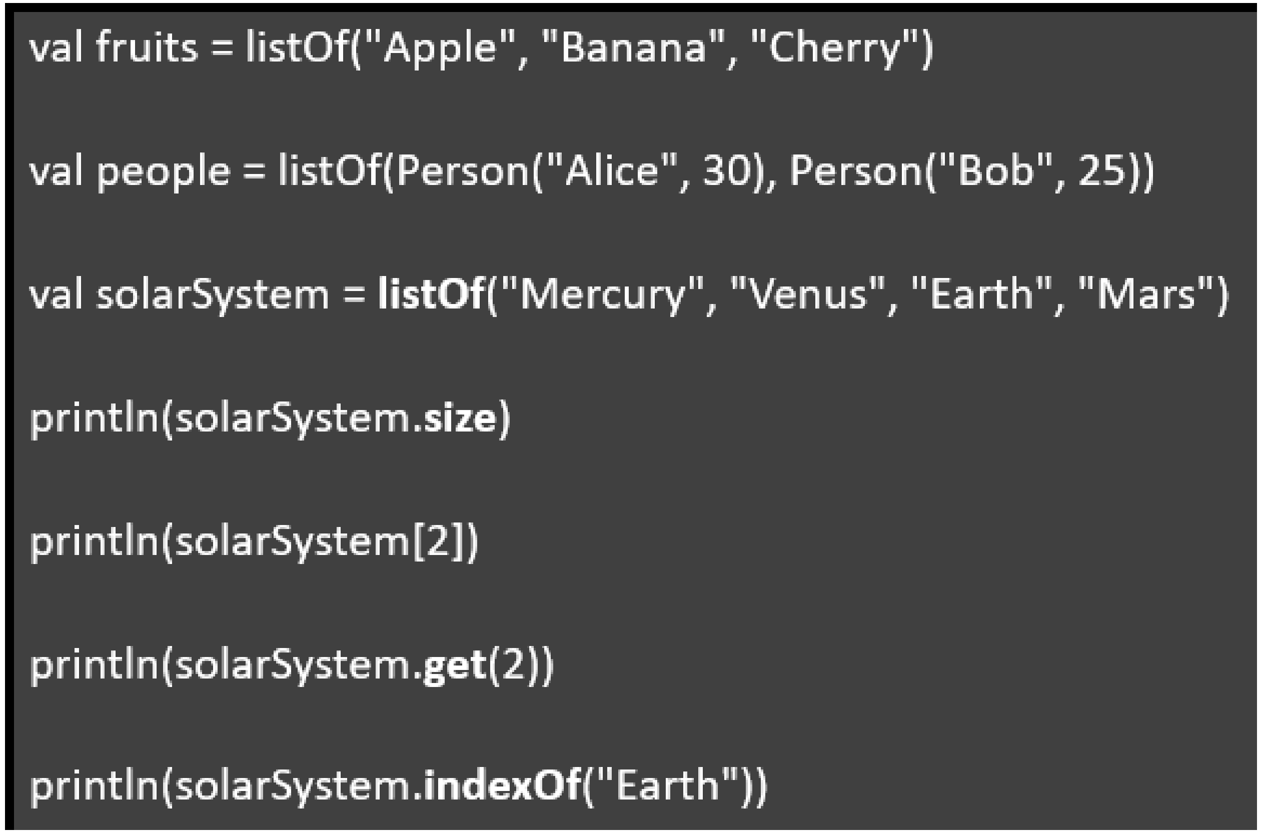
DIY

COMPLEX DATA TYPES

List / Mutablelist

Guess what?

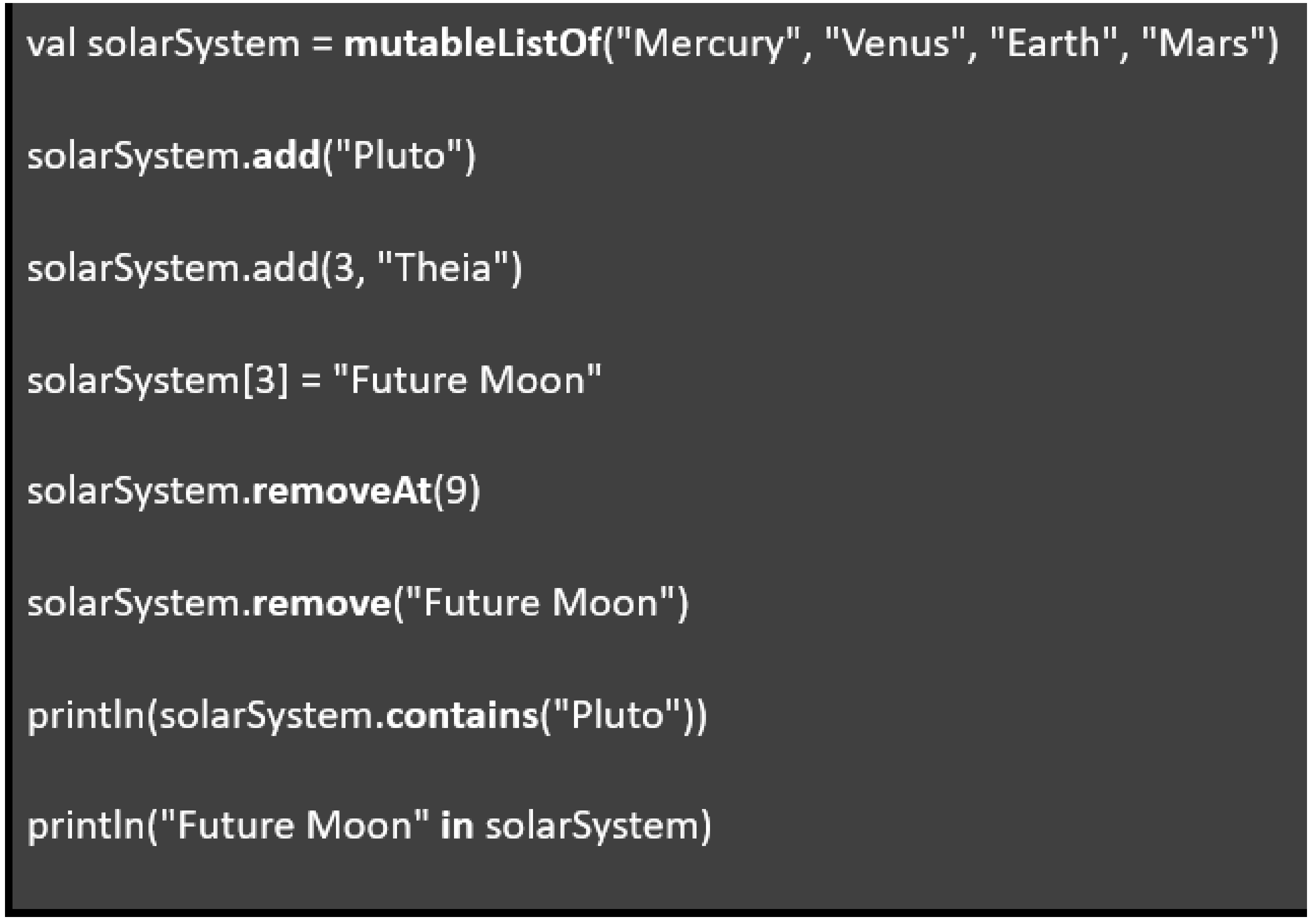
- **Ordered** collections that can store elements **of the same or different types**
- *List* is an interface that defines properties and methods related to a read-only ordered collections
- *MutableList* extends the *List* interface by defining methods to modify a list



```
val fruits = listOf("Apple", "Banana", "Cherry")

val people = listOf(Person("Alice", 30), Person("Bob", 25))

val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars")

println(solarSystem.size)

println(solarSystem[2])

println(solarSystem.get(2))

println(solarSystem.indexOf("Earth"))
```

