



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
Ingeniería en Sistemas Computacionales



PROGRAMACIÓN ORIENTADA A OBJETOS

Práctica IV

DESARROLLO DE APLICACIONES MÓVILES NATIVAS

Alumno:

Saucedo Moreno César Enrique.

Profesora:

Morales Guitrón Sandra Luz.

Grupo 7CV1

17 / Septiembre / 2024

2025 ~ 1

INTRODUCCIÓN.....	1
DESARROLLO.....	2
Ejercicio 1 - Creando una clase.....	2
Ejercicio 2 - Utilizando constructores para nuestras clases.....	4
Ejercicio 2 - herencia de clases.....	5
Ejercicio 3 - Abstracción.....	6
Ejercicio 4 - Class Interface.....	8
Ejercicio 5 - Clases genéricas.....	11
Ejercicio 6 - Class Variables.....	12
Ejercicio 7 - Member Variables.....	13
Ejercicio 8 - Estructuras de datos en Kotlin.....	14
Hashmaps.....	14
ArrayList.....	16
Listof y mutableListOf.....	21
CONCLUSIÓN.....	23

INTRODUCCIÓN

La Programación Orientada a Objetos (POO) es uno de los paradigmas más utilizados en el desarrollo de software moderno, y aprenderla es esencial para dominar lenguajes como Kotlin, especialmente porque gran parte del código de hoy en día sigue este enfoque. La POO facilita la creación de aplicaciones modulares, escalables y mantenibles al organizar el software en clases y objetos, lo que ayuda a gestionar la complejidad de proyectos grandes.

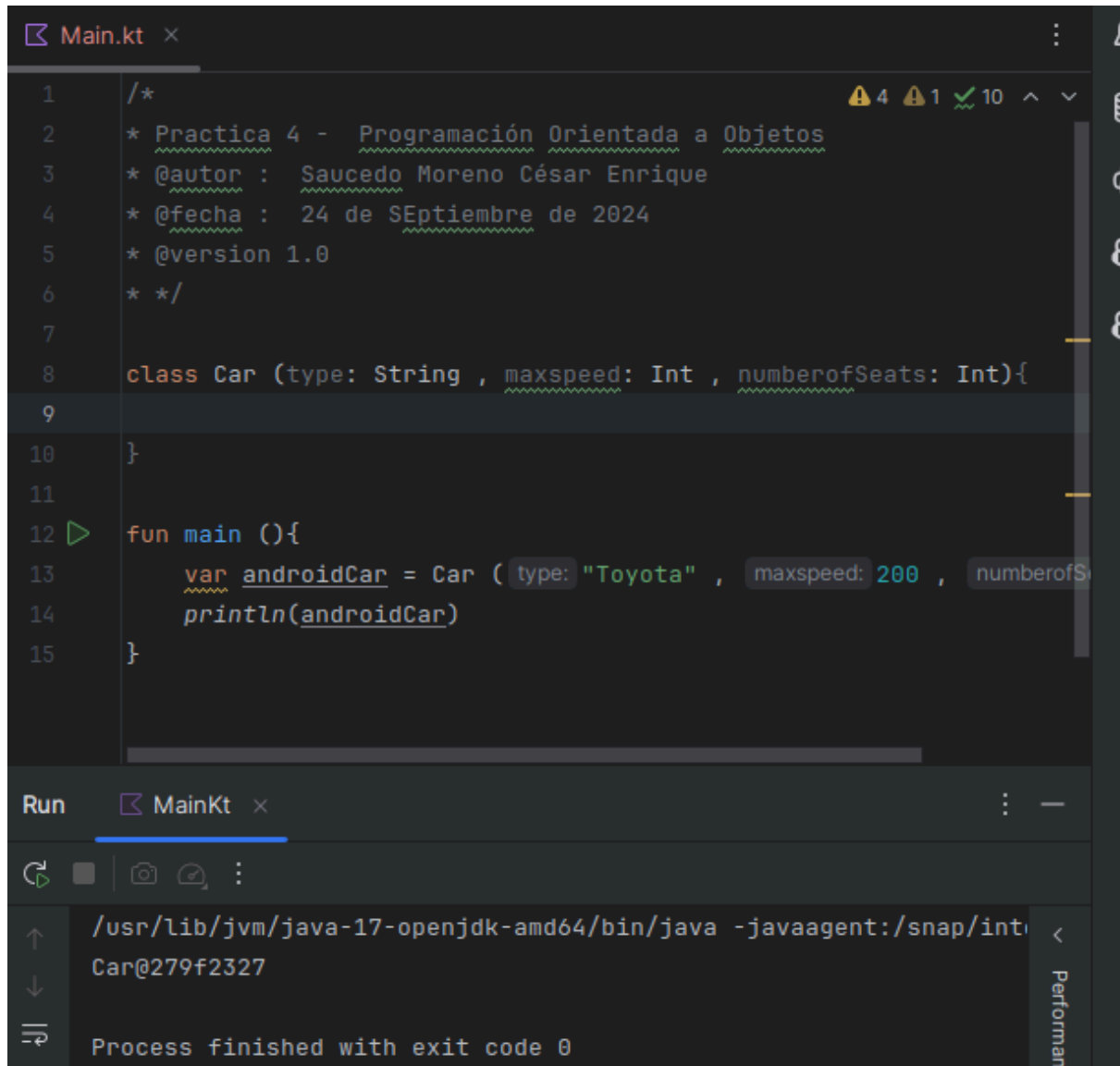
Al aprender Kotlin, entender este paradigma es fundamental, ya que es un lenguaje que sigue muchos de los principios de la POO, al igual que lenguajes como Java, C++, o Python. En Kotlin se trabajan conceptos como la herencia, encapsulamiento, polimorfismo y abstracción y se implementan de manera similar a estos otros lenguajes, lo que facilita la transición si hemos trabajado con alguno de ellos.

