Tema 3. Introducción a Kotlin

Tema 3. Introducción a Kotlin

- 1. Sintaxis básica
- 2. Elementos del lenguaje
  - 2.1 Comentarios
  - 2.2 Variables e identificadores
  - 2.3 Tipos de datos
    - 2.3.1 Primitivos
    - 2.3.2 Estructurados/complejos (Objetos)
  - 2.4 Operadores
- 3. Funciones
- 4. Control de flujo
  - 4.1 When
  - 4.2 for
  - 4.3 while y do-while
  - 4.4 Iterator

Tema 3. Introducción a Kotlin

- 5. Clases
  - 5.1 Clase Pair
- 6. Enumerados
- 7. Colecciones
- 8. Herencia

1. Sintaxis básica

JAVA

```
public class Main
{
    public static void main(String[] args)
    {
        System.out.println("Hola Mundo");
    }
}
```

KOTLIN

```
fun main()
{
        println("Hola Mundo")
}
```

2.1 Comentarios

JAVA

```
// Comentario de una línea

/* Comentario de varias líneas */

/** Javadoc */
```

KOTLIN

```
// Comentario de una línea

/* Comentario de varias líneas */

/** KDoc */
```

2.2 Variables e identificadores

JAVA

Declaración de una variable

int x;

double temperatura;

char letra;

KOTLIN

Declaración de una variable

x: Int

temperatura: Double

letra: Char

2.2 Variables e identificadores

JAVA

Inicialización de una variable

x = 2;

temperatura = 23.4;

letra = ‘c’;

KOTLIN

Inicialización de una variable

x = 2

temperatura = 23.4

letra = ‘c’

2.2 Variables e identificadores

JAVA

Declaración e Inicialización de una variable en la misma línea

```
int x = 2;

double temperatura = 23.4;

char letra = ‘c’;
```

KOTLIN

Declaración e Inicialización de una variable en la misma línea

```
x : Int = 2

temperatura : Double = 23.4

letra : Char = ‘c’
```

2.2 Variables e identificadores

JAVA

- *Es **IMPOSIBLE** usar una variable que no haya sido declarada previamente.*
- *Las constantes se definen con el calificador **final**.*

  ***final** int x = 2;*

KOTLIN

- *Es **POSIBLE** usar variables sin declararlas previamente. El compilador asigna el tipo a partir del valor del literal.*

  ***val** c = 3*

- *Las constantes se definen con el calificador **val**. Las variables que puedan ser modificadas a lo largo del programa deben llevar el calificador **var**.*

2.2 Variables e identificadores

JAVA

- *Es **POSIBLE** mostrar el valor de una variable que no haya sido inicializada previamente.*
  - *Las variables **númericas** las inicializa a 0*
  - *Las variables de **caracteres**, las inicializa con \0*
  - *Las variables tipo **boolean** java las inicializa como false.*
  - *Las **referencias** a cadenas de caracteres y a **objetos** las inicializa con null.*

KOTLIN

- *Es **IMPOSIBLE** mostrar el valor de una variable que no haya sido inicializada previamente.*

  var e: Int

  println(e)

compiler error: `Variable 'e' must be initialized`.

2.2 Variables e identificadores

KOTLIN

```
val d: Int

if (someCondition())
        {d = 1 }
 else
        { d = 2 }

println(d)
```

- *La variable se declara pero no se inicializa*
- *La variable es inicializada, tanto si cumple la condición como si no*
- *Correcto, porque la variable es inicializada en cualquier caso.*

2.2 Variables e identificadores - Null safety

KOTLIN

```
var neverNull: String = "This can't be null"

neverNull = null

var nullable: String? = "You can keep a null here"

nullable = null
```

- Declares a non-`null` String variable.
- When trying to assign `null` to non-nullable variable, a compilation error is produced.
- Declares a nullable String variable.
- Sets the `null` value to the nullable variable. This is **OK**.