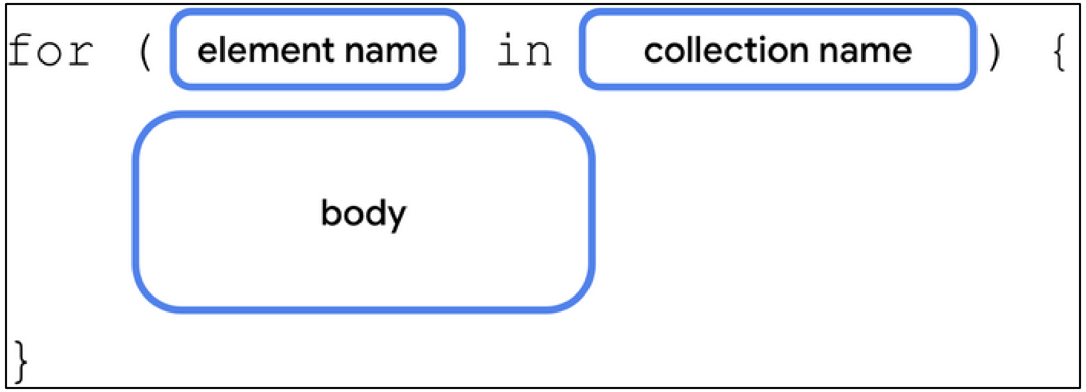
COMPLEX DATA TYPES

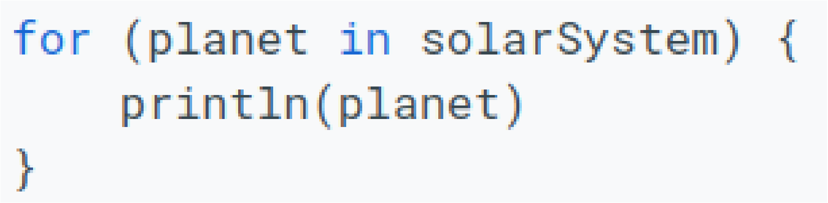
List / MutableList



```
val solarSystem = mutableListOf("Mercury", "Venus", "Earth", "Mars")

solarSystem.add("Pluto")

solarSystem.add(3, "Theia")

solarSystem[3] = "Future Moon"

solarSystem.removeAt(9)

solarSystem.remove("Future Moon")

println(solarSystem.contains("Pluto"))

println("Future Moon" in solarSystem)
```

Lists are collections easy to iterate using a *for* loop





```
for (planet in solarSystem) {
    println(planet)
}
```

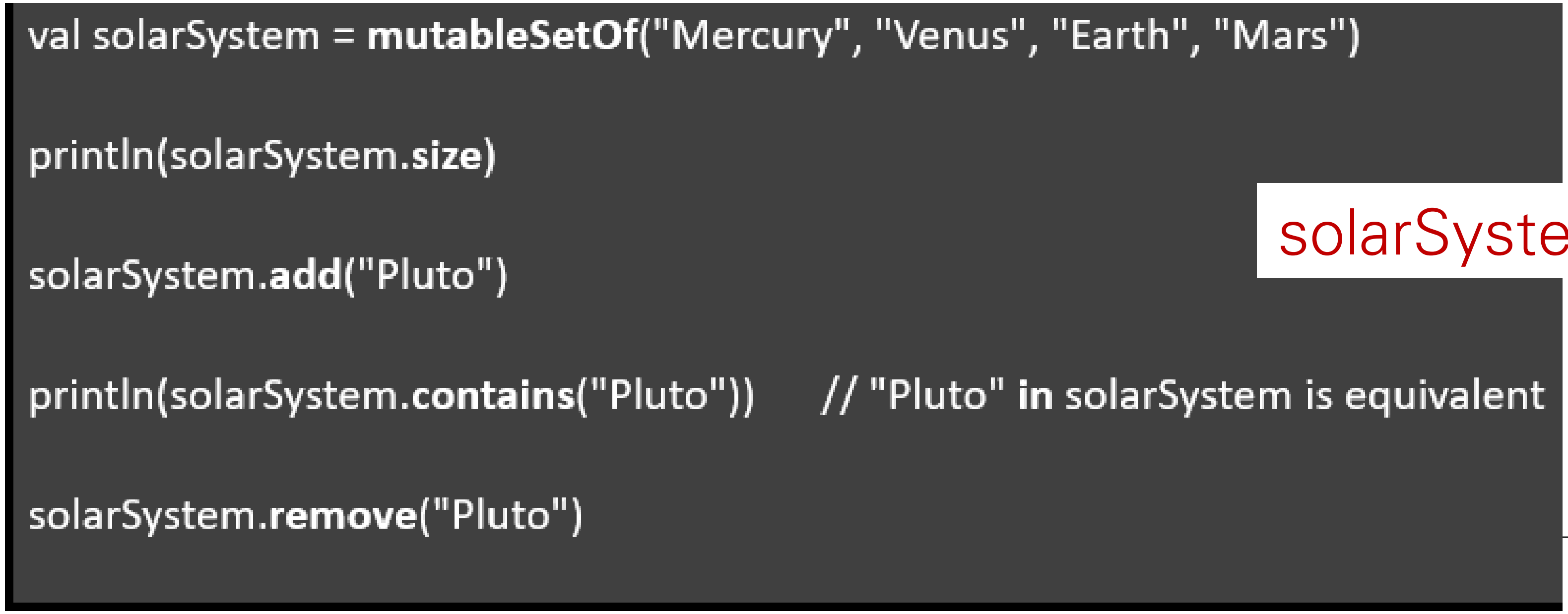
DIY

COMPLEX DATA TYPES

Set / MutableSet

Guess what?

- Collection that has **no order and no duplicate values** (due to *hash code*)
- Hash code is an *Int* produced by *hashCode()* of any Kotlin class
- A small change to the object results in a vastly different hash code
- **Searching** for a specific element in a set is **faster than in lists**
- But sets tend to **use more memory than lists** for the same amount of data



```
val solarSystem = mutableSetOf("Mercury", "Venus", "Earth", "Mars")

println(solarSystem.size)

solarSystem.add("Pluto")

println(solarSystem.contains("Pluto"))     // "Pluto" in solarSystem is equivalent

solarSystem.remove("Pluto")
```

solarSystem.**removeAt**(2) ???