La importancia de dominar estos principios radica en que te permitirán adaptarte fácilmente a otros lenguajes orientados a objetos, además de fomentar buenas prácticas de diseño que te serán útiles independientemente de la tecnología que utilices.

DESARROLLO

Ejercicio 1 - Creando una clase

En esta primera parte del ejercicio observamos cómo es que se manejan las clases en Kotlin, esto es fundamental para aprender el nuevo lenguaje.



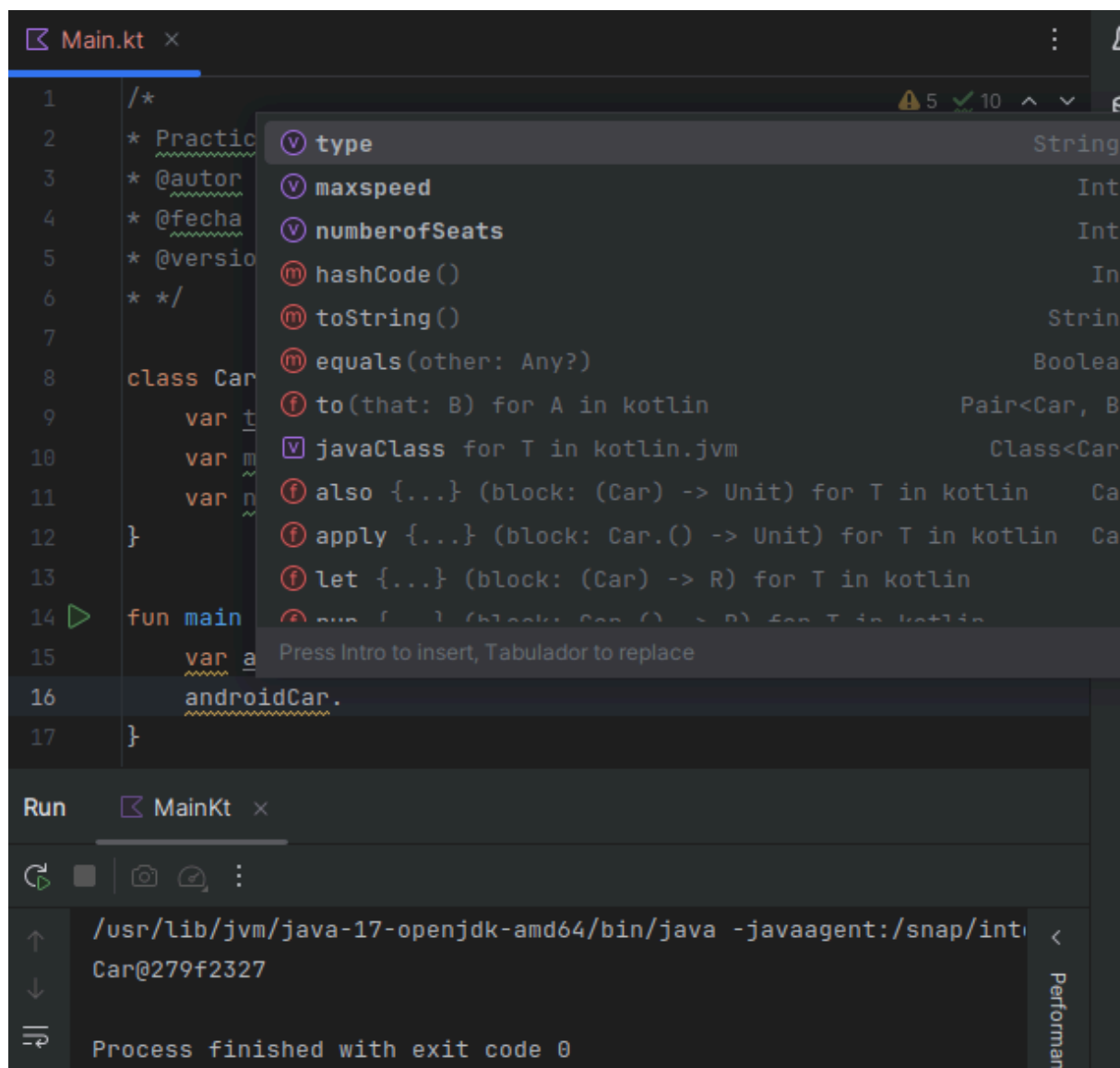
```
1  /*
2  * Practica 4 - Programación Orientada a Objetos
3  * @autor : Saucedo Moreno César Enrique
4  * @fecha : 24 de SEptiembre de 2024
5  * @version 1.0
6  * */
7
8  class Car (type: String , maxspeed: Int , numberOfSeats: Int){
9
10 }
11
12 fun main (){
13     var androidCar = Car ( type: "Toyota" , maxspeed: 200 , numberOfS
14     println(androidCar)
15 }
```

