Programming Fundamentals in Kotlin

Notes by José A. Espiño P. [1]

Spring Semester 2023–2024

[1] The content in these notes is sourced from what was covered in the course the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Introduction to Kotlin

Kotlin is the main language Android development is done in nowadays. It was introduced in 2016 as an improvement on Java, previously the dominant language in Android mobile development. In order to make the transition from one to the other as seamless as possible, a core aspect of Kotlin is its backwards compatibility with Java. This means that Kotlin can run on the Java Virtual Machine (JVM) and can be used in conjunction with Java in the same project — and by transitivity in conjunction with all the amazing Java libraries available already. You can use Kotlin in either your local IDE of choice (IntelliJ being the most popular one for Kotlin) or in Kotlin Playground, a web–based IDE for Kotlin.
A program is a set of instructions that a computer can execute. These instructions are written in a programming language, which is a set of rules and symbols that we use to write code. The code is then translated into machine code, which is the language that the computer understands.
In Kotlin, values are any data that can be stored in a variable, which in turn is a container for data values. These data values are grouped according to their type, such as integers, strings, and booleans. The type of a variable is determined by the data it contains. Some of the most common types of variables are:

- Integers: whole numbers, such as 1, 2, 3, and so on.
- Doubles: numbers with a decimal point, such as 1.0, 2.0, 3.0, and so on.
- Strings: sequences of characters, such as "Hello, world!". They are defined using double quotes. You can define a multiline string using triple quotes.
- Char: a single character, such as 'a' or 'b'. It is defined using single quotes.
- Booleans: true or false values.

In Kotlin, there are two special keywords used to declare variables: `val` and `var`. The `val` keyword is used to declare a read–only variable, which means that its value cannot be changed after it has been assigned. `var`, on the other hand, declares a mutable variable. As opposed to Java, Kotlin does not require explicit type declaration for variables. The type of a variable is inferred from the value it is assigned. Once declared, however, the type of a variable cannot be changed. For example,

```kotlin
val name = "Alice"
var age = 30
//Explicit type declaration, Java-style is also valid.
val name: String = "Alice"
var age: Int = 30
//Making a variable of type Any makes it able to store any type of data.
var anyType: Any = "Hello"
anyType = 1
```

As you might have noticed, there is another difference between Kotlin and Java: the absence of semicolons at the end of each line. This is because Kotlin is designed to be more concise and readable than Java. Kotlin accepts semicolons, but they are optional.
There are four main ways to represent numbers in Kotlin:

- Int: 123
- Long: 123L (accommodates larger integer values)
- Double: 123.5 (default type for floating–point numbers)
- Float: 123.5F (accommodates smaller floating–point numbers)

You can add underscores to numbers to make them more readable. For example, 1_000_000 is the same as 1000000.
The main operators used in Kotlin are +, −, ∗, /, and % (modulus). The modulus operator returns the remainder of a division. For example, 5 % 2 returns 1. Something worth noting is that you can convert between number types through the functions `toInt()`, `toLong()`, `toFloat()`, and so on:

```kotlin
val x: Int = 123
val y: Long = x.toLong()
```

By default, the result of the division of two integers will be an integer. If you want the result to contain the decimal part, transform one of the values to a floating–point number beforehand.
Prefix and postfix operations are also available in Kotlin. For example, `++x` and `x++` are both valid. The difference between the two is that the first one increments the value of `x` before using it, while the second one increments it after using it.
Some useful string methods and functions are:

- `length`: returns the length of the string.
- `toUpperCase()`: returns the string in uppercase.
- `toLowerCase()`: returns the string in lowercase.
- `substring()`: returns a part of the string.
- `replace()`: replaces a part of the string with another string.
- `contains()`: returns true if the string contains a given substring.
- `startsWith()`: returns true if the string starts with a given prefix.

- `endsWith()`: returns true if the string ends with a given suffix.
- `println()`: prints the string to the console and adds a new line.
- `print()`: prints the string to the console.
- `code()`: returns the Unicode value of the first character in the string.
- `codeAt()`: returns the Unicode value of the character at a given index.
- `equals()`: returns true if the string is equal to another string. Unlike in Java, in Kotlin you can use the == operator to compare strings.
- + operator: concatenates two strings into a new string.