COMPLEX DATA TYPES

Map / MutableMap

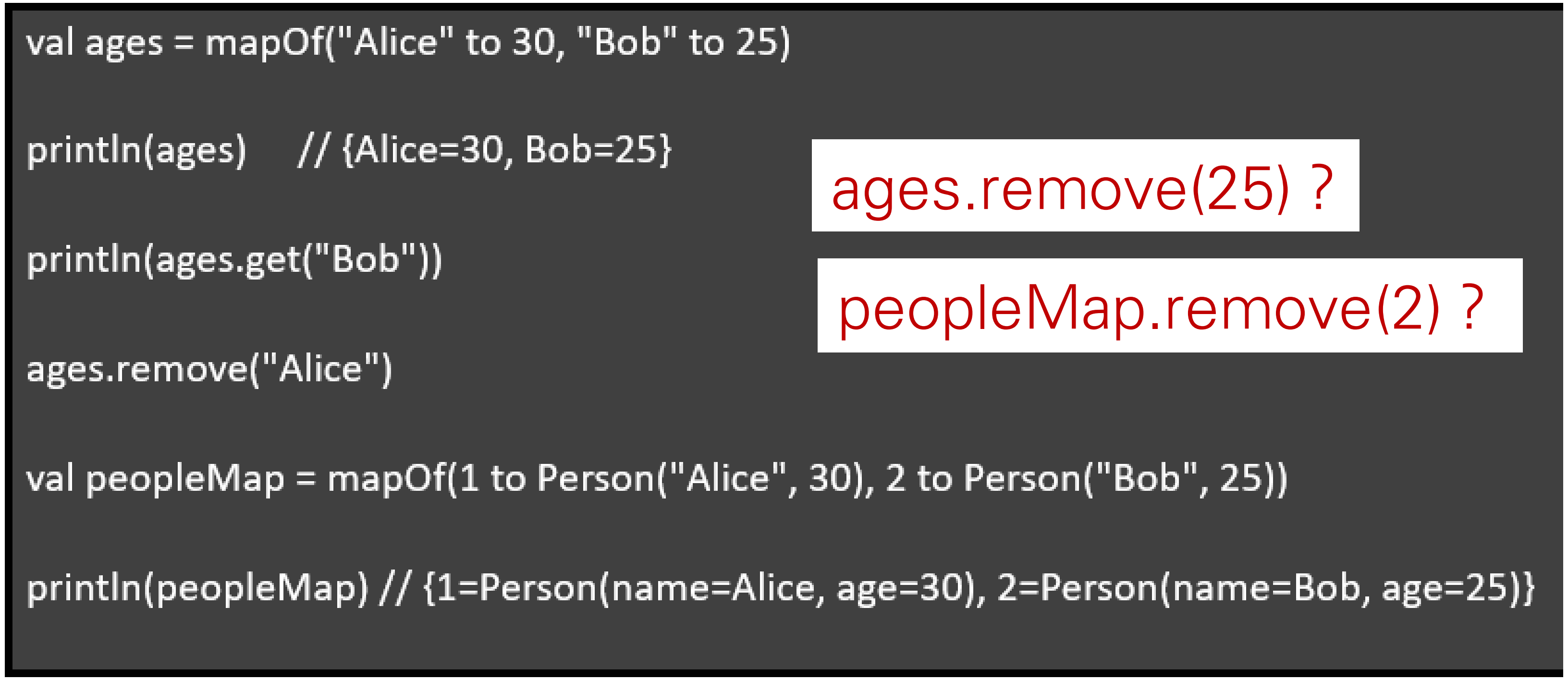
Guess what?

Maps **associate keys with values**, allowing you to create complex data structures to represent relationships or configurations

DIY

```
mutableMapOf< key type , value type >()
```

```
val map name = mapOf(
    key to value ,
    key to value ,
    key to value ,
)
```



```
val ages = mapOf("Alice" to 30, "Bob" to 25)

println(ages)     // {Alice=30, Bob=25}

println(ages.get("Bob"))

ages.remove("Alice")

val peopleMap = mapOf(1 to Person("Alice", 30), 2 to Person("Bob", 25))

println(peopleMap) // {1=Person(name=Alice, age=30), 2=Person(name=Bob, age=25)}
```

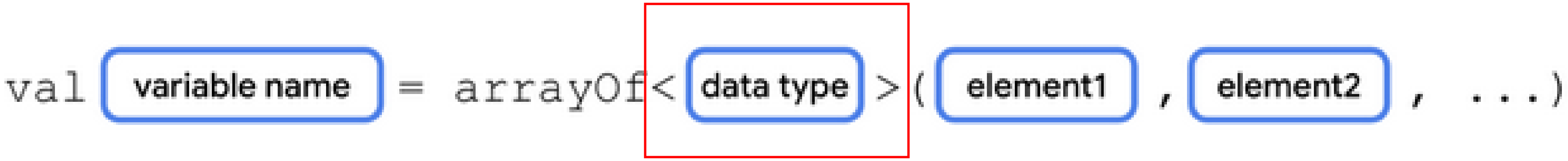
ages.remove(25) ?

peopleMap.remove(2) ?

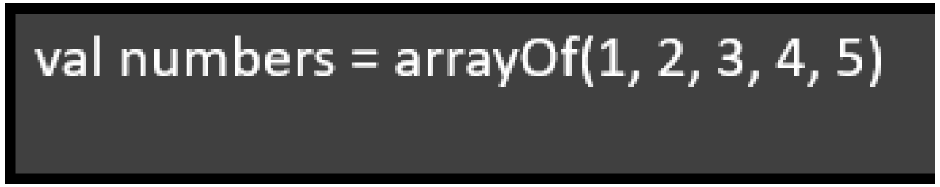
COMPLEX DATA TYPES

Arrays

- Arrays are fixed-size collections that can store elements **of the same type**
- The data type is optional as it can be inferred



```
val variable name = arrayOf< data type >( element1 , element2 , ...)
```

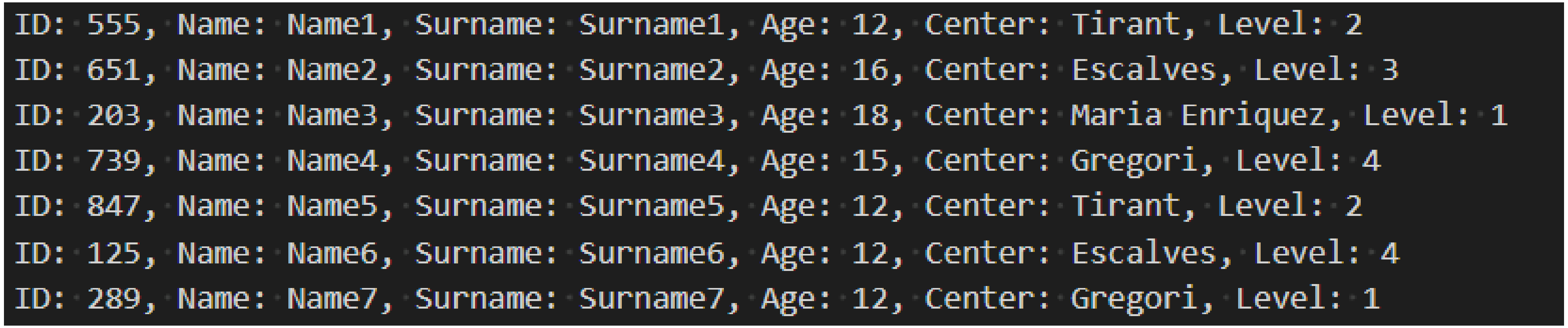


```
val numbers = arrayOf(1, 2, 3, 4, 5)
```

COMPLEX DATA TYPES

Exercise – Part 1

- Create a collection of Students and add 20 students
- Give incremental name and surnames. Age, level and ID must be random (between numbers that make sense) and center must be a random HighSchool object among 5 predefined ones
- Show the resulting Students

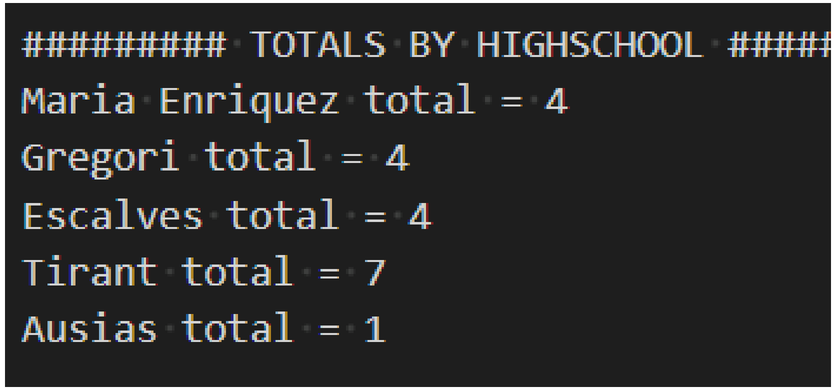


```
ID: 555, Name: Name1, Surname: Surname1, Age: 12, Center: Tirant, Level: 2
ID: 651, Name: Name2, Surname: Surname2, Age: 16, Center: Escalves, Level: 3
ID: 203, Name: Name3, Surname: Surname3, Age: 18, Center: Maria Enriquez, Level: 1
ID: 739, Name: Name4, Surname: Surname4, Age: 15, Center: Gregori, Level: 4
ID: 847, Name: Name5, Surname: Surname5, Age: 12, Center: Tirant, Level: 2
ID: 125, Name: Name6, Surname: Surname6, Age: 12, Center: Escalves, Level: 4
ID: 289, Name: Name7, Surname: Surname7, Age: 12, Center: Gregori, Level: 1
```