Run MainKt x

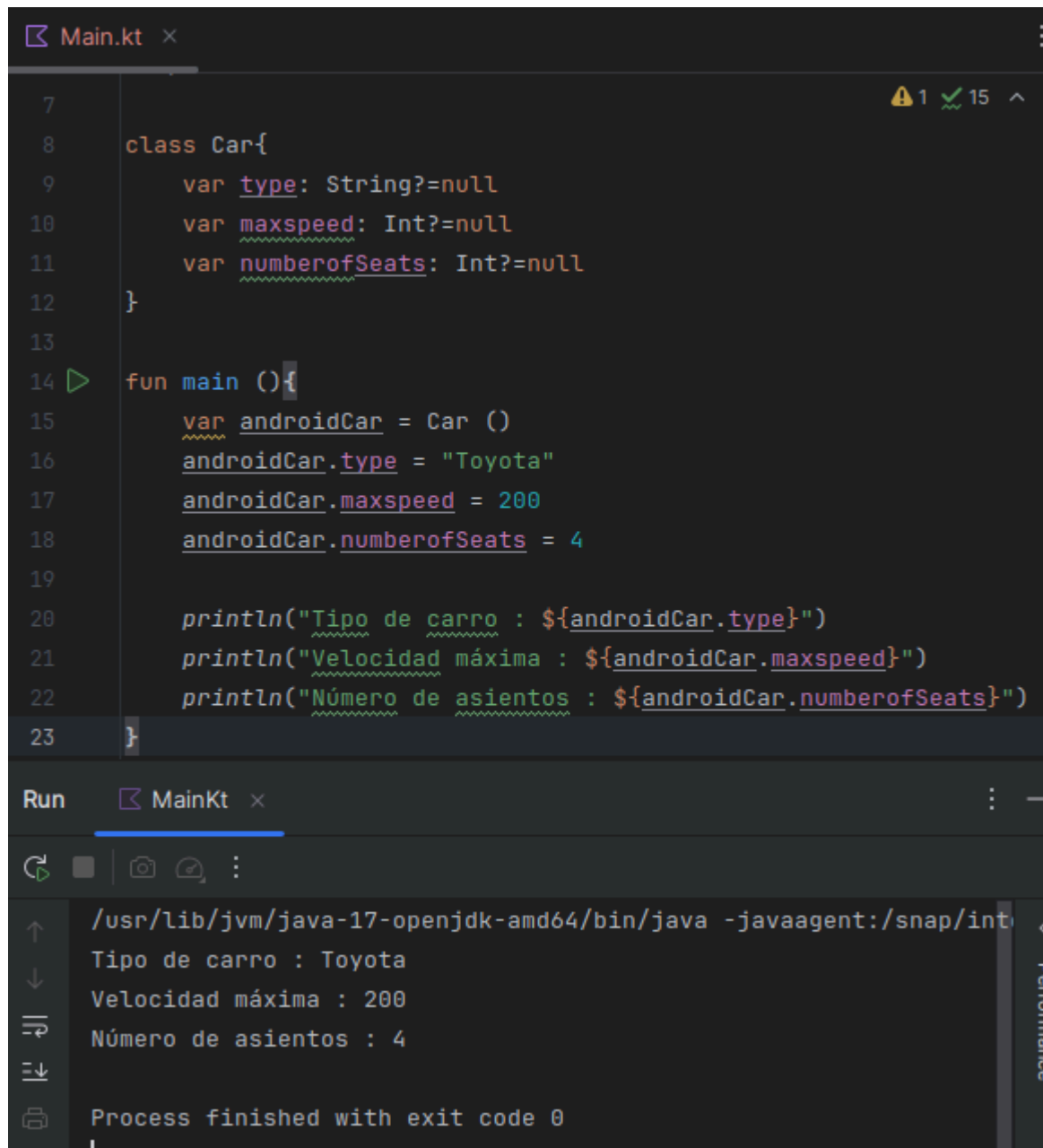
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intel
Car@279f2327

Process finished with exit code 0

Posteriormente definimos el tipo de atributos que pueden ser definidos después de crear la clase y cómo podemos utilizar "." para acceder a atributos de una clase para definir algunos de sus atributos.