2.2 Variables e identificadores - late init

KOTLIN

```
public class MyTest

{ lateinit var subject: TestSubject


@SetUp fun setup()

{ subject = TestSubject() }


@Test fun test()

{ subject.method() // dereference directly
} }
```

Late-initialized properties and variables

Normally, properties declared as having a non-nullable type must be initialized in the constructor.

However, it is often the case that doing so is not convenient.

2.2 Variables e identificadores - Lazy

KOTLIN

```
val adapter: EventAdapter by lazy
{ initializeAdapter() }
```

Initialization by Lazy

El código a continuación corresponde a la inicialización de la variable adapter pero esta solo se realizará durante la primera vez en que se haga uso de la variable adapter.

2.3 Tipos de datos

JAVA

Distingue entre:

- *Tipos primitivos*
- *Tipos complejos (Objetos)*

KOTLIN

- En el lenguaje de programación **Kotlin** TODO es un objeto.

2.3.1 Tipos de datos primitivos

JAVA

Palabra Reservada en JAVA	Descripción	Tamaño
byte	Entero c/longitud byte	8-bit
short	Entero corto	16-bit
int	Integer	32-bit
long	Entero largo	64-bit
float	Punto flotante simple p.	32-bit
double	Punto flotante doble p.	64-bit
char	Un carácter simple	16-bit
boolean	Un carácter booleano	true o false

2.3.2 Objetos

JAVA

- *Wrappers*

Para todos los **tipos de datos primitivos**, existen unas clases llamadas **Wrapper**, también conocidas como envoltorio, ya que **proveen una serie de mecanismos que nos permiten envolver a un tipo de dato primitivo permitiéndonos con ello el tratarlos como si fueran objetos.**

Tipos primitivos (no son objetos y por tanto no poseen métodos)	Wrappers(son objeto y por tanto poseen métodos)
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

2.3.2 Objetos

KOTLIN

Existen **5 tipos básicos** en lenguajes de programacion kotlin, que son:

- *Char.*
- *Numbers.*
- *Boolean.*
- *Strings.*
- *Array.*

Numbers

En el caso del tipo *numbers*, cabe especificar que existe una clasificación interna. La cual consiste en 6 tipos de números en Kotlin:

- *Double.*
- *Float.*
- *Long.*
- *Int.*
- *Short.*
- *Byte.*

2.3.2 Objetos

KOTLIN

- **Any**

Es la clase **raíz** en la jerarquía de clases de **Kotlin**. Todas las clases en Kotlin, ya sean clases de usuario o las propias clases de la biblioteca estándar de Kotlin, heredan directa o indirectamente de Any. Esto significa que **Any** es una superclase común para todas las clases en Kotlin, similar a **Object** en Java.

La clase **Any** define algunos métodos comunes que están disponibles para todas las clases en Kotlin. Algunos de estos métodos incluyen:

- **equals(other: Any?): Boolean**
- **hashCode(): Int**
- **toString(): String**
- **javaClass: Class<out Any?>**: Esta propiedad permite acceder a la clase de tiempo de ejecución del objeto. Puede ser útil cuando se necesita realizar reflexión en objetos.

2.3.2 Objetos

KOTLIN

```
fun main() {
val cadena = "Hola, mundo!"
val numero = 42
val lista = listOf(1, 2, 3)

println("Clase de cadena: ${cadena.javaClass}")
println("Clase de numero: ${numero.javaClass}")
println("Clase de lista: ${lista.javaClass}")

}
```

- La salida del programa sería:

```
Clase de cadena: class kotlin.String
Clase de numero: class kotlin.Int
Clase de lista: class java.util.Arrays$ArrayList
```

2.3.2 Objetos - Array

KOTLIN

https://www.develou.com/arrays-en-kotlin/

https://kotlindoc.blogspot.com/2019/05/arrays-en-kotlin.html

En Kotlin un *Array* es una **clase con una propiedad** que define su tamaño (el número de elementos) **y con funciones get y set** y algunos otros métodos.