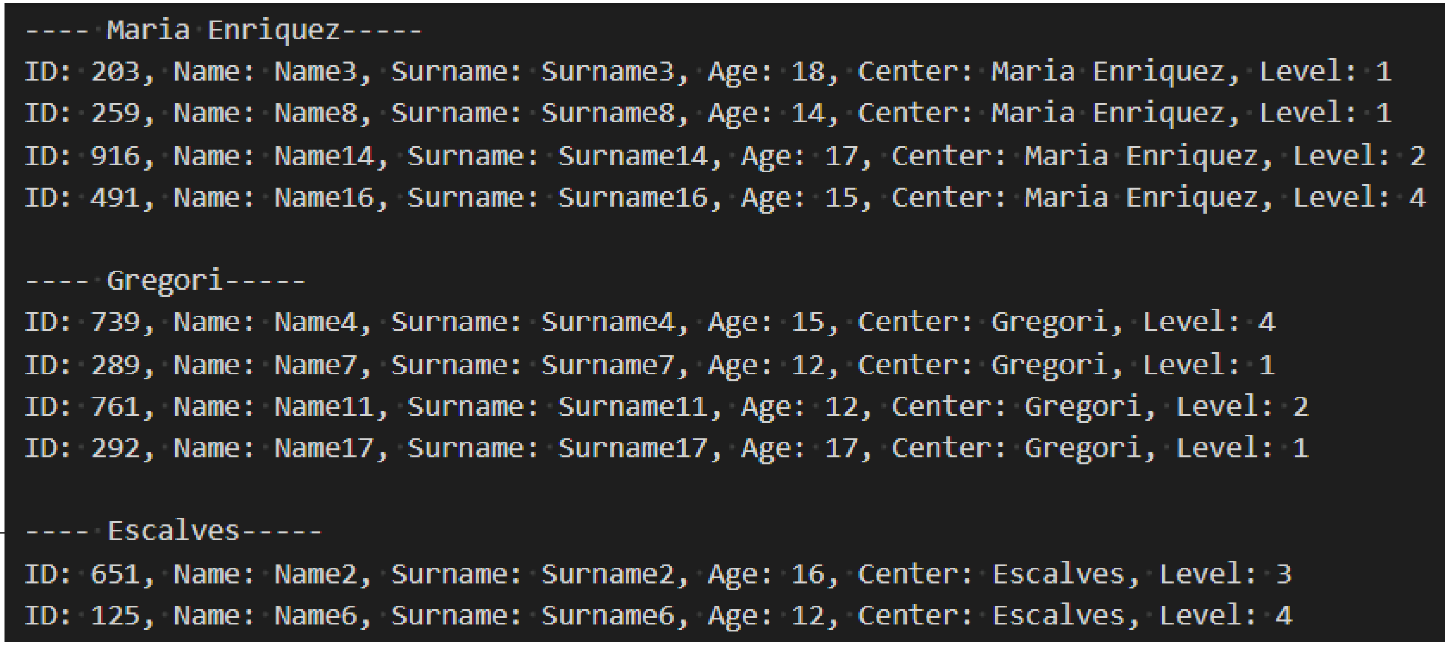
COMPLEX DATA TYPES

Exercise – Part 2

- Surf the collection and create a Map with
  - **Keys** = High Schools names
  - **Value** = collection of students
- Then print the total number of students per High School
- Print the detail of each student per High School



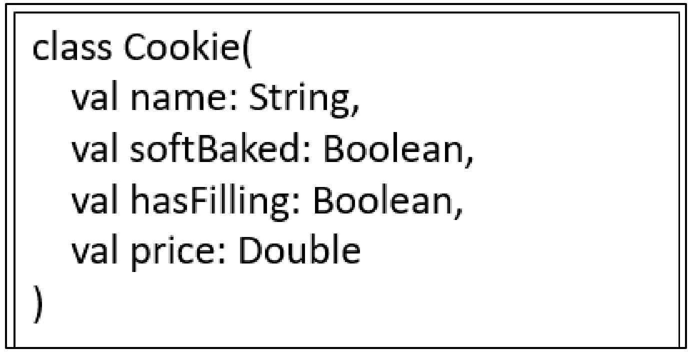
```
########## TOTALS BY HIGHSCHOOL #####
Maria Enriquez total = 4
Gregori total = 4
Escalves total = 4
Tirant total = 7
Ausias total = 1
```



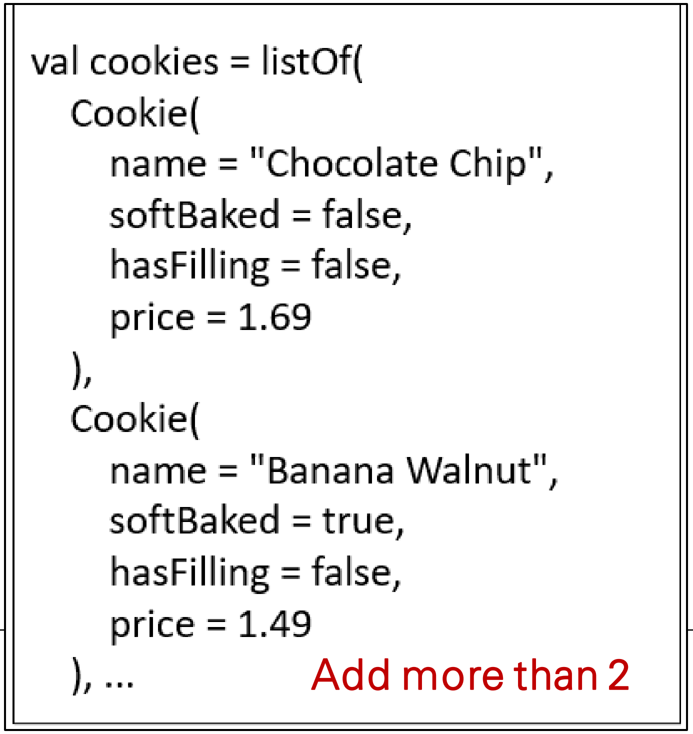
```
---- Maria Enriquez-----
ID: 203, Name: Name3, Surname: Surname3, Age: 18, Center: Maria Enriquez, Level: 1
ID: 259, Name: Name8, Surname: Surname8, Age: 14, Center: Maria Enriquez, Level: 1
ID: 916, Name: Name14, Surname: Surname14, Age: 17, Center: Maria Enriquez, Level: 2
ID: 491, Name: Name16, Surname: Surname16, Age: 15, Center: Maria Enriquez, Level: 4

---- Gregori-----
ID: 739, Name: Name4, Surname: Surname4, Age: 15, Center: Gregori, Level: 4
ID: 289, Name: Name7, Surname: Surname7, Age: 12, Center: Gregori, Level: 1
ID: 761, Name: Name11, Surname: Surname11, Age: 12, Center: Gregori, Level: 2
ID: 292, Name: Name17, Surname: Surname17, Age: 17, Center: Gregori, Level: 1

---- Escalves-----
ID: 651, Name: Name2, Surname: Surname2, Age: 16, Center: Escalves, Level: 3
ID: 125, Name: Name6, Surname: Surname6, Age: 12, Center: Escalves, Level: 4
```