Posteriormente podemos definirlos.



```
7
8 class Car{
9     var type: String?=null
10    var maxspeed: Int?=null
11    var numberOfSeats: Int?=null
12 }
13
14 fun main (){
15     var androidCar = Car ()
16     androidCar.type = "Toyota"
17     androidCar.maxspeed = 200
18     androidCar.numberOfSeats = 4
19
20     println("Tipo de carro : ${androidCar.type}")
21     println("Velocidad máxima : ${androidCar.maxspeed}")
22     println("Número de asientos : ${androidCar.numberOfSeats}")
23 }
```

Run MainKt

```
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/int
Tipo de carro : Toyota
Velocidad máxima : 200
Número de asientos : 4
Process finished with exit code 0
```

Ejercicio 2 - Utilizando constructores para nuestras clases.

En él nos explican cómo es que Kotlin utiliza o se declaran los constructores de nuestras clases, en donde, lo correcto sería definir todas las propiedades dentro del constructor o inicializarlas correctamente.

```
14 class Student (name: String , college: String , age: Int){
15     var name: String ?= name
16     var college: String ?= college
17     var age: Int ?= age
18 }
19
20 fun main (){
21     var itStudent = Student( name: "Cesar", college: "ESCOM", a
22     println("Nombre: ${itStudent.name}")
23     println("Colegio: ${itStudent.college}")
24     println("Edad: ${itStudent.age}")
25 }
26
```

Run MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/int
Nombre: Cesar
Colegio: ESCOM
Edad: 21
Process finished with exit code 0

Ejercicio 2 - herencia de clases.

Es un mecanismo en la OOP que nos permite heredar propiedades y comportamientos de otras clases, esto es útil para reutilizar código y crear una jerarquía de clases que se comportan o tienen ciertas características comunes.

Por defecto, todas las clases son finales por defecto, es decir, no se pueden heredar a menos que se marque explícitamente con la palabra clave open.

Además, nos permite sobrescribir métodos o propiedades de las superclases y se utiliza la palabra override.

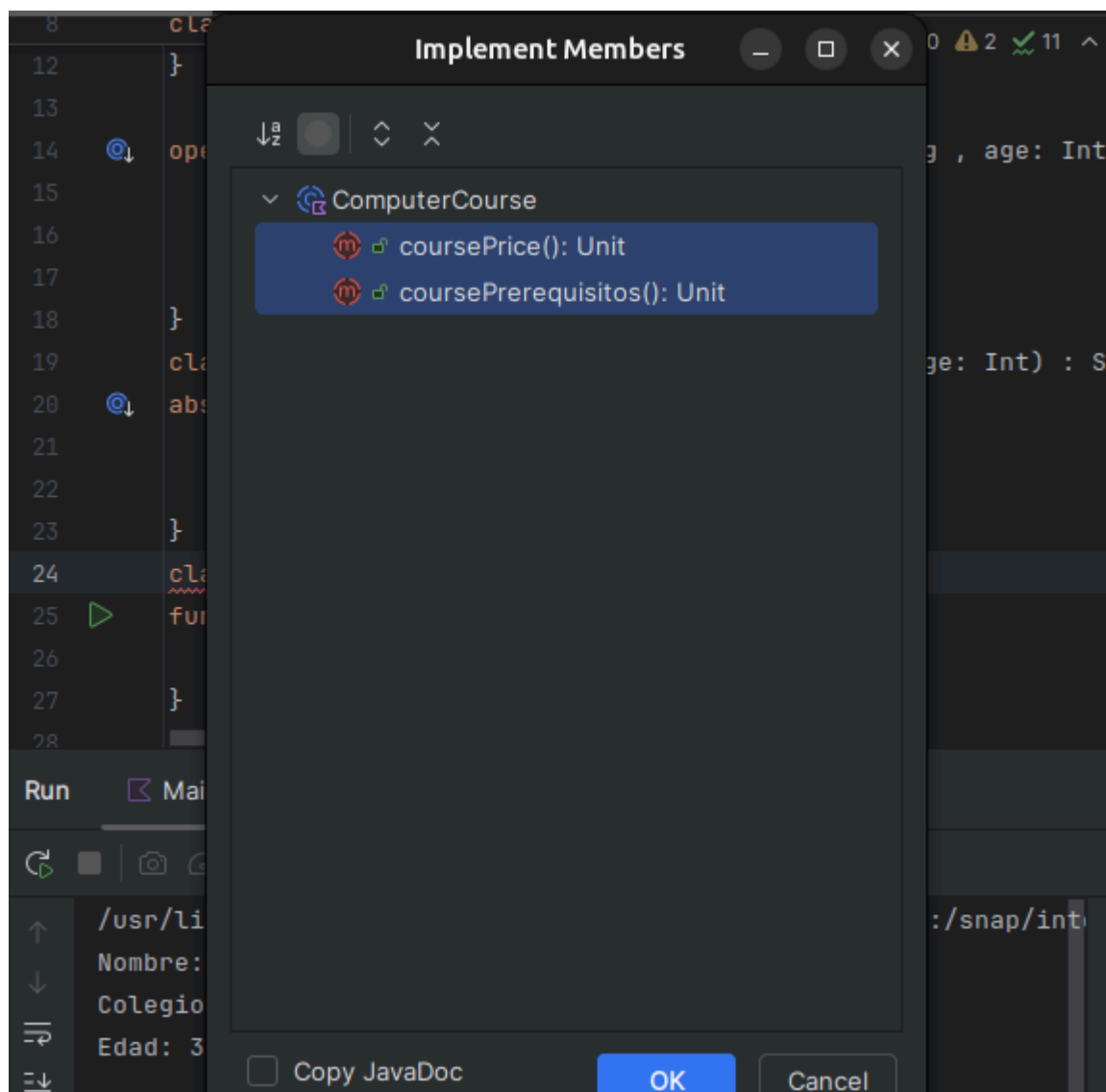
```
14  @ open class Student (name: String , college: String , age: Int) {
15      var name: String ?= name
16      var college: String ?= college
17      var age: Int ?= age
18  }
19  class Teacher (name: String , college: String , age: Int) : Student() {
20  fun main () {
21      /*var itStudent = Student("Cesar", "ESCOM", 21)
22      println("Nombre: ${itStudent.name}")
23      println("Colegio: ${itStudent.college}")
24      println("Edad: ${itStudent.age}")*/
25      var itStudent = Teacher( name: "Vanesa", college: "ESCOM", age: 32)
26      println("Nombre: ${itStudent.name}")
27      println("Colegio: ${itStudent.college}")
28      println("Edad: ${itStudent.age}")
29  }
```