2.4 Operadores

KOTLIN

https://www.develou.com/operadores-en-kotlin/

Operadores Aritméticos

Te permiten expresar operaciones aritméticas entre dos operandos.

Operador	Operación	Expresión	Función Equivalente
`+`	Suma	`a+b`	`a.plus(b)`
`-`	Resta	`a-b`	`a.minus(b)`
`*`	Multiplicación	`a*b`	`a.times(b)`
`/`	División	`a/b`	`a.div(b)`
`%`	Residuo	`a%b`	`a.rem(b)`

Operadores Relacionales

Los operadores relacionales te permiten verificar enunciados de igualdad y desigualdad entre dos valores. El tipo de dato resultante de la expresión es `Boolean`, indicando la veracidad del enunciado expresado.

Aquí hay una la tabla de estos operadores:

Operador	Enunciado	Expresión	Función Equivalente
`==`	*a* es igual a *b*	`a==b`	`a?.equals(b) ?: (b === null)`
`!=`	*a* es diferente de *b*	`a!=b`	`!(a?.equals(b) ?: (b === null))`
`<`	*a* es menor que *b*	`a<b`	`a.compareTo(b)<0`
`>`	*a* es mayor que *b*	`a>b`	`a.compareTo(b)>0`
`<=`	*a* es menor ó igual que *b*	`a<=b`	`a.compareTo(b) <=0`
`>=`	*a* es mayor o igual que *b*	`a>=b`	`a.compareTo(b)>= 0`

3. Funciones

KOTLIN

```
fun printMessage (message: String): Unit

{     println(message)  } //1
```

```
fun printMessageWithPrefix (message: String, prefix: String = "Info")

{   println("[$prefix] $message") } //2
```

1. Función con un parámetro de tipo **String**. Cuando el tipo devuelto es **Unit**, significa que no devuelve nada (**void** en Java).

2. Función con dos parámetros de tipo **String**. El parámetro *prefix* tiene un *valor por defecto* preasignado. No devuelve nada, por eso no hay nada escrito a continuación del paréntesis.

3. Funciones

KOTLIN

```
fun sum(x: Int, y: Int): Int
{
  return x + y
}
    //3
```

```
fun multiply(x: Int, y: Int) = x * y


//4
```

3. Función con dos parámetros de tipo **Int**. El valor devuelto es también de tipo **Int**. Para devolver el valor, se usa la instrucción **return**.

4. Función con dos parámetros de tipo **Int**. En lugar de especificar el tipo devuelto, se indica cómo se calcula el resultado de la función. La instrucción **return** es omitida.

3. Funciones

KOTLIN

```
fun main()
 {
        printMessage("Hello")                                   // 5
        printMessageWithPrefix("Hello", "Log")                  // 6
        printMessageWithPrefix("Hello")                         // 7
        printMessageWithPrefix(prefix = "Log", message = "Hello")   // 8
        println(sum(1, 2))                                      // 9
        println(multiply(2, 4))                                 // 10
}
```

7. La función ha sido declarada con dos atributos, pero la llamada puede hacerse solo con uno, al haber declarado un valor por defecto.

8. La función es llamada, colocando los parámetros en un orden distinto al que fueron declarados.

4. Control de flujo - when

KOTLIN

```kotlin
fun main() {
    cases("Hello")
    cases(1)
    cases(0L)
    cases(MyClass())
    cases("hello")
}


fun cases(obj: Any) {
    when (obj) {                          // 1
        1 -> println("One")                 // 2
        "Hello" -> println("Greeting")          // 3
        is Long -> println("Long")              // 4
        !is String -> println("Not a string")     // 5
        else -> println("Unknown")              // 6
    }
}
```

1. This is a `when` statement.
2. Checks whether `obj` equals to `1`.
3. Checks whether `obj` equals to `"Hello"`.
4. Performs type checking.
5. Performs inverse type checking.
6. Default statement (might be omitted).

