POO en Kotlin

POO significa **Programación Orientada a Objetos.**

La programación procedimental consiste en escribir procedimientos o métodos que realizan operaciones sobre los datos, mientras que la programación orientada a objetos consiste en crear objetos que contienen tanto datos como métodos.

La programación orientada a objetos tiene varias ventajas sobre la programación procedimental:

- La programación orientada a objetos es más rápida y fácil de ejecutar
- La programación orientada a objetos proporciona una estructura clara para los programas.
- La programación orientada a objetos ayuda a mantener el código Kotlin DRY"Don't Repeat Yourself" y hace que el código sea más fácil de mantener, modificar y depurar.
- La programación orientada a objetos permite crear aplicaciones totalmente reutilizables con menos código y un tiempo de desarrollo más corto.

Consejo: El principio "No repetirse" (DRY) trata de reducir la repetición del código. Debe extraer los códigos que son comunes para la aplicación, colocarlos en un solo lugar y reutilizarlos en lugar de repetirlos.

Kotlin: ¿Qué son las clases y los objetos?

Las clases y los objetos son los dos aspectos principales de la programación orientada a objetos.

Mire lo siguiente para ver la diferencia entre clase y objetos:

clase: Fruit
objetos: Apple, Banana, Mango

Otro ejemplo:

clase: Auto
objetos: volvo, Audi, toyota

Entonces, una clase es una plantilla para objetos y un objeto es una instancia de una clase.

Clases/Objetos de Kotlin

Todo en Kotlin está asociado con clases y objetos, junto con sus propiedades y funciones. Por ejemplo: en la vida real, un coche es un objeto. El automóvil tiene propiedades, como marca, peso y color, y funciones, como conducción y freno.

POO en Kotlin

Una clase es como un constructor de objetos o un "modelo" para crear objetos.

Crear una clase

Para crear una clase, use la palabra **class** clave y especifique el nombre de la clase:

Ejemplo
Cree una clase de **Car** junto con algunas propiedades (brand, model y year)

```
class Car {
  var brand = ""
  var model = ""
  var year = 0
}
```

Una **propiedad** es básicamente una **variable** que pertenece a la clase.

Es bueno saberlo: se considera una buena práctica comenzar el nombre de una clase con una letra mayúscula, para una mejor organización.

Crear un objeto

Ahora podemos usar la clase llamada **Car** para crear objetos.

En el siguiente ejemplo, creamos un objeto de **Car** llamado **c1** y luego accedemos a las propiedades de **c1** usando la sintaxis de puntos (.), tal como lo hicimos para acceder a las propiedades de matriz y cadena:

Ejemplo
```
// Create a c1 object of the Car class
val c1 = Car()

// Access the properties and add some values to it
c1.brand = "Ford"
c1.model = "Mustang"
c1.year = 1969

println(c1.brand)   // Outputs Ford
println(c1.model)   // Outputs Mustang
println(c1.year)    // Outputs 1969
```

Múltiples objetos

Puedes crear múltiples objetos de una clase:

Ejemplo
```
val c1 = Car()
c1.brand = "Ford"
```

```
c1.model = "Mustang"
c1.year = 1969

val c2 = Car()
c2.brand = "BMW"
c2.model = "X5"
c2.year = 1999

println(c1.brand)  // Ford
println(c2.brand)  // BMW
```

Constructor Kotlin

Anteriormente, creamos un objeto de una clase y especificamos las propiedades dentro de la clase, así:

Ejemplo

```
class Car {
  var brand = ""
  var model = ""
  var year = 0
}

fun main() {
  val c1 = Car()
  c1.brand = "Ford"
  c1.model = "Mustang"
  c1.year = 1969
}
```

En Kotlin, existe una forma más rápida de hacer esto mediante el uso de un **constructor**.

Un constructor es como una función especial y se define mediante el uso de dos paréntesis () después del nombre de la clase. Puede especificar las propiedades dentro de los paréntesis (como pasar parámetros a una función normal).

El constructor inicializará las propiedades cuando crees un objeto de una clase. Sólo recuerde especificar el tipo de propiedad/variable:

Ejemplo

```
class Car(var brand: String, var model: String, var year: Int)

fun main() {
  val c1 = Car("Ford", "Mustang", 1969)
}
```

Ahora es aún más fácil especificar varios objetos de una clase:

Ejemplo

```
class Car(var brand: String, var model: String, var year: Int)
```

```
fun main() {
  val c1 = Car("Ford", "Mustang", 1969)
  val c2 = Car("BMW", "X5", 1999)
  val c3 = Car("Tesla", "Model S", 2020)
}
```

Funciones de clase de Kotlin

También puedes usar funciones dentro de una clase, para realizar ciertas acciones:

Ejemplo
Crea una drive()función dentro de la Carclase y llámala:

```
class Car(var brand: String, var model: String, var year: Int) {
  // Class function
  fun drive() {
    println("Wrooom!")
  }
}

fun main() {
  val c1 = Car("Ford", "Mustang", 1969)

  // Call the function
  c1.drive()
}
```

Consejo: cuando una función se declara dentro de una clase, se la conoce como función de clase o función miembro.

Nota: Cuando se crea un objeto de la clase, tiene acceso a todas las funciones de la clase.