Run MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intel-opencl/...
Nombre: Vanesa
Colegio: ESCOM
Edad: 32
Process finished with exit code 0

Ejercicio 3 - Abstracción.

Debido a que es una clase abstracta no puede ser instanciada directamente, sino que sirve como una base para otras clases, nos puede ayudar a definir comportamientos y propiedades comunes que las subclases deben implementar o heredar, además, nos permite crear estructuras genéricas con comportamiento parcial, que luego puede ser completado por las subclases.



```
20  @  abstract class ComputerCourse {
21      abstract fun coursePrice()
22      abstract fun coursePrerequisitos()
23  }
24  class languageCourse (): ComputerCourse() {
25      override fun coursePrice() {
26          println("Precio del curso :")
27      }
28      override fun coursePrerequisitos() {
29          println("Prerequisitos del curso :")
30      }
31  }
32  fun main () {
33      var x = languageCourse()
34      x.coursePrice()
35      x.coursePrerequisitos()
36  }
```

Run MainKt x

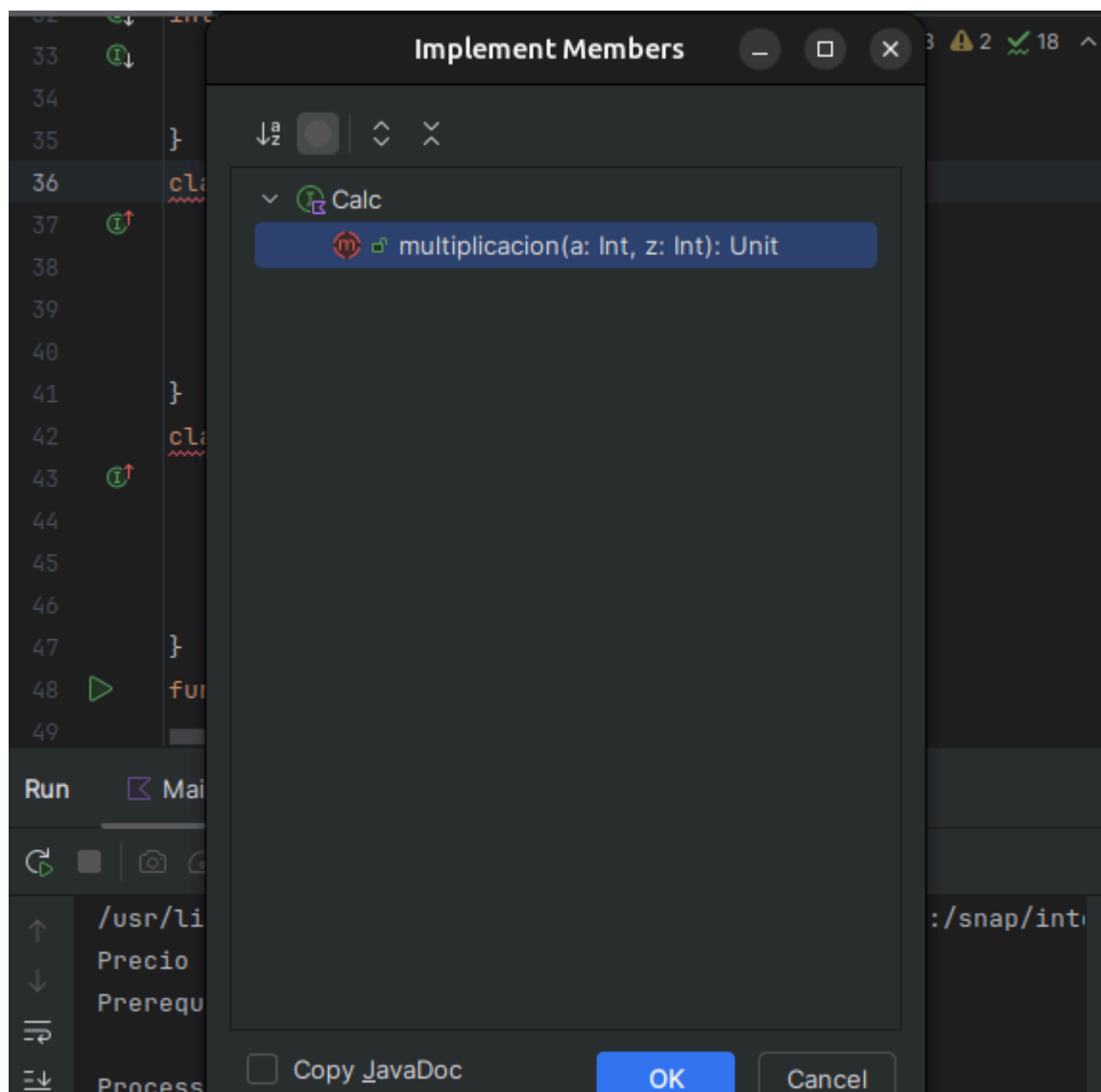
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intel
Precio del curso :
Prerequisitos del curso :
Process finished with exit code 0