4. Control de flujo - when

KOTLIN

```
fun main() {
    println(whenAssign("Hello"))
    println(whenAssign(3.4))
    println(whenAssign(1))
    println(whenAssign(MyClass()))
}

fun whenAssign(obj: Any): Any {
    val result = when (obj) {                    // 1
        1 -> "one"                               // 2
        "Hello" -> 1                             // 3
        is Long -> false                           // 4
        else -> 42                               // 5
    }
    return result
}
```

1. This is a `when` expression.
2. Sets the value to `"one"` if `obj` equals to `1`.
3. Sets the value to one if `obj` equals to `"Hello"`.
4. Sets the value to `false` if `obj` is an instance of `Long`.
5. Sets the value `42` if none of the previous conditions are satisfied. Unlike in `when` *statement*, the default branch is usually required in `when` *expression*, except the case when the compiler can check that other branches cover all possible cases.

4. Control de flujo - for

KOTLIN

```
val cakes = listOf("carrot", "cheese", "chocolate")

for (cake in cakes) {                    // 1
    println("Yummy, it's a $cake cake!")
}
```

1. Se crea una *constante* de elementos de tipo *String*.
2. **for-in** similar al de java

4. Control de flujo - while y do-while

KOTLIN

```
fun eatACake() = println("Eat a Cake")
fun bakeACake() = println("Bake a Cake")

fun main(args: Array<String>) {
    var cakesEaten = 0
    var cakesBaked = 0

    while (cakesEaten < 5) {                    // 1
        eatACake()
        cakesEaten ++
    }

    do {                                        // 2
        bakeACake()
        cakesBaked++
    } while (cakesBaked < cakesEaten)

}
```

Se han creado dos funciones, que simplemente imprimer un texto por pantalla.

1. **while** (similar a java)
2. **do-while** (similar a java)

4. Control de flujo - Iterator

KOTLIN

```
val numbers = listOf("one", "two", "three", "four")

val numbersIterator = numbers.iterator()

while (numbersIterator.hasNext())
{
    println(numbersIterator.next())
}
```

1. Se crea una constante que es una lista de elementos de tipo String
2. Se crea un iterador
3. Recorremos los elementos de la lista usando el iterador.

4. Control de flujo - Iterator

KOTLIN

```
class Animal(val name: String)

class Zoo(val animals: List<Animal>) {

    operator fun iterator(): Iterator<Animal> {     // 1
        return animals.iterator()                   // 2
    }
}

fun main() {

    val zoo = Zoo(listOf(Animal("zebra"),
Animal("lion")))

    for (animal in zoo) {                           // 3
        println("Watch out, it's a ${animal.name}")
    }

}
```

1. Defines an iterator in a class. It must be named `iterator` and have the `operator` modifier.

2. Returns the iterator that meets the following method requirements:
   - `next(): Animal`
   - `hasNext(): Boolean`

3. Loops through animals in the zoo with the user-defined iterator.

5. Clases - Ejemplo

KOTLIN

https://kotlinlang.org/docs/classes.html

```
class Punto2D(val x: Int, val y: Int)
{
    constructor() : this(0, 0) // Constructor secundario sin parámetros que inicializa a (0, 0)
    constructor(punto: Punto2D) : this(punto.x, punto.y) // Constructor secundario de copia

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Punto2D) return false
        return this.x == other.x && this.y == other.y
    }

    override fun hashCode(): Int {
        var result = x
        result = 31 * result + y
        return result}

    override fun toString(): String {
        return "Punto2D(x=$x, y=$y)"
    }
}
```

5. Clases - Asignar valor por defecto en el constructor

KOTLIN

```
class Punto2D(val x: Int = 0, val y: Int = 0)
{
    constructor() : this(0, 0) // Constructor secundario sin parámetros que inicializa a (0, 0)
    constructor(punto: Punto2D) : this(punto.x, punto.y) // Constructor secundario de copia

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Punto2D) return false
        return this.x == other.x && this.y == other.y
    }

    override fun hashCode(): Int {
        var result = x
        result = 31 * result + y
        return result}

    override fun toString(): String {
        return "Punto2D(x=$x, y=$y)"
    }
}
```