It is important to mention the dollar sign operator ($), which is used to insert a variable into a string. For example,

```
val name = "Alice"
val num = 10
println("Hello, $name!")
println("This is a little more complex: ${num + 5} is the result of 10 +
    5.")
```

Strings can be indexed like arrays, using square brackets, and you can use the `get()` method to access a character at a given index.
The logical operators in Kotlin are `!` (not), `&&` (and), and `||` (or). The `!` operator negates a boolean value, while the other two operators return a boolean value.
The format for an if–else statement in Kotlin is

```
if (condition) {
    // code to execute if the condition is true
} else if (condition) {
    // code to execute if the first condition is false and this one is
    true
} else {
    // code to execute if all the conditions are false
}
//if you only have one line of code to execute, you can omit the curly
    braces
if (condition) code
else code
//Conditional statements can also be used to define variables:
val im_hungry = true
val eat: String = if (im_hungry) "I'm going to eat" else "I'm not going
    to eat"
```

If–else if–else statements are not as common in Kotlin. This is because `when` statements are a more powerful way of achieving the same result. The format for a `when` statement is

```
when {
    condition1 -> {
        // code to execute if the condition is value1
    }
```

```
    condition2 -> {
        // code to execute if the condition is value2
    }
    else -> {
        // code to execute if the condition is none of the above
    }
}
//You can also use when to define variables:
val x = 10
val y = when (x) {
    1 -> "x is 1"
    2 -> "x is 2"
    else -> "x is neither 1 nor 2"
}
//Lastly, when can also be used to compare a variable to different
    possible values:
val x: Int = scanner.nextInt()
when (x) {
    in 1..9 -> println("x is between 1 and 9")
    10 -> println("x is 10")
    else -> println("x is not in the range of 1 to 10")
}
```

Note that in Kotlin, range checks are performed differently to Java. In Kotlin, you can use the `in` operator to check if a value is within a range, specified by two numbers separated by two dots. For instance,

```
val x = 10
if (x in 1..9) {
    println("x is between 1 and 9")
}
```

Loops are used to execute a block of code multiple times, either based on a condition or iterating through a series of values. The syntax for `while` loops is as follows:

```
while (condition) {
    // code to execute while the condition is true
}
//For example:
fun main() {
    var i = 1
    while (i * i <= 100) {
    println(i * i)
        i++
         // or i += 1
         // or i = i + 1
    }
}
```

For–loops are a common way to iterate through the elements of a list. Its syntax in Kotlin is:

```
for (element in collection) {
    // code to execute for each element in the collection
}
//For example:
```

```
fun main() {
    val names = listOf("Alice", "Bob", "Charlie")
    for (name in names) {
        println(name)
    }
//or can iterate through a range
for (i in 1..10) {
    println(i)
}
//this is a range from 1 to 10, inclusive
for (i in 1 until 10) {
    println(i)
}
//this is an open range, from 1 to 9
for (i in 10 downTo 1) {
    println(i)
}
//this is a range from 10 to 1, inclusive
for (i in 10 downTo 1 step 2) {
    println(i)
}
//this is a range from 10 to 1, inclusive, with a step of 2
}
```

Nested loops work as you expect:

```
//The following code prints a triangle made of the character "A"
fun main() {
    for (i in 1..5) {
        val numberOfSpaces = 5 - i
        for (j in 1..numberOfSpaces) {
            print(" ")
        }
        val numberOfStars = i * 2 - 1
        for (j in 1..numberOfStars) {
            print("*")
        }
        println()
    }
}
```

2 Functions, classes, and objects

Functions are sets of instructions that you execute to accomplish a specific task. They usually have certain data as an input, output certain data, and can be **called** within a program — although these first two features are not a must. In Kotlin, the `main` function has a special role: it will always be the starting point of execution for your program.
The way you define a function is somewhat different to the way you do in Java:

```
    fun selfPresentation(name: String, age: Int) {
        println("Hello, my name is $name and I'm $age years old.")
    }
```

```
    //You can also explicitly state the return type, but it is not
    mandatory
    fun sum(a: Int, b: Int): Int {
        return a + b
    }
    //You can add default values for the parameters of a function
    fun greet(name: String = "Alice") {
        println("Hello, $name!")
    }
    //If the user does not provide a value for the parameter, the default
     value will be used

    //Single expression functions have a special syntax
    fun triangleArea(width: Double, height: Double): Double = width *
    height / 2
```

