CLASE INCIO KOTLIN 23/10/2023

- Vamos a practicar kotlin en Intellij
- En kotlin, además de la clase, también podemos tener un File, que es un fichero donde no necesariamente tiene que haber una clase. Dentro de File puede haber una clase, una variable, o una función
- No hay que poner ***;***
- No es necesario poner el ***new***
- La palabra reservada ***lateinit*** va precediendo a *val* o *var*, se utiliza para indicar que una variable no necesita ser inicializada en el momento de su declaración, pero deberá inicializarse antes de su primer uso. Si no la inicializamos antes de usarla nos va a dar un error

VARIABLES

- Pueden ser: estáticas - dinámicas // bloque - clase // referencia : en Kotlin no hay primitivas (por lo que usaremos siempre las envolventes con mayúscula ejem Int
- DEFINICIÓN: primero le decimos si es estática ***val*** o dinámica ***var,*** acompañado del nombre de la variable = valor, siendo optativo ponerle el tipo antecedido de dos puntos:

var nombreVariable: tipo = valor —> mutable
val nombreVariable: tipo = valor —> no mutable

```
var nombre : String = “Maria"
var edad = 27
val DNI ="12345678Y"
```

- **NULL SAFETY**: Si yo quiero definir una variable pero no quiero darle el valor, podemos ponerle la ? y así le indicamos que es un posible nulo y no nos dará una excepción

```
var correo : String? = null
```

FUNCIONES

- Para crear una función ponemos la palabra reservada fun nombre de la función (parámetros de la función: el tipo) : tipo del retorno y las llaves abro y cierro:

fun nombreFuncion (param1: Tipo, param2: Tipo) : TipoRetorno { }

- En Kotlin no existe void , aquí es Unit cuando no nos retorna nada

```
fun par (x: Int): Unit {
    if (x % 2 == 0){
        println("Numero evaluado ${x} es par")
    }
}
```

- Para llamar a una variable en una concatenación podemos utilizar $ ($x), o incluso hacer alguna operación con esa variable (${x*2})

- Si queremos cambiar un tipo a una variable, sólo tenemos que poner un punto y to lo que quieras pasar

```
fun division (x: Int, y: Int): Double {
    return x.toDouble()/y
```

- Las Funciones de Flecha (ver por mi cuenta)

LECTURAS POR TECLADO

- Para leer por teclado podemos utilizar el *readLine ()* o el *readln (),* La diferencia entre ambos es que mientras que readln ( ) devuelve un String, el readLine ( ) devuelve un String? con posibilidad de nulo.
- Si cogemos el readLine, hay que añadirle una interrogación para la posibilidad de nulo, y en ese caso, para que siga ejecutando (ejem .toInt( )) ponemos ?:0, y así tomará ese valor. En la función de ejemplo, si y es nulo, cogerá el valor 0

```
// NOS CREAMOS UNA FUNCIÓN SIN PARAMETROS
fun division2 (): Double {
    println("Introduce el primer parámetro")
    var x: Int = readln().toInt() // poneemos el toInt porque el readLine nos devuelve un String
    println("Introduce el segundo parámetro")
    var y: Int = readLine()?.toInt() ?:0 // ponemos el ?:0 por si el valor es nulo, en vez de dar erro, tome el valor 0
    return x.toDouble()/y
```

NULOS Y EXCEPCIONES

- Si yo quiero trabajar una variable que he declarado nula, puedo poner la alternativa si llega a ser nula con ?: “Lo que quieras decir”, como si fuera una excepción

```
var correo:String? = null
println(correo?.length ?: "No existe el correo")
```

- También podemos darle un valor a nuestra variable nula, en nuestro ejemplo es asignándole a sueldo el valor de 10000 si fuera nulo. Hemos pasado a toDouble. También el casting se puede hacer con as Double

```
// voy a calcular mis pagas salariales
var sueldo: Int? = null
var pagas= sueldo?.toDouble() ?:10000  /14
println(pagas)
```

ARRAYLIST - VAL

- Siempre es preferible utilizar una variable *val* a una *var*, ya que la no mutable es una memoria cerrada ya que no va a cambiar

- A una variable no mutable val no puedo cambiarle el valor, pero si hago un ArrayList puedo ir añadiendo valores, ya que no estoy reasignado, si no que estoy cambiando elementos de su estructura interna

```
// ArrayList y val
   val trabajos: ArrayList<String> = ArrayList()
   trabajos.add("PMDM")
   trabajos.add("DI")
   trabajos.add("PSP")
   trabajos.add("SGE")
```

CLASES

- Las clases hay que meterlas dentro de un **paquete**, por lo que es lo primero que hacemos. Los paquetes lo que hacen es organizan el código para poder estructurar mediante secuenciación de *private*, *protected*, y *public*, los diferentes tipos de acceso
- Como en Java, en una clase nos encontramos con atributos, constructores, métodos, getters y setters
- **ATRIBUTOS**: pueden ser val o var, y siempre hay que darle un valor, no como en java que los podíamos dejar sin inicializar. Por defecto son públicos, y el privado no tiene aquí tanto sentido como en java
- Las variables o atributos los podemos declarar null y luego inicializarlas en el constructor
- **CONSTRUCTOR**: existen dos tipos de constructores: el primario, y los secundarios que pueden ser tantos como yo quiera