5. Clases - Bloque init

KOTLIN

https://www.delftstack.com/es/howto/kotlin/kotlin-init/?utm_content=cmp-true

```
class Punto2D(val x: Int-= 0, val y: Int = 0)
{

    init {
        require (x>0)
    }

    constructor(punto: Punto2D) : this(punto.x, punto.y)
// Constructor secundario de copia

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Punto2D) return false
        return this.x == other.x && this.y == other.y
    }
```

```
    override fun hashCode(): Int {
        var result = x
        result = 31 * result + y
        return result}

    override fun toString(): String {
        return "Punto2D(x=$x, y=$y)"
    }
}

fun main() {
    val p1: Punto2D
    val p2: Punto2D

    p1 = Punto2D(1,2)
    p2 = Punto2D() // ¡ERROR!

    println(p1)
    println(p2)

}
```

5. Clases - Bloque init

KOTLIN

https://www.delftstack.com/es/howto/kotlin/kotlin-init/?utm_content=cmp-true

```
class Punto2D(val x: Int = 0, val y: Int = 0)
{

    init {
      require(x>0)
    }

    constructor(punto: Punto2D) : this(punto.x, punto.y)
// Constructor secundario de copia

 constructor() : this(0,0) // Constructor sin parámetros

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Punto2D) return false
        return this.x == other.x && this.y == other.y
    }
```

```
    override fun hashCode(): Int {
        var result = x
        result = 31 * result + y
        return result}

    override fun toString(): String {
        return "Punto2D(x=$x, y=$y)"
    }
}

fun main() {
    val p1: Punto2D
    val p2: Punto2D

    p1 = Punto2D(1,2)
    p2 = Punto2D()  // CORRECTO

    println(p1)
    println(p2)

}
```

5. Clases - Data class

KOTLIN

```
class Punto2D(val x: Int, val y: Int)
{
    constructor(punto: Punto2D) : this(punto.x, punto.y)
    constructor() : this(0,0)

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Punto2D) return false
        return this.x == other.x && this.y == other.y}

     override fun hashCode(): Int {
        var result = x
        result = 31 * result + y
        return result}

    override fun toString(): String {
        return "Punto2D(x=$x, y=$y)"}

    fun copy(x: Int = this.x, y: Int = this.y) = Punto2D(x,y)
}
```

```
data class Punto2D(val x: Int, val y: Int)
{
    constructor(punto: Punto2D) : this(punto.x, punto.y)
    constructor() : this(0,0)
}
```

En una **data class,** Kotlin automáticamente genera los métodos ***equals***, ***hashCode***, ***toString*** y ***copy*** basados en los atributos de la clase, lo que simplifica la definición y uso de clases que principalmente almacenan datos. Además, los atributos x e y se mantienen como solo lectura (valores inmutables) en una data class.

5. Clases - Método copy

KOTLIN

```
data class Persona(val nombre: String, val edad: Int)

fun main()
{
    val personaOriginal = Persona("Juan", 30)

    // Crear una copia de personaOriginal con un cambio en la edad

    val personaModificada = personaOriginal.copy(edad = 31)

    // Imprimir los objetos originales y la copia

    println("Persona original: $personaOriginal")
    println("Persona modificada: $personaModificada")
}
```

Método **copy** en Kotlin:

- El método **copy** es específico de las **data class** en Kotlin. En una data class, Kotlin genera automáticamente un método **copy** que te permite crear una copia del objeto con la opción de modificar algunos de sus atributos mientras mantienes otros sin cambios.
- El método copy en Kotlin es una forma conveniente de **crear copias de objetos inmutables** y generalmente se usa en ese contexto.
- El método copy en Kotlin es altamente personalizable y *te permite especificar los valores que deseas cambiar al crear la copia*.