In general, the scope of a variable in Kotlin is based on the block enclosed by curly braces where it is declared. If a variable is declared within a certain block, it is only accessible in it and in any other sub–block defined within the same block. If it is declared outside of any block, it is accessible throughout the entire program. This kind of variable is known as a **global variable** or a **top–level variable**.

A **recursive function** is one that calls itself. They can be inefficient because of their large memory consumption, but there are cases in which their readability makes them desirable. When defining such a function, you must keep in mind two essential conditions: the base case and the recursive case. The base case is the condition that will stop the recursion, while the recursive case is the one that will keep the recursion going (and getting closer to the base case on each iteration). For example, the factorial of a number is defined as the product of all the positive integers up to that number. The base case for the factorial function is when the input is 0, and the recursive case is when the input is greater than 0. The factorial function can be defined as follows:

```
fun factorial(n: Int): Int {
    return if (n == 0) 1 else n * factorial(n - 1)
}
```

In Kotlin, every value is an object. Using classes, you can define your own type of object: a class is a blueprint of how objects should be created and wich methods or properties it consists of. An object is just a specific instance of a class. You can create your own classes using the keyword `class`:

```
class Dog (
    val name: String, //read only
    var age: Double
) {
    isCute = true
    fun bark() {
        println("Woof!")
    }
}
```

If you want you instantiate an object of this class, you need to call a **constructor**, a special function that creates the object. In Kotlin, the primary constructor is defined in the class

header, and the secondary constructors are defined in the class body. Any values that you add in the secondary constructor must have a default value. Instantiation is simple, for example,

```
    val Hiko: Dog = Dog("Hiko", 3)
    // or
    val Hiko = Dog("Hiko", 3)
    Hiko.bark()
    Hiko.age = 3.5
```

Methods are functions that are specific to a certain class and are defined in the class body (between the curly braces). Methods have direct access to every property of an object of the class they belong to, and there must be an object instantiated for you to be able to access the method. This is because when you call a method, the object of their class is passed to their body, a **receiver object**. It can then be accessed using the `this` keyword, the **receiver reference**. For example,

```
class Dog(val name: String) {
    var hunger = 10
    fun feed() {
        println("Feeding ${this.name}")
        this.hunger -= 1
    }
}
fun main(){
    val dog = Dog("Hiko")
    dog.feed()
}
//note that often, such as in this example, the use of the this keyword
    is optional
```

As you know, everything in Kotlin is an object. Thus, the language is efficient in the implementation of the **object–oriented programming** (OOP) paradigm. OOP is portable and reusable, using layers of abstraction that allow for code to be understandable and easily reusable. The three main components of this paradigm are classes, objects, and methods. Classes are the blueprints for objects, and objects are instances of classes. Methods are functions that are specific to a certain class. That being said, the four main principles of OOP are:

1. **Inheritance**: Creating a class that is based on another class. The new class inherits the properties and methods of the class it is based. The original class is known as the **superclass** or **parent class**, and the new class is known as the **subclass** or **child class**. The subclass can override the methods of the superclass, and it can also have its own methods and properties.

2. **Polymorphism**: The ability of a method to do different things based on the object it is called on. This is achieved through method overriding, which is when a subclass provides a specific implementation of a method that is already defined in the superclass. An example is the built–in + operator, which can be used to add numbers and to concatenate strings.

3. **Encapsulation**: The process of hiding the internal workings of an object from the outside world. This is achieved by making the properties of an object private, so that they can only be accessed and modified by the methods of the object. This is done to prevent the object from being modified in a way that would make it invalid. Encapsulation is generally achieved using access modifiers, such as `private` and `protected`.

4. **Abstraction**: The process of simplifying complex systems by hiding unnecessary details. This is achieved by defining a class that contains only the essential properties and methods of an object, and by hiding the implementation details of the class. This is done to make the class easier to understand and to use.