La abstracción en POO permite definir una clase o una interfaz que solo expone comportamientos esenciales, ocultando los detalles de implementación. Esto es útil para diseñar sistemas de manera jerárquica, donde las clases más específicas (subclases) se encargan de completar la funcionalidad.

Ejercicio 4 - Class Interface.

Son aquellas clases que definen un conjunto de comportamientos que las clases que las implementan deben cumplir. A diferencia de las clases abstractas, las interfaces no contienen estado (atributos) y pueden ser implementadas por múltiples clases, lo que permite un diseño más flexible y modular.

Creando la clase dedicada para la suma.



```
1 interface Calc {
2     fun sum(x: Int, y: Int)
3     fun multiplicacion(a: Int , z: Int)
4 }
5
6 class Math : Calc {
7     override fun sum(x: Int, y: Int) {
8         var sumtotal = x + y
9         println("La suma de $x + $y = $sumtotal")
10    }
11
12    override fun multiplicacion(a: Int, z: Int) {
13        var multi = a * z
14        println("La multiplicación de $a * $z = $multi")    }
15 }
16
17 class Math2 : Calc {
18     override fun sum(x: Int, y: Int) {
19         var multi = x * y
20         println("La suma de $x + $y = $multi")
21    }
22
23    override fun multiplicacion(a: Int, z: Int) {
24        var multi = a * z
25        println("La multiplicación de $a * $z = $multi")
26    }
27 }
```

on MainKt x

Ejecución del código.

```
}  
class Math2 : Calc {  
    override fun sum(x: Int, y: Int) {  
        var multi = x * y  
        println("La suma de $x + $y = $multi")  
    }  
    override fun multiplicacion(a: Int, z: Int) {  
        var multi = a * z  
        println("La multiplicación de $a * $z = $multi")  
    }  
}  
  
fun main () {  
    var operacion = Math()  
    operacion.sum(x: 5, y: 5)  
}
```

MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intelli
La suma de 5 + 5 = 10

Process finished with exit code 0

Algunas de las ventajas de utilizar interfaces son implementar múltiples comportamientos de diferentes interfaces en una sola clase, ofreciendo más flexibilidad.

Ejercicio 5 - Clases genéricas.

Las clases genéricas en Kotlin son aquellas que permiten definir estructuras de datos o clases que pueden trabajar con cualquier tipo de dato sin tener que especificar el tipo exacto en el momento de su definición. Esto es útil para crear clases, interfaces y funciones que pueden ser reutilizadas de forma más flexible, sin depender de tipos específicos.

Una clase genérica en Kotlin utiliza parámetros de tipo, que son representados por parámetros genéricos como `<T>`. Este parámetro puede ser reemplazado con cualquier tipo (como `Int`, `String`, `Double`, etc.) cuando se instancia la clase.

Función `main` que nos ayuda a ejecutar los métodos de la clase generica, donde se le asigna sus respectivos atributos.

```
Main.kt x
55 // Ambas son correctas para definir una clase generica
56 // Esta primera nos permite definir en un inicio el tipo de dato que se va a utilizar
57 // class Permission(userName: String , password: String)
58 class Permission <T> {
59     var userName: T ?= null
60     var password: T ?= null
61     var ID: T ?= null
62 }
63 fun main (){
64     val x = Permission<String> ()
65     x.userName = "cesar123@gmail.com"
66     x.password = "contrasenia123"
67     println("El usuario es : ${x.userName} y su contraseña es : ${x.password}")
68
69     val y = Permission<String> ()
70     y.userName = "adriana@gmail.com"
71     y.password = 12345678.toString()
72     println("El usuario es : ${y.userName} y su contraseña es : ${y.password}")
73
Run MainKt x
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intellij-idea-ultimate/530/
El usuario es : cesar123@gmail.com y su contraseña es : contrasenia123
El usuario es : adriana@gmail.com y su contraseña es : 12345678
El ID es : 2024
```

Ejercicio 6 - Class Variables.