    - **PRIMARIO**: es el constructor base con los atributos obligatorios o variables de clase. Se declaran al lado de la definición de la clase

      class Usuario (var nombre: String, var apellido: String, var dni: String){

Ejemplo: nos creamos un usuario para ver el ejemplo del constructor primario:

```
// nos creamos un usuario para ver el ejemplo del constructor de la clase
Usuario
(Dentro del Main)
var usuario1 : Usuario = Usuario("Guada","Vargas","12345678Y")
usuario1.mostrarDatos()
```

```
(Dentro de la clase Usuario)
fun mostrarDatos(): Unit {
println("Los datos son: Nombre: $nombre, Apellido: $apellido, DNI: $dni)
```

- **SECUNDARIOS**: es el constructor base más lo adicional. Para crearlo sería con la palabra reservada constructor(los atributos de clase + el atributo opcional), y la llamada al constructor primario, con dos puntos: this (atributos primarios)

```
// secundarios --> base + adicional

var correo: String? = null // el atributo correo es opcional

// creo el constructor secundario

constructor(nombre: String, apellido: String, dni: String, correo:
String): this (nombre, apellido, dni){

    this.correo = correo
}
```

Otro ejemplo de constructor secundario:

```
// Me creo otra variable opcional y lo igualo a nulo

var telefono: Int? = null

// Me creo otro constructor

constructor(nombre: String, apellido: String, dni: String, telefono:
Int): this (nombre, apellido, dni){

    this.telefono = telefono
}
```

Init: es parte del código ejecutado siempre después del constructor. Es para hacer inicializaciones por defecto de variables. Si hay algo no inicializado, se inicializará de alguna forma. En Android no tiene mucho sentido, utilizaremos más el ***lateinit***, que significa que la variable te la iniciará en algún momento

- **GETTERS Y SETTERS**: van implícitos, obtienen el valor y cambian el valor

```
// getters y setters van implícitos

usuario1.nombre = "Sergio"  // --> esto es un setter

usuario1.nombre // --> esto es un getter
```

Si quieres modificar los getters y setters los escribes con el *set (value)* y el *get* , haces lo que quieras con ellos, pero si quieres que funcionen de la misma forma ni siquiera los tienes que escribir.

```
// Se pueden modificar con el set (value)

var correo2: String? = null
```

```
    set(value) {
        it = value  // el it hace referencia al correo
    }
```

```
// Modificar el get
var correo3: String? = null
    get() {
        return "asdaas"
    }
```

HERENCIAS

- Sólo puedes heredar de una clase
- Cuando queremos que una clase herede de otra, o implemente de otra, sólo hay que poner después del nombre de la clase dos puntos : y el nombre de la clase. Si tiene algún atributo nuevo lo añadimos (en nuestro ejemplo salario).

```
// Creo una clase Trabajador que quiero que herede los atributos de Usuario y
además un salario
class Trabajador(nombre: String, apellido: String, dni: String, telefono: Int,
var salario: Int) :
    Usuario(nombre, apellido, dni, telefono) {
}
```

- Para heredar de una clase ésta tiene que estar abierta, por lo que al nombre de la clase de la que hereda tiene que tener la palabra reservada *open*

```
open class Usuario (var nombre: String, var apellido: String, var dni:
String){
```

- Si yo tuviera un método en la clase heredada, también lo tengo que abrir con ***open*** y sobrescribirla en nuestra nueva clase con @override

```
// métodos, le añado open para que se pueda sobrescribir en la que hereda
open fun mostrarDatos(): Unit {
    println("Los datos son: Nombre: $nombre, Apellido: $apellido, DNI: $dni")
}
```

```
  // Método que hereda de Usuario, al que le añado el nuevo atributo salario
override fun mostrarDatos() {
    super.mostrarDatos()
        println("Salario: $salario")
}
```

Y ya podemos crear en nuestro main al nuevo objeto trabajador y que lo imprima con mostrarDatos( )

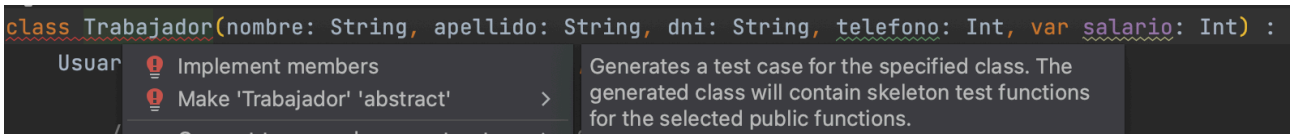
```
var trabajador : Trabajador = Trabajador ("Borja", "Martín", "12658Y", 6857455,
5000)

trabajador.mostrarDatos()
```

INTERFACES

- Igual que en las herencias
- Es una clase que únicamente tiene métodos abstractos
- Sólo hay que poner una coma y el nombre de la interfaz

```
class Trabajador(nombre: String, apellido: String, dni: String, telefono: Int, var salario: Int) :
    Usuario(nombre, apellido, dni, telefono), Votante {
```

- Cuando una clase implementa una interfaz me va a obligar o bien a sobrescribir el método abstracto, o a convertir la clase en abstracta



- Creo en el main un trabajador2 de tipo votante, en este caso el único método que puede utilizar es votar, pero no puede mostrarDatos ( )

```
var trabajador2: Votante = Trabajador ("Borja", "Martín", "12658Y", 6857455,
5000)

println (trabajador2.votar())

// trabajador2.mostrarDatos() da error porque un trabajador que sea de la clase
Votante sólo puede votar
```