As explained earlier, **polymorphism** is the possibility to specify a method that accepts objects produced using different classes. This is achieved through method overriding, which is when a subclass provides a specific implementation of a method that is already defined in the superclass. To have an object that can take many forms, you need to specify what the object can do in an **interface**, a collection of abstract methods and properties that can be implemented by a class. The properties held by interfaces should have no values, since they are just blueprints for the properties of the classes that implement them. For example,

```
interface Animal {
    val species: String
    fun makeSound()
    val humanFriendly: Boolean
}
//let us implement the interface
class Pig: Animal(
    val name: String,
    override val species: String = "Pig",
    override val humanFriendly: Boolean = true
)
{
    override fun makeSound() {
        println("Oink!")
    }
}
//another one
class Dog: Animal(
    val name: String,
    override val species: String = "Dog",
    override val humanFriendly: Boolean = true
)
{
    override fun makeSound() {
        println("Woof!")
    }
}
//if there is a child class that does not implement the interface fully, it must be declared as abstract
abstract class Fish: Animal(
    val name: String,
    override val species: String = "Fish",
```

```
        override val humanFriendly: Boolean = false
    )
    {
        override fun makeSound() {
        }
    }

    //when you define a function, you can specify that it accepts objects
     of a certain interface
    fun makeSound(animal: Animal) {
        animal.makeSound()
    }
    //instantiations:
    fun main(){
        val pig = Pig("Simba")
        val dog = Dog("Hiko")
        makeSound(pig)
        makeSound(dog)
        val fish = Fish("Nemo")
        makeSound(fish)
    //alternatively, you can call the method directly on the object
    pig.makeSound()
    dog.makeSound()
    fish.makeSound()
    }
```

Note that you can always use a subtype where a supertype is expected, but not the other way around. This is because the subtype has all the properties and methods of the supertype, but the supertype does not necessarily have all the properties and methods of the subtype.
Inheritance is the process of creating a class that is based on another one. The original one is known as the **superclass** or **parent class**, and the new one is known as the **subclass** or **child class**. The subclass can override the methods of the superclass, and it can also have its own methods and properties. By default, all classes in Kotlin are `final`, which means that they cannot be inherited. If you want a class to be inherited, you must declare it as `open`, as well as all the specific methods you want it to be able to override. The difference between a superclass and an interface is that a superclass can have a body, while an interface cannot. For example,

```
    open class house(val color: String) {
        open fun openDoor() {
            println("The door is open")
        }
    }
    class mansion(color: String, val size: Int): house(color) {
        //notice that we are calling the parent class constructor in the
    subclass constructor
        override fun openDoor() {
            println("The door is closed and it is huge")
            println("Let me open it")
            super.openDoor() //super allows you to call the method of the
     superclass
        }
```

```
    }
    fun main(){
        val mansion = mansion("white", 100)
        mansion.openDoor()
    }
```

When you define a class, you need to think of how you want other developers interact with it. For example, if you have a bank account class, you want other developers to be able to deposit and withdraw money from it, but you do not want them to be able to change the balance directly. This is where **encapsulation** comes in. Encapsulation is the process of hiding the internal workings of an object from the outside world. This is achieved through the use of **access modifiers**, which allow you to hide properties and methods so that they can be used inside of a class but not outside of it. The main access modifiers in Kotlin are:

- `public`: the default access modifier, which means that the property or method can be accessed from anywhere.
- `private`: the property or method can only be accessed from within the class it is defined in.
- `protected`: only used in classes that are open or abstract; the property or method can only be accessed from within the class it is defined in and its subclasses.
- `internal`: the property or method can only be accessed from within the module it is defined in.