Se definen utilizando la palabra clave companion object. Estas variables son compartidas por todas las instancias de la clase y se comportan como variables estáticas en otros lenguajes de programación.

Se definen dentro de un companion object en la clase y se utilizan para almacenar datos que deben ser compartidos entre todas las instancias de la clase o para definir constantes y métodos estáticos. Esta nos permite definir el tipo de dato en el momento de la creación del objeto.

```
// Esta segunda nos permite definir el tipo de dato en el momento
class Juego{
    companion object {
        val juegosJugados = 10
    }
}

fun main(){
    println("La cantidad de juegos jugados son:\t" + Juego.juegosJugados)
}
```

MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intel...
La cantidad de juegos jugados son: 10

Process finished with exit code 0

Ejercicio 7 - Member Variables.

Son propiedades que pertenecen a una instancia específica de una clase. Estas variables se definen dentro de la clase y cada instancia de la clase tiene su propia copia de estas variables.

Se utilizan para almacenar datos específicos de una instancia de la clase.

```
open class Avion() {
    var tipo: String?=null
    var capacidad: Int?=null
}

class Bus():Avion()
var volvo_bus = Bus()

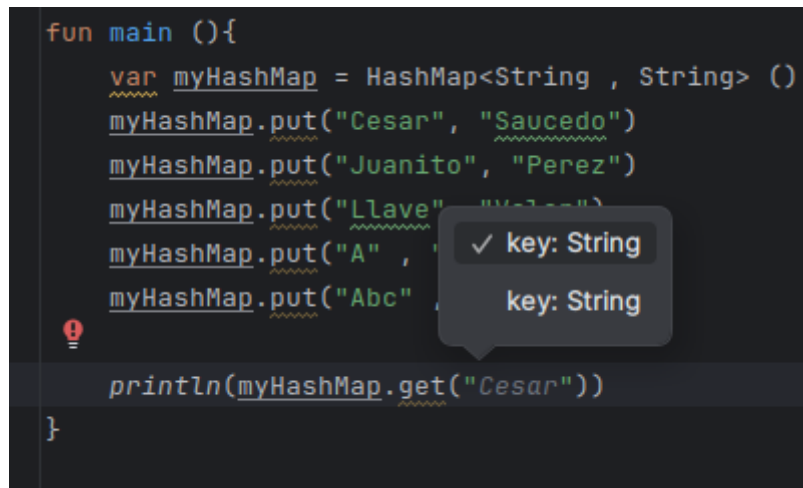
fun main(){
    println(volvo_bus.capacidad)
}
```

Ejercicio 8 - Estructuras de datos en Kotlin.

Hashmaps.

Es una estructura de datos que almacena pares clave-valor y permite la búsqueda rápida de valores basados en sus claves. En ella se permite modificaciones como agregar, eliminar y actualizar elementos.

En la siguiente captura vemos que el propio IDE nos ayuda al solicitar la llave.



```
fun main (){  
    var myHashMap = HashMap<String , String> ()  
    myHashMap.put("Cesar", "Saucedo")  
    myHashMap.put("Juanito", "Perez")  
    myHashMap.put("Llave", "Valor")  
    myHashMap.put("A" , " ")  
    myHashMap.put("Abc" , " ")  
    println(myHashMap.get("Cesar"))  
}
```

The screenshot shows a Kotlin code snippet in an IDE. A tooltip is visible over the code, displaying '✓ key: String' and 'key: String', indicating the IDE's suggestion for the key type in the HashMap.

En la siguiente captura vemos que para imprimir los valores es necesario acceder a las llaves con el método get de la clase HashMap.


```
> fun main (){  
    var myHashMap = HashMap<String , String> ()  
    myHashMap.put("Cesar", "Saucedo")  
    myHashMap.put("Juanito", "Perez")  
    myHashMap.put("Llave", "Valor")  
    myHashMap.put("A" , "Android")  
    myHashMap.put("Abc" , "Google")  
  
    println("Valor:\t" + myHashMap.get("Cesar"))  
    println("Valor:\t" + myHashMap.get("Juanito"))  
    println("Valor:\t" + myHashMap.get("Llave"))  
    println("Valor:\t" + myHashMap.get("A"))  
    println("Valor:\t" + myHashMap.get("Abc"))  
}
```

MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:
Valor: Saucedo
Valor: Perez
Valor: Valor
Valor: Android
Valor: Google

Forma de acceder a los valores mediante un ciclo for mediante sus llaves o keys.

```
fun main (){
    var myHashMap = HashMap<String , String> ()
    myHashMap.put("Cesar", "Saucedo")
    myHashMap.put("Juanito", "Perez")
    myHashMap.put("Llave", "Valor")
    myHashMap.put("A" , "Android")
    myHashMap.put("Abc" , "Google")


    for (x in myHashMap.keys){
        println ("Valor de $x:\t" + myHashMap.get(x))
    }
}
```

MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/s
Valor de Llave: Valor
Valor de A: Android
Valor de Juanito: Perez
Valor de Abc: Google
Valor de Cesar: Saucedo

ArrayList.