High-order Functions – forEach()

- A higher-order function is a function that takes functions as parameters or returns a function
- *forEach()* can be combined with string templates and lambdas to iterate along a collection and perform actions on each element

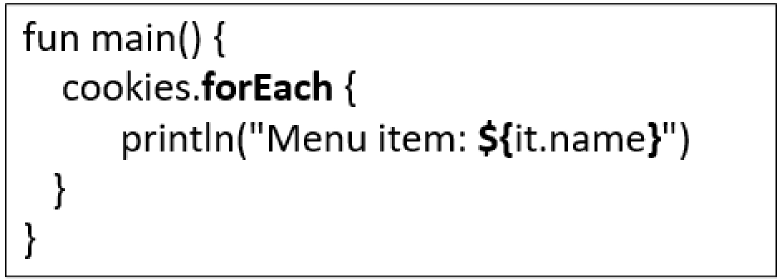


```
class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)
```



```
val cookies = listOf(
    Cookie(
        name = "Chocolate Chip",
        softBaked = false,
        hasFilling = false,
        price = 1.69
    ),
    Cookie(
        name = "Banana Walnut",
        softBaked = true,
        hasFilling = false,
        price = 1.49
    ), ...
```

Add more than 2

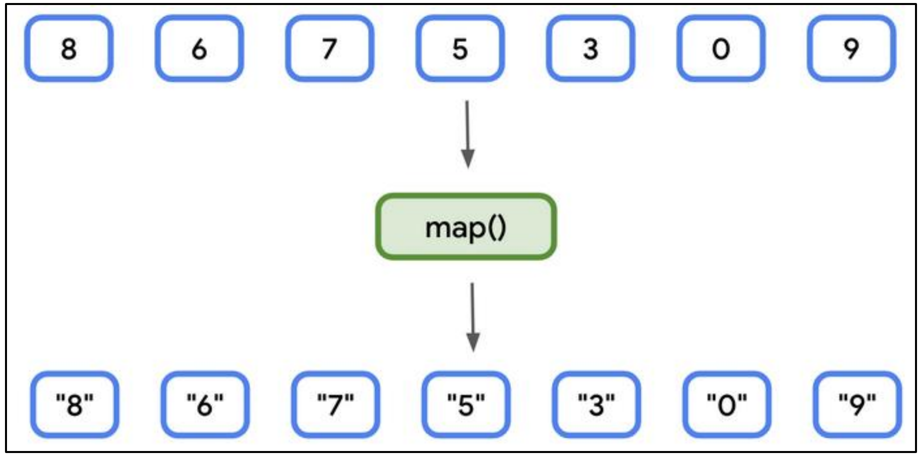


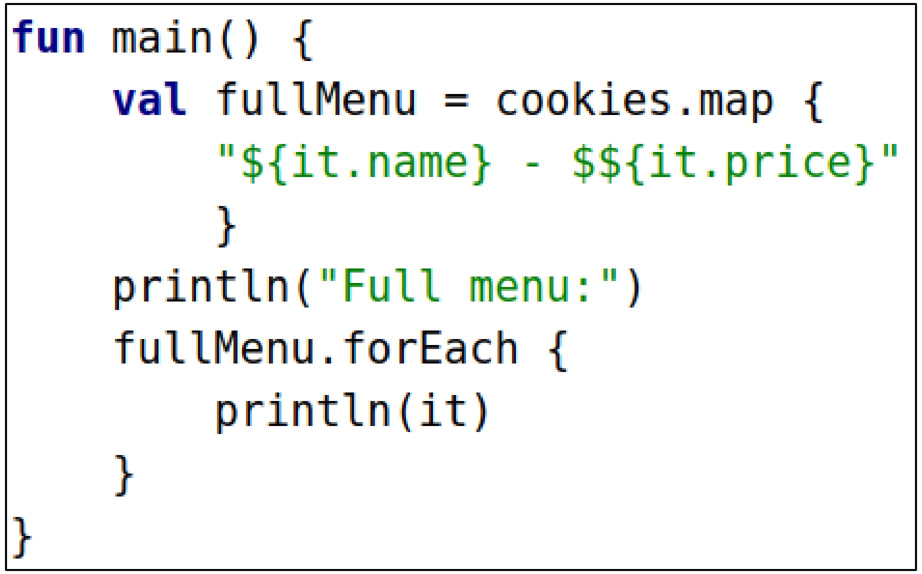
```
fun main() {
    cookies.forEach {
          println("Menu item: ${it.name}")
    }
}
```

COMPLEX DATA TYPES

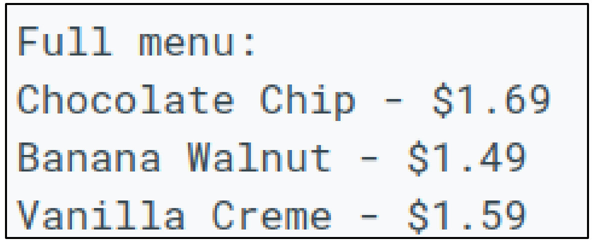
High-order Functions – map()

Lets you transform a collection into a new collection with the same number of elements while adding some transformation





```
fun main() {
    val fullMenu = cookies.map {
        "${it.name} - $${it.price}"
        }
    println("Full menu:")
    fullMenu.forEach {
        println(it)
    }
}
```

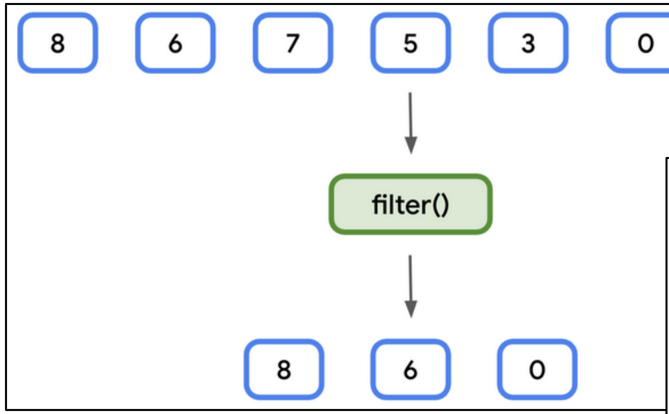


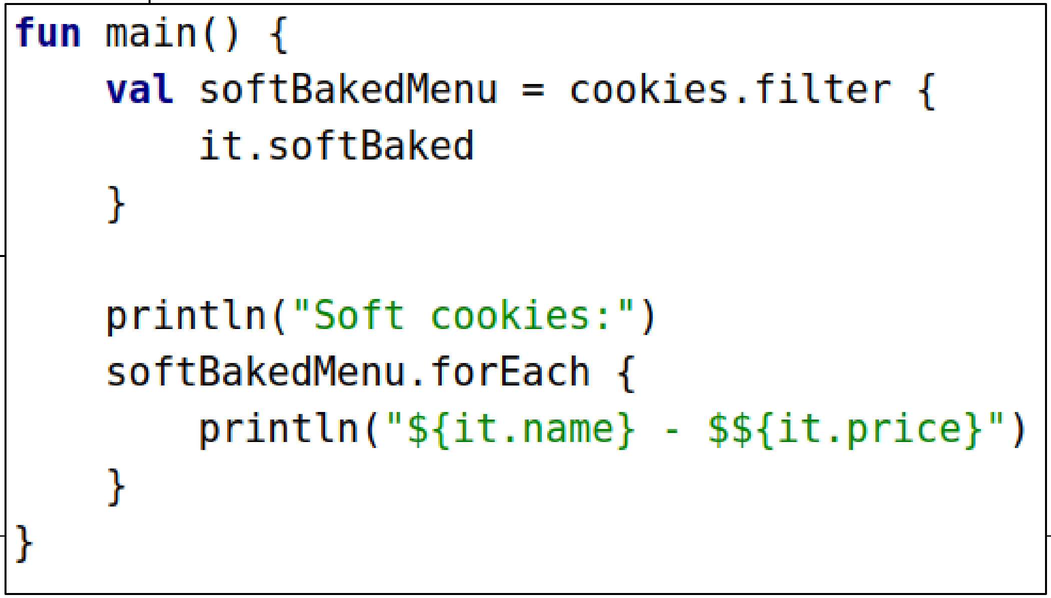
```
Full menu:
Chocolate Chip - $1.69
Banana Walnut - $1.49
Vanilla Creme - $1.59
```