Abstract classes are a hybrid of an open class and an interface. They can have abstract methods or properties, which have no body (similar to interface definitions) and need to be overriden in subclasses, and concrete methods. Abstract classes cannot be used to create objects, but you can inherit subclasses from them. This is a consequence of the fact that they have abstract methods or properties, which must always be overriden before instantiation. Since each class can only inherit a single class but implement multiple interfaces, interfaces are preferred over abstract classes when you want to define a type that can be used in many different ways. The benefit of an abstract class is that methods can be open or non open (only open in interfaces) and that methods can be either abstract or non–abstract (abstract by default in interfaces). For example,

```
abstract class SomeAbstractClass {
    abstract fun abstractMethod()
    fun callAbstractTwice() {
        abstractMethod() // You can use abstract methods inside the class, because it is assumed they
        // will be overridden in the child class.
        abstractMethod()
    }
}

class SomeRegularClass : SomeAbstractClass {
    override fun abstractMethod() {
        println("Calling abstract method")
```

```
        }
    }

    fun main() {
        val regular = SomeRegularClass()
        regular.abstractMethod() // Calling abstract method
        regular.callAbstractTwice()
        // Calling abstract method
        // Calling abstract method
    }
```

3 Advanced Classes, Nullability, and Collections

3.1 Advanced Classes

By default, every object in a custom class is unique — upon printing the object, you will get a unique identifier, that ignores the properties of the object. FOr the instances in which you want to compare objects based on their properties, you can use the `data` keyword. This will destructure the object into its properties, and will allow you to compare objects based on their properties. For example,

```
    data class Dog(val name: String, val age: Int)
    fun main(){
        val dog1 = Dog("Hiko", 3)
        val dog2 = Dog("Hiko", 3)
        println(dog1 == dog2) //true
        val (name, age) = dog1
        //here, position of variable is essential
    }
```

In order to modify a val value in a data class, you can use the `copy` method. This method creates a new object with the same properties as the original object, but with the specified properties modified. For example,

```
    data class Dog(val name: String, val age: Int)
    fun main(){
        val dog1 = Dog("Hiko", 3)
        val dog2 = dog1.copy(age = 4)
        println(dog1) //Dog(name=Hiko, age=3)
        println(dog2) //Dog(name=Hiko, age=4)
    }
```

The `data` keyword is mostly used for classes that represent a bundle of data. Another useful data class is the `Pair` class, which is used to store two values. This class can be instantiated using the `to` function or the `Pair` constructor. For example,

```
    val pair = 1 to "one"
    val pair2 = Pair(1, "one")
    println(pair == pair2) //true
```

Enum is a special class that represents a possible set of values. It is defined by the `enum` modifier, followed by the `class` keyword and then the class name. In the body of the class, you specify the possible values of the enum. For example,

```
enum class Color {
    RED, GREEN, BLUE
}
fun paint(color: Color) {
    when (color) {
        Color.RED -> println("Painting the wall red")
        Color.GREEN -> println("Painting the wall green")
        Color.BLUE -> println("Painting the wall blue")
    }
}
fun main(){
    val color = Color.RED
    println(color) //RED
    paint(color) //Painting the wall red
}
```

Benefits of enum classes include that they are type–safe, meaning that you can only use the values that are defined in the enum class, and that they are easy to read and understand. Furthermore, they are easy to iterate through. For example,

```
enum class Color {
    RED, GREEN, BLUE
}
fun main(){
    for (color in Color.values()) {
        println(color)
    }
}
```

Enum classes can also have a constructor. For example,

```
enum class PizzaSize(
val sizeInCm: Int
) {
SMALL(15),
MEDIUM(20),
LARGE(25),
EXTRALARGE(30)
}

fun printSize(pizzaSize: PizzaSize) {
    println("$pizzaSize is ${pizzaSize.sizeInCm} cm")
}

fun main() {
    printSize(PizzaSize.MEDIUM) // MEDIUM is 20 cm
    printSize(PizzaSize.EXTRALARGE) // EXTRALARGE is 30 cm
}
```

Previously, we have seen the hierarchy between parent and child classes. The aforementioned parent classes are **open** because you can span an arbitrary amount of child classes from it. When you do not want this to be the case, as it happens with network operations, you can use the `sealed` keyword. Every subclass of such a class needs to be defined within the same package — and there is a finite number of possible subclasses to be defined from it. For

example,

```
sealed class Role
class CeoRole(): Role()
class ManagerRole(val name: String): Role()
class WorkerRole(val name: String): Role()

fun constructLabel(role: Role): String {
    return when (role) {
        is CeoRole -> "The boss"
        is ManagerRole -> "Manager ${role.name}"
        is WorkerRole -> role.name
        }
}

fun main() {
val label = constructLabel(ManagerRole("Leonard"))
println(label) // Manager Leonard
}
```