Parámetros de función de clase

Al igual que con las funciones normales, puedes pasar parámetros a una función de clase:

Ejemplo
Cree dos funciones: drive()y speed()y pase parámetros a la speed()función:

```
class Car(var brand: String, var model: String, var year: Int) {
  // Class function
  fun drive() {
    println("Wrooom!")
  }

  // Class function with parameters
  fun speed(maxSpeed: Int) {
    println("Max speed is: " + maxSpeed)
  }
```

```
}

fun main() {
  val c1 = Car("Ford", "Mustang", 1969)

  // Call the functions
  c1.drive()
  c1.speed(200)
}
```

Herencia de Kotlin (subclase y superclase)

En Kotlin, es posible heredar propiedades y funciones de una clase a otra. Agrupamos el "concepto de herencia" en dos categorías:

subclase (secundaria): la clase que hereda de otra clase
superclase (padre): la clase de la que se hereda

En el siguiente ejemplo, MyChildClass(subclase) hereda las propiedades de la MyParentClassclase (superclase):

Ejemplo

```
// Superclass
open class MyParentClass {
  val x = 5
}

// Subclass
class MyChildClass: MyParentClass() {
  fun myFunction() {
    println(x) // x is now inherited from the superclass
  }
}

// Create an object of MyChildClass and call myFunction
fun main() {
  val myObj = MyChildClass()
  myObj.myFunction()
}
```

Utilice la palabra clave **open** delante de la **superclase**/parent, para que esta sea la clase de la que otras clases deberían heredar propiedades y funciones.

Para heredar de una clase, especifique el nombre de la **subclase**, seguido de dos puntos **:** y luego el nombre de la **superclase**.

Ejercicios

E1001: Implementar una clase llamada Alumno que tenga como propiedades su nombre y su nota. Al constructor llega su nombre y nota.
Imprimir el nombre y su nota. Mostrar un mensaje si está regular (nota mayor o igual a 4)
Definir dos objetos de la clase Alumno.

```
class Alumno (val nombre: String, val nota: Int){

    fun imprimir() {
        println("Alumno: $nombre tiene una nota de $nota")
    }

    fun mostrarEstado () {
        if (nota >= 4)
            println("$nombre se encuentra en estado REGULAR")
        else
            println("$nombre no está REGULAR")
    }
}

fun main(parametros: Array<String>) {
    val alumno1 = Alumno("Ana", 7)
    alumno1.imprimir()
    alumno1.mostrarEstado()
    val alumno2 = Alumno("Carlos", 2)
    alumno2.imprimir()
    alumno2.mostrarEstado()
}
```

E1002: Declarar luego una clase llamada CalculadoraCientifica que herede de Calculadora y añada las responsabilidades de calcular el cuadrado del primer número y la raíz cuadrada.

```
open class Calculadora(val valor1: Double, val valor2: Double ){
    var resultado: Double = 0.0
    fun sumar() {
        resultado = valor1 + valor2
    }

    fun restar() {
        resultado = valor1 - valor2
    }

    fun multiplicar() {
        resultado = valor1 * valor2
    }

    fun dividir() {
        resultado = valor1 / valor2
    }

    fun imprimir() {
```

```
        println("Resultado: $resultado")
    }
}

class CalculadoraCientifica(valor1: Double, valor2: Double):
Calculadora(valor1, valor2) {
    fun cuadrado() {
        resultado = valor1 * valor1
    }

    fun raiz() {
        resultado = Math.sqrt(valor1)
    }
}

fun main(parametro: Array<String>) {
    println("Prueba de la clase Calculadora (suma de dos números)")
    val calculadora1 = Calculadora(10.0, 2.0)
    calculadora1.sumar()
    calculadora1.imprimir()
    println("Prueba de la clase Calculadora Cientrífica (suma de dos
números y el cuadrado y la raiz del primero)")
    val calculadoraCientifica1 = CalculadoraCientifica(10.0, 2.0)
    calculadoraCientifica1.sumar()
    calculadoraCientifica1.imprimir()
    calculadoraCientifica1.cuadrado()
    calculadoraCientifica1.imprimir()
    calculadoraCientifica1.raiz()
    calculadoraCientifica1.imprimir()
}
```

E1003: Declarar una clase *Dado* que genere un valor aleatorio entre 1 y 6, mostrar su valor. Crear una segunda clase llamada *DadoRecuadro* que genere un valor entre 1 y 6, mostrar el valor recuadrado en asteriscos. Utilizar la herencia entre estas dos clases.

```
open class Dado{
    protected var valor: Int = 1
    fun tirar() {
        valor = ((Math.random() * 6) + 1).toInt()
    }

    open fun imprimir() {
        println("$valor")
    }
}

class DadoRecuadro: Dado() {
    override fun imprimir() {
        println("***")
        println("*$valor*")
        println("***")
    }
```

```
}

fun main(parametro: Array<String>) {
    val dado1 = Dado()
    dado1.tirar()
    dado1.imprimir()

    val dado2 = DadoRecuadro()
    dado2.tirar()
    dado2.imprimir()
}
```

E1004: Diseñar la clase Hora, que representa un instante de tiempo compuesto por la hora (de 0 a 23) y los minutos. Dispone de los métodos:

- Hora(hora, minuto) que construye un objeto con los datos pasados como parámetros.
- inc() que incrementa la hora en un minuto
- setMinutos(valor) que asigna un valor, si es válido, a los minutos. Develve true o false según haya sido posible modificar los minutos o no.
- setHora(valor), que asigna un valor, si está comprendido entre 0 y 23, a la hora. Devuelve true o false según haya sido posible cambiar la hora o no.
- toString(), que devuelve un String con la representación de la hora.

E1005: A partir de la clase Hora implementar la clase HoraExacta, que incluye en la hora los segundos. Además de los métodos heredados de Hora, dispondrá de:

- HoraExacta(hora, minuto, segundo), que construye un objeto con los datos pasados como parámetros.
- setSegundo(valor), que asigna, si está comprendido entre 0 y 59, el valor indicado a los segundos.
- inc() que incrementa la hora en un segundo