COMPLEX DATA TYPES

High-order Functions – filter()

- Lets you create a subset of a collection
- The lambda has a single parameter representing each item in the collection and returns a Boolean value





```
fun main() {
    val softBakedMenu = cookies.filter {
        it.softBaked
    }

    println("Soft cookies:")
    softBakedMenu.forEach {
        println("${it.name} - $${it.price}")
    }
}
```

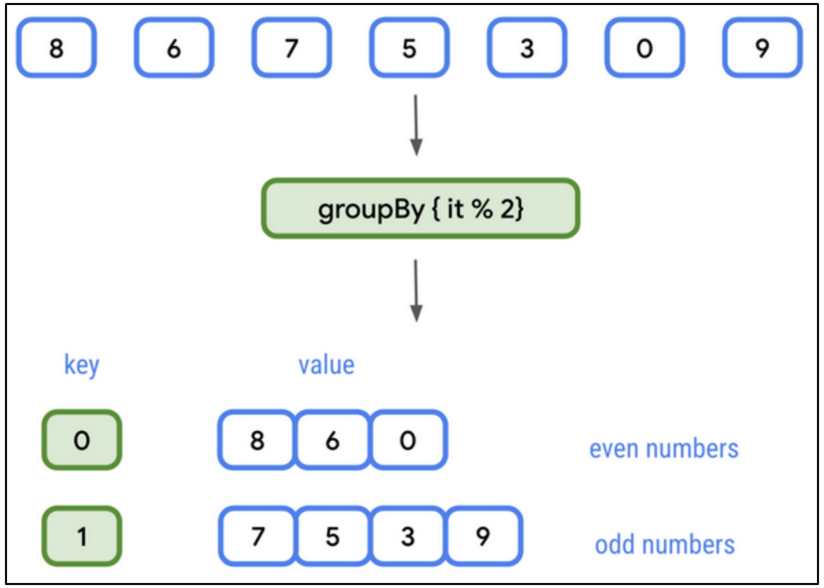
If the result of the lambda expression is true, the item is included

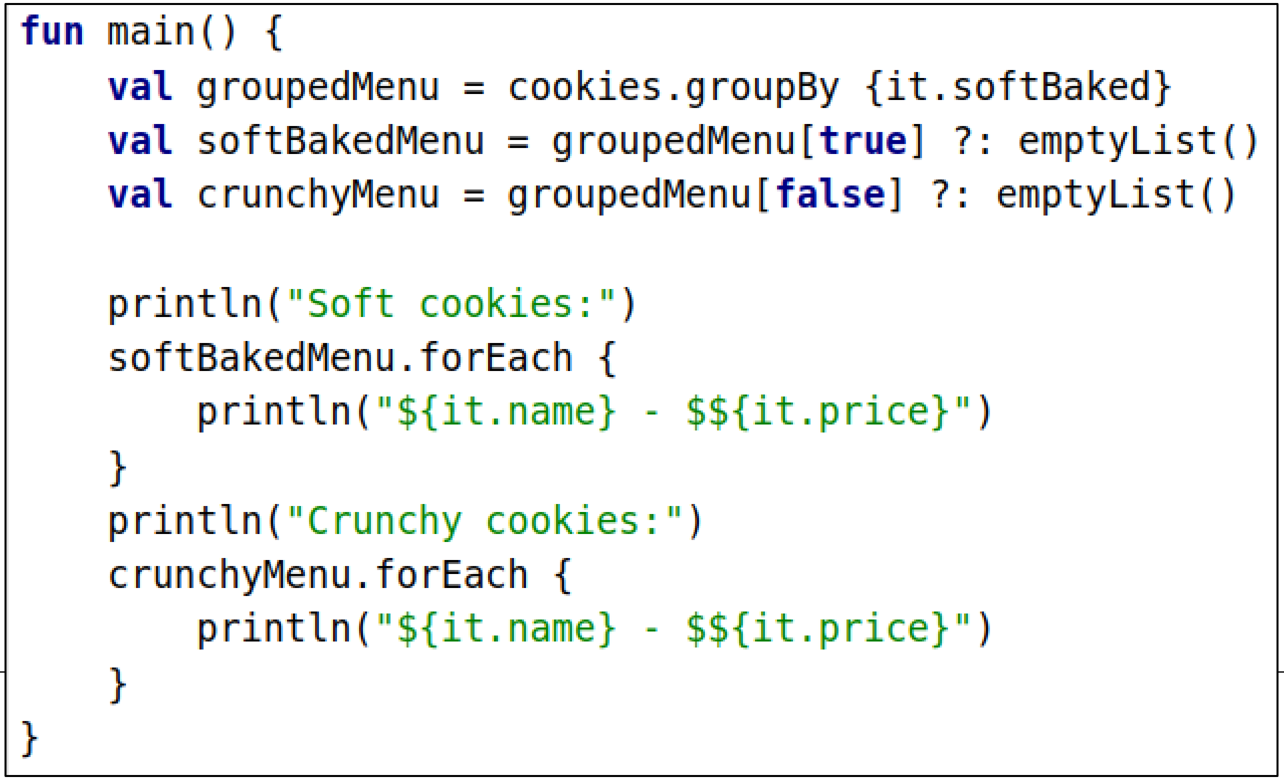
If the result is false, it is not

High-order Functions – groupBy()

List
groupBy()
Map

- Used to turn a list into a map, based on a function
- Each **unique return** value of the function **becomes a *key*** in the resulting map
- The *values* for each key are **all the items that produced that unique return** value





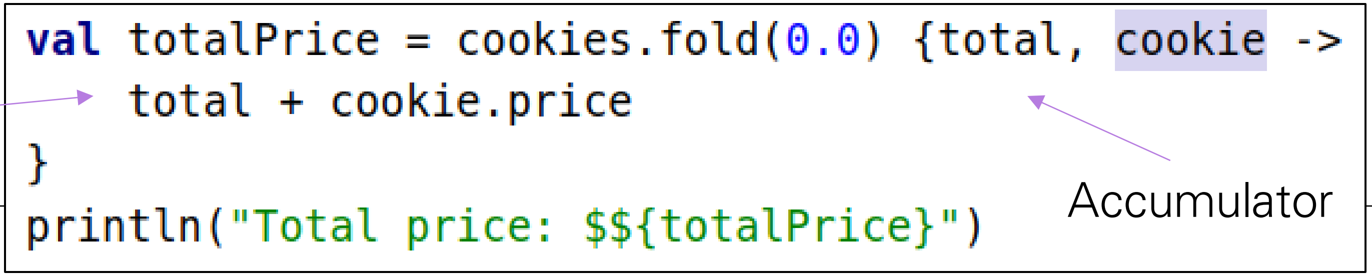
```kotlin
fun main() {
    val groupedMenu = cookies.groupBy {it.softBaked}
    val softBakedMenu = groupedMenu[true] ?: emptyList()
    val crunchyMenu = groupedMenu[false] ?: emptyList()

    println("Soft cookies:")
    softBakedMenu.forEach {
        println("${it.name} - $${it.price}")
    }
    println("Crunchy cookies:")
    crunchyMenu.forEach {
        println("${it.name} - $${it.price}")
    }
}
```