5. Clases - Método equals y operador ==

KOTLIN

```
data class Punto2D(val x: Int, val y: Int)

fun main()
{
    val punto1 = Punto2D(2, 3)
    val punto2 = Punto2D()
    val punto3 = punto1.copy()

    constructor() : this(0,0)

    // Comparación de puntos utilizando equals (función generada automáticamente)
    println("punto1 == punto2: ${punto1 == punto2}")
    println("punto1 == punto3:${punto1.equals(punto3)}")

}
```

```
punto1 == punto2: false
punto1 == punto3: true
```

Método **equals** en Kotlin:

- Cuando utilizamos el operador ***"=="*** en Java, devuelve **true**, sólo si los objetos a ambos lados de la comparación, comparten la misma referencia.
- En Kotlin, se hace una llamada al método **equals**.
- ***==, !=*** *- equality operators (translated to calls of equals() for non-primitive types).*

5. Clases - Visibility

KOTLIN

```
class Persona (private var nombre: String = "", private var edad: Int = 0)
{
    // Getter público para el nombre
    fun getNombre(): String {
        return nombre
    }

    // Setter público para el nombre
    fun setNombre(nuevoNombre: String) {
        nombre = nuevoNombre
    }

    // Getter público para la edad
    fun getEdad(): Int {
        return edad
    }

    // Setter público para la edad
    fun setEdad(nuevaEdad: Int) {
        if (nuevaEdad >= 0) {
            edad = nuevaEdad
        }
    }
}
```

```
fun main()
{
    val p1: Persona
    val p2: Persona

    p1 = Persona("Lucas", 25)
    p2 = Persona("Bill Gates", 68)


  println("Nombre de la persona 1: ${p1.getNombre()}")

  println("Edad de la persona 2: ${p2.getEdad()}")

}
```

```
Nombre de la persona 1: Lucas
Edad de la persona 2: 68
```

5. Clases - Atributos y métodos estáticos

KOTLIN

https://play.kotlinlang.org/byExample/03_special_classes/04_Object

```kotlin
object MiObjetoSingleton {
    val atributo: Int = 42

    fun metodo() {
        println("Este es un método en el objeto singleton")
    }
}

fun main() {
    println(MiObjetoSingleton.atributo)
    MiObjetoSingleton.metodo()
}
```

Objetos Singleton

- En lugar de utilizar clases estáticas, Kotlin permite la creación de objetos singleton mediante la palabra clave object. Estos objetos singleton pueden contener propiedades, métodos y funciones, y se comportan de manera similar a las clases estáticas en otros lenguajes.

5. Clases - Atributos y métodos estáticos

KOTLIN

```kotlin
class Ejemplo {
    companion object {
        val atributo: Int = 42

        fun metodo() {
            println("Este es un método en el objeto de compañía")
        }
    }
}

fun main() {
    println(Ejemplo.atributo)
    Ejemplo.metodo()
}
```

Companion Objects

- Kotlin también permite la creación de objetos de compañía (***companion objects)*** dentro de las clases. Estos objetos de compañía pueden tener propiedades y métodos que son compartidos por todas las instancias de la clase, lo que a menudo se asemeja a la funcionalidad de atributos y métodos estáticos.

5.1 Clases - Clase Pair

KOTLIN

```kotlin
fun main() {
    // Crear un par de valores
    val miPair: Pair<String, Int> = Pair("Manzana", 5)

    // Acceder a los valores del par
    val fruta = miPair.first
    val cantidad = miPair.second

    println("Fruta: $fruta, Cantidad: $cantidad")

    // También puedes usar destructuring para acceder a los valores
    val (fruta2, cantidad2) = miPair
    println("Fruta: $fruta2, Cantidad: $cantidad2")
}
```

Clase Pair

- En Kotlin, la clase **Pair** es una clase de la biblioteca estándar que se utiliza para representar un par ordenado de dos elementos.
- Cada objeto **Pair** contiene exactamente dos valores, uno llamado ***first*** (primero) y el otro llamado ***second*** (segundo).
- Puedes usar la clase **Pair** para combinar dos valores de tipos diferentes o iguales en una estructura de datos simple.