An exception is an event that occurs during the execution of a program that disrupts the flow of the program. It is usually caused by an error in the program, such as a division by zero or an attempt to access a non–existent file. In Kotlin, exceptions can be handled using the `try-catch` block. The `try` block contains the code that might throw an exception, and the `catch` block contains the code that will be executed if an exception is thrown. Within the catch block, you can specify what type of exception you will be handling. Exceptions are objects, and they all are instances of or of classes that inherit from the `Throwable` class. For example,

```
fun main() {
    val x = 10
    val y = 0
    try {
        val z = x / y
        println(z)
    } catch (e: ArithmeticException) {
        println("An error occurred: ${e.message}")
        //notice that e is a reference to the error
        //you can then use e to access the error or any properties of
 It
        e.printStackTrace()
        //the previous line shows where the error happened
    }
    catch (e: Throwable) {
        println("An non-arithmetic error occurred: ${e.message}")
    }
    //you can use several catch statements.
    //Throwable is the superclass that encompasses all exceptions
}
```

Another block can be added to the `try-catch` block: the `finally` block. This specifies code that will always be invoked, even in the presence of an exception. Look at how this is implemented and a custom exception is defined:

```
class MyException: Throwable("Boo boo this is an error!!")

fun someNastyFunction(){
    throw MyException()
    println("This line will never be executed")
}
fun main(){
    try {
        someNastyFunction()
    } catch (e: MyException) {
        println("An error occurred: ${e.message}")
    }
    finally {
        println("This line will always be executed")
    }
}
```

Some important exceptions defined in Kotlin are:

- `ArithmeticException`: Thrown when an exceptional arithmetic condition has occurred.
- `ArrayIndexOutOfBoundsException`: Thrown to indicate that an array has been accessed with an illegal index.
- `NullPointerException`: Thrown when an application attempts to use null in a case where an object is required.
- `NumberFormatException`: Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.
- `IllegalArgumentException`: Thrown when the values of properties have values that are not accepted by our function call.
- `IllegalStateException`: Thrown when a method has been invoked at an illegal or inappropriate time.
- `IndexOutOfBoundsException`: Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
- `NoSuchElementException`: Thrown by the nextElement method of an Enumeration to indicate that there are no more elements in the enumeration.
- `UnsupportedOperationException`: Thrown to indicate that the requested operation is not supported.

The last special kind of class to be covered is the `annotation` class. Annotations are a special kind of class that can be used to add metadata to your code. They are defined using the `annotation` keyword, and they can be applied to classes, methods, and properties. They also require using the `@` symbol. For example,

```
annotation class MyAnnotation(val someDescription: String)

@MyAnnotation("Class annotation")
class A(
    @MyAnnotation("Constructor property annotation")
val a: Int
) {
    @MyAnnotation("Method annotation")
    fun b() {}
}
```

3.2 Nullability

A `null` value is a value that does not exist. It is used to represent the absence of a value, and it is often used to indicate that a variable has not been initialized. In Kotlin, need to use the `?` operator to indicate that a variable can be `null`; this design convention of making every `null` value explicit aims to decrease the amount of errors due to `null` values. The risk of using nullable types is that if you try to access a property or function on a null variable, it will trigger a null pointer exception. To avoid this, you can either check for null using an `if` statement before accessing the property or function, or use the `?.` operator. The `?.` operator is called the **safe call operator** and should be used where `.` is normally used. If the variable is not `null`, the property or function will be accessed, but if it is `null`, the entire expression will evaluate to `null`. For example,

```
fun main() {
    var message: String? = "Hello World"
    print(message?.uppercase()) //Will print HELLO WORLD
    message = null //Assign null to the variable
    print(message?.uppercase()) //Will print null
}
```

Another way in which nullability can be handled is through the use of the **elvis** operator, `?:`. This operator is used to provide a default value for a nullable variable. If the variable is not `null`, the value of the variable will be used, but if it is `null`, the default value will be used. For example,

```
fun main() {
    var message: String? = "Hello World"
    print(message ?: "Default message") //Will print Hello World
    message = null //Assign null to the variable
    print(message ?: "Default message") //Will print Default message
}
```