COMPLEX DATA TYPES

High-order Functions – fold()

- Used to generate **a single value from a collection**
- The fold() function takes two parameters:
  - An initial value
  - A lambda expression that returns a value with the same type as the initial value
- The lambda expression additionally has two parameters:
  - **Accumulator**: Each time the lambda expression is called, the accumulator is equal to the return value from the previous time the lambda was called
  - The second is the **same type as each element** in the collection

Initial value of accumulator



```
val totalPrice = cookies.fold(0.0) {total, cookie ->
    total + cookie.price
}
println("Total price: $${totalPrice}")
```

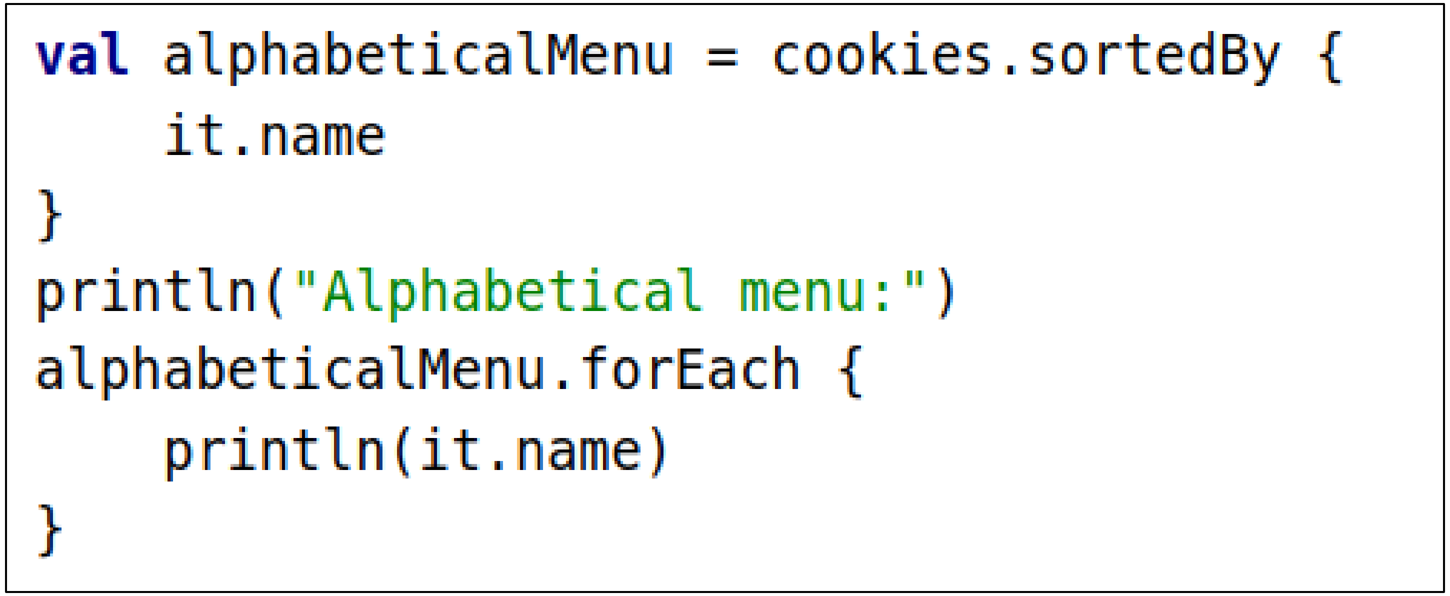
total = total + cookie.price
return total

Accumulator

COMPLEX DATA TYPES

High-order Functions – sortedBy()

- Lets you specify a lambda that returns the property you'd like to sort by
- As far as the data type has a natural sort order, it will be sorted just like a collection of that type

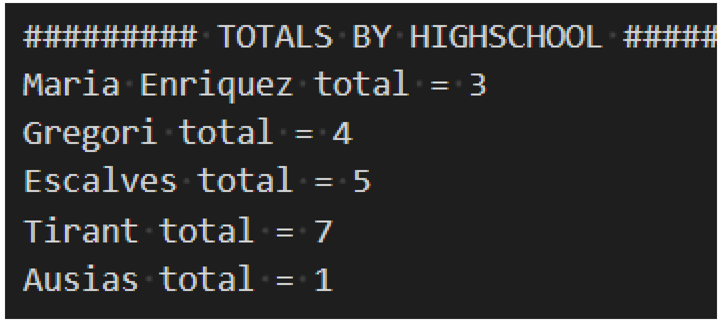


```
val alphabeticalMenu = cookies.sortedBy {
    it.name
}
println("Alphabetical menu:")
alphabeticalMenu.forEach {
    println(it.name)
}
```

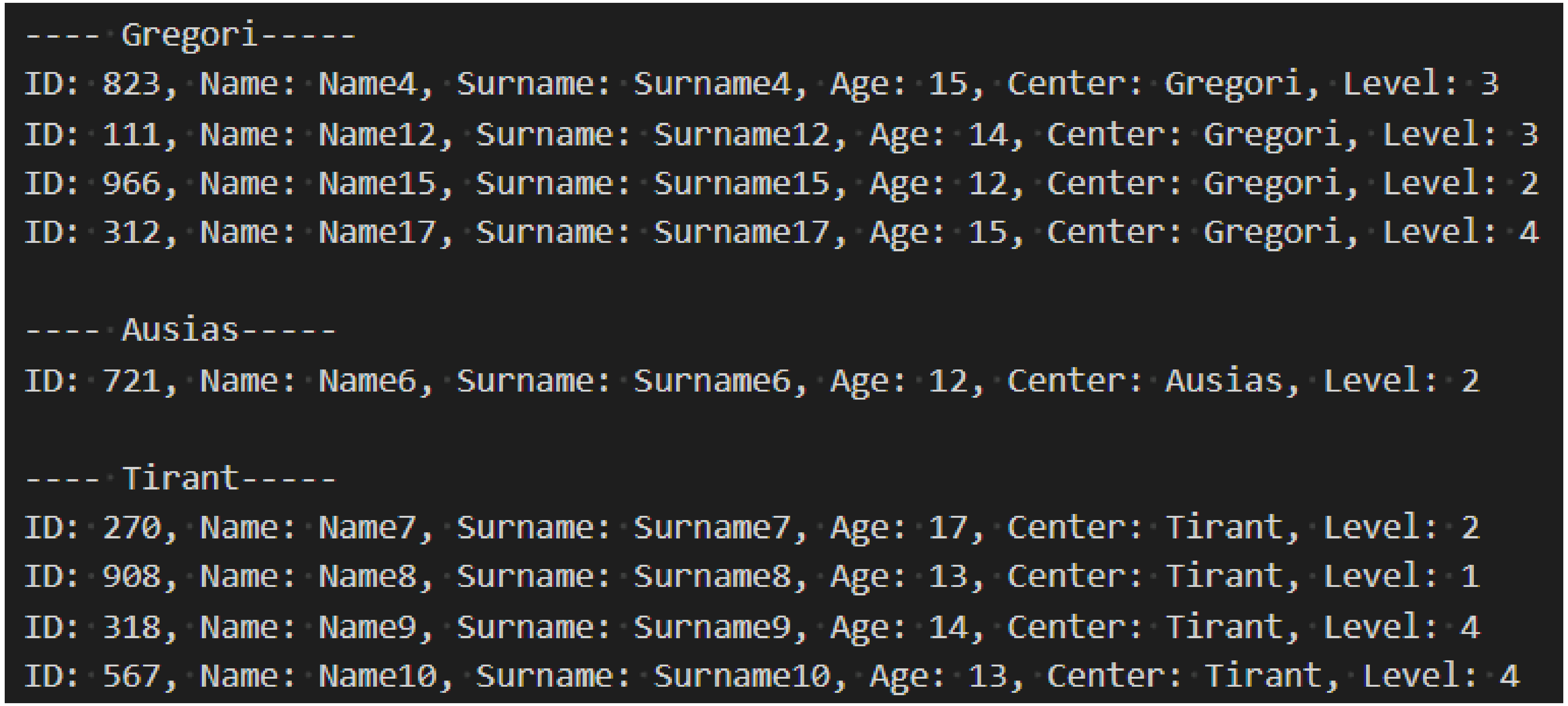
COMPLEX DATA TYPES

Exercise

- Improve the previous program (Collections) by using High-Order functions
- Results must be the same