Es una estructura de datos que permite almacenar una colección de elementos que pueden ser modificados (agregar, eliminar, actualizar). Es mutable y mantiene el orden de inserción de los elementos. Además, permite acceder a los elementos mediante su índice.

```
6  ▶ fun main () {
7      var myArrayList = ArrayList<Int> ()
8      myArrayList.add(20)
9      myArrayList.add(30)
10     myArrayList.add(50)
11     
12     println("El array es:\t" + myArrayList)
13 }
14
```

un MainKt x

```
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/s
El array es:    [20, 30, 50]

Process finished with exit code 0
```

Además, el ArrayList nos permite acceder a los datos mediante su índice mediante el método get de la clase del ArrayList.

```
6  ▶ fun main () {
7      var myArrayList = ArrayList<Int> ()
8      myArrayList.add(20)
9      myArrayList.add(30)
10     myArrayList.add(50)
11     println("Accediento al valor del indice 0 :\t" + myArrayList[0])
12     //println("El array es:\t" + myArrayList)
13 }
14
```

un MainKt x

```
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/s
Accediento al valor del indice 0 : 20

Process finished with exit code 0
```

Existen muchas formas de acceder a las colecciones del arrayList mediante bucles, el siguiente es una forma de cómo mostrarlos.

```
fun main () {  
    var myArrayList = ArrayList<Int> ()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
    //println("Accediento al valor del indice 0 :\t" +  
    //println("El array es:\t" + myArrayList)  
    for (index in 0 .. myArrayList.size-1) {  
        println(myArrayList[index])  
    }  
}
```

MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/sr
20
30
50

Además se puede hacer validaciones si existen números dentro de nuestro ArrayList.

```
fun main (){
    myArrayList.add(20)
    myArrayList.add(30)
    myArrayList.add(50)
    //println("Accediento al valor del indice 0 :\t" + myArrayList[0])
    //println("El array es:\t" + myArrayList)
    /*for (index in 0..myArrayList.size-1) {
        println(myArrayList[index])
    }*/
    if (myArrayList.contains(30)){
        println("El valor 30 se encuentra en el array")
    }else{
        println("El valor 30 no se encuentra en el array")
    }
}
```

MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intelli

El valor 30 se encuentra en el array

Process finished with exit code 0

Además, se nos permite eliminar elementos de nuestro array, esto utilizando el método `remove` de nuestra clase `ArrayList`.

También existe el método `set` de la clase `ArrayList` en Kotlin se utiliza para reemplazar el elemento en una posición específica con un nuevo valor. Este método toma dos parámetros: el índice del elemento que se desea reemplazar y el nuevo valor que se desea establecer en esa posición.

```
6 fun main () {
7     var myArrayList = ArrayList<Int> ()
8     myArrayList.add(20)
9     myArrayList.add(30)
10    myArrayList.add(50)
11    myArrayList.set(1,100)
12    for (index in 0 ..< myArrayList.size-1){
13        println("Elemento en el indice $index es ${myArrayList.get(index)})")
14    }
15 }
16
```

un MainKt x

```
/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intel
Elemento en el indice 0 es 20
Elemento en el indice 1 es 100
Elemento en el indice 2 es 50
```

Listof y mutableListOf

Son funciones para crear listas. La diferencia principal entre ellas es la mutabilidad.

Mientras que `listOf` es inmutable, ya que no permite agregar, eliminar o modificar elementos desde su creación la hace ideal para listas que no necesitan cambiar.

Mientras que `mutableListOf` permite agregar, eliminar y modificar sus elementos, es ideal para listas que necesitan cambios dinámicos.

```
76 ▶ fun main () {
77     var myList = mutableListOf<Any> (1, "Cesar Saucedo", 500.52)
78     myList[0] = 200
79     for (index in 0 ≤ .. ≤ myList.size-1) {
80         println(myList[index])
81     }
82 }
83
```

Run MainKt x

/usr/lib/jvm/java-17-openjdk-amd64/bin/java -javaagent:/snap/intel
200
Cesar Saucedo
500.52

CONCLUSIÓN

Aprender programación orientada a objetos (POO) es fundamental cuando se aborda el aprendizaje de un nuevo lenguaje de programación. Esta metodología no solo permite identificar si el lenguaje en cuestión soporta paradigmas orientados a objetos, sino que también ofrece herramientas esenciales para desarrollar aplicaciones más legibles y modulares. Al estructurar el código en objetos y clases, se promueve la reutilización y el mantenimiento del código, evitando la repetición innecesaria.

Por lo tanto, este tipo de paradigmas nos ayuda a agilizar el proceso de desarrollo, además que fomenta en nosotros la creación de aplicaciones escalables y eficientes, habilidades clave en cualquier entorno de desarrollo.