Sample problem (directly from course):

```
data class Student(val fullName: String, var id: Int,var grade:Double
)
val students = listOf<Student>(
    Student("John", 223,140.0),
    Student("Mark", 548,120.0),
    Student("Natali", 342,150.0),
    Student("Sara", 781,130.0)
```

```
)

fun main() {
    println("Please, Enter the student's ID")
    val id = readln().toInt()
    println( getStudentById(id))
    println("Please, Enter the student's name")
    val name= readln()
    println(searchInStudents(name)?:"the student is not found")
}

fun getStudentById(id:Int):Student{
return  students.find { it.id==id }!!
}

fun searchInStudents(name:String):Student?{
    return students.find { it.fullName.lowercase()==name.lowercase()}
}
```

3.3 Collections

Collections are objects that represent groups of elements. Normally, collections can be iterated on an element–by–element basis using `for`–loops. Every element of this type can use the `.size` method, which returns the amount of elements stored in it. Furthermore, they can be directly printed, in which case they will be turned into strings beforehand. In general, you can always check if an element is or is not inside a collection using the `in` operator.
The type of collection you may want to use depends on the type of information you need to store.The most common collection elements in Kotlin are:

- List: A list is an ordered list of elements and the elements can be repeated several times. They are created with the `listOf()` method. By default, lists are **read only**, but you can add/delete elements by using the +/− signs. For example,

```
        val list = listOf(1, 2, 3, 4, 5)
        println(list[0]) //1
        println(list.size) //5
        println(list) //[1, 2, 3, 4, 5]
        val list2 = list + 6
        println(list2) //[1, 2, 3, 4, 5, 6]
        val list3 = list2 - 6
        println(list3) //[1, 2, 3, 4, 5]
        for (i in list) {
            println(i)
        }

```

  Mutable lists can be created using the `mutableListOf()` method. These kinds of lists have the same methods as read–only lists, but they can also be modified using the `.add()` and `.remove()` methods.

- Set: A set is an **unordered** list without duplicates. Sets are used when you do not care about the order of elements and there are no duplicates. Importantly, sets can retrieve items faster than lists. They are created with the `setOf()` method, and elements can be added or removed using the +/− signs. For example,

```
        val set = setOf(1, 2, 3, 4, 5)
        println(set) //[1, 2, 3, 4, 5]
        val set2 = set + 6
        println(set2) //[1, 2, 3, 4, 5, 6]
        val set3 = set2 - 6
        println(set3) //[1, 2, 3, 4, 5]

```

  You can check if a set is empty using the `.isEmpty()` method. Besides the `in` keyword, you can also use the `.contains()` method to check if an element is in a set. Lastly, if you want to create a mutable set that can use methods like `.add()` and `.remove()`, you can use the `mutableSetOf()`.

- Map: A map is an unordered collection of **key–value pairs**. Keys are unique and each maps to only one value. These values can be dupplicated. Maps are particularly strong when there are logical connections between key and value, such as an employee ID mapping to the employee's name. They are created with the `mapOf()` method. For example,

```
        val capitals = mapOf("Ethiopia" to "Addis Abeba", "Poland"
    to "Warsaw", "Ukraine" to "Kyiv")
        //    val capitals = mapOf(
        //        Pair("Ethiopia", "Addis Abeba"),
        //        Pair("Poland", "Warsaw"),
        //        Pair("Ukraine", "Kyiv")
        //    )
        println(capitals) // {Ethiopia=Addis Abeba, Poland=Warsaw,
    Ukraine=Kyiv}
        val capital: String? = capitals["Ethiopia"]
        //used nullable because if the key is not in the map, the
    value will be null
        val capitals2 = capitals + ("Panama" to "Panama City")

```

  As before, you can create a mutable map using the `mutableMapOf()` method. This kind of map can use methods like `.put()` and `.remove()`.

Kotlin collections are immutable by default; however their individual items are mutable. When you add/remove elements to a collection with the `var` keyword, you are actually creating a new object and making the variable point to it.