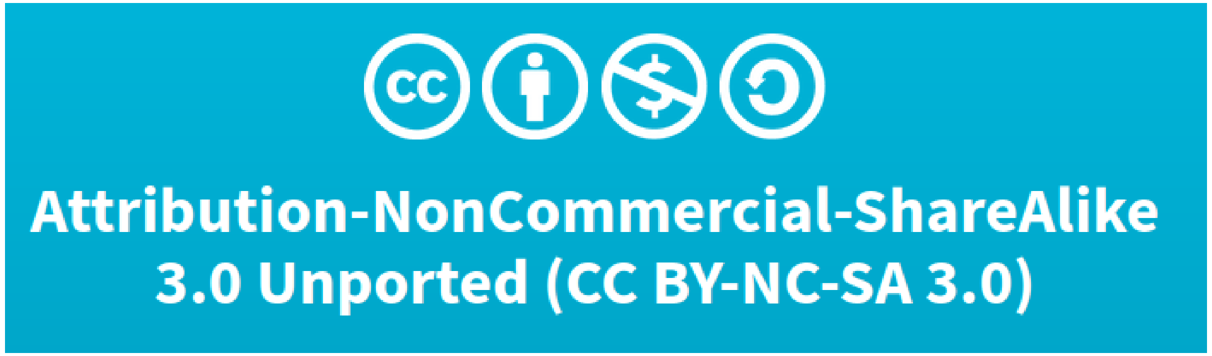


```
########## TOTALS BY HIGHSCHOOL #####
Maria Enriquez total = 3
Gregori total = 4
Escalves total = 5
Tirant total = 7
Ausias total = 1
```

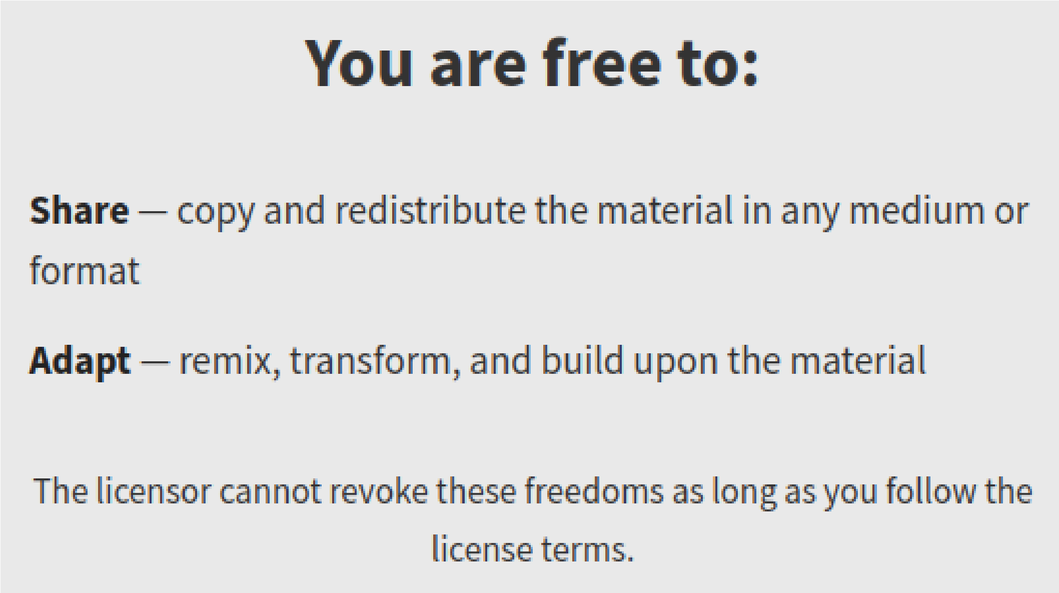


```
---- Gregori-----
ID: 823, Name: Name4, Surname: Surname4, Age: 15, Center: Gregori, Level: 3
ID: 111, Name: Name12, Surname: Surname12, Age: 14, Center: Gregori, Level: 3
ID: 966, Name: Name15, Surname: Surname15, Age: 12, Center: Gregori, Level: 2
ID: 312, Name: Name17, Surname: Surname17, Age: 15, Center: Gregori, Level: 4

---- Ausias-----
ID: 721, Name: Name6, Surname: Surname6, Age: 12, Center: Ausias, Level: 2

---- Tirant-----
ID: 270, Name: Name7, Surname: Surname7, Age: 17, Center: Tirant, Level: 2
ID: 908, Name: Name8, Surname: Surname8, Age: 13, Center: Tirant, Level: 1
ID: 318, Name: Name9, Surname: Surname9, Age: 14, Center: Tirant, Level: 4
ID: 567, Name: Name10, Surname: Surname10, Age: 13, Center: Tirant, Level: 4
```

LICENSE



Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0)

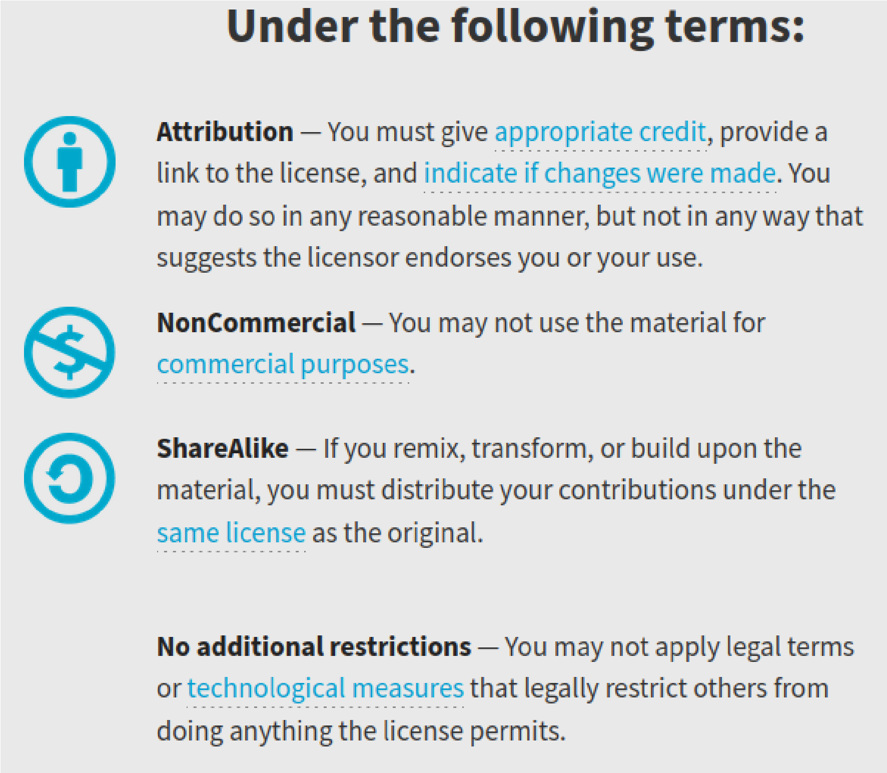


You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.



Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

https://creativecommons.org/licenses/by-nc-sa/3.0/