6. Enumerados

KOTLIN

```
enum class State {
    IDLE, RUNNING, FINISHED                    // 1
}

fun main() {
    val state = State.RUNNING                  // 2
    val message = when (state) {               // 3
        State.IDLE -> "It's idle"
        State.RUNNING -> "It's running"
        State.FINISHED -> "It's finished"
    }
    println(message)
}
```

// Este es el uso de los enumerados en Java

1. Define una clase **enum** simple con tres constantes enum. El número de constantes es siempre finito y todas son distintas.

2. Accede a una constante enum a través del **nombre** de la clase.

3. Con los **enums**, el compilador puede inferir si una expresión when es exhaustiva, por lo que no necesitas el caso else.

6. Enumerados

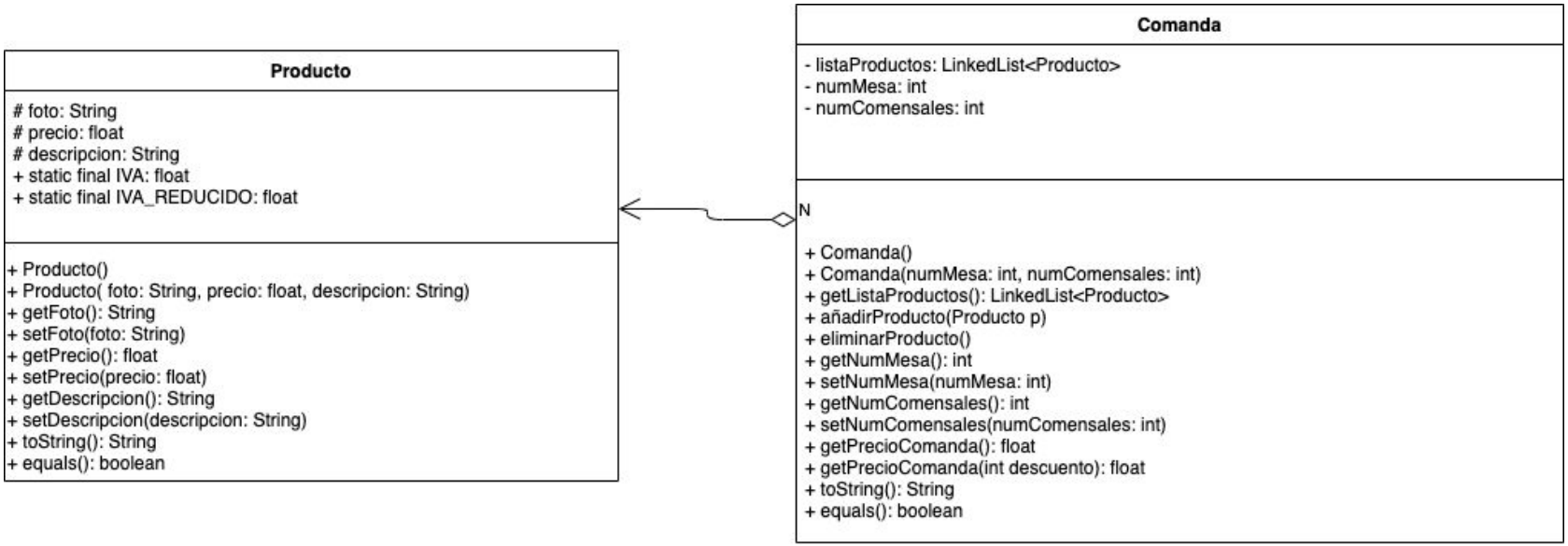
KOTLIN

```
enum class Color(val rgb: Int) {                    // 1
    RED(0xFF0000),                                  // 2
    GREEN(0x00FF00),
    BLUE(0x0000FF),
    YELLOW(0xFFFF00);

 fun containsRed() = ((this.rgb and 0xFF0000) != 0) // 3

}


fun main() {
    val red = Color.RED
    println(red)                                    // 4
    println(red.containsRed())                      // 5
    println(Color.BLUE.containsRed())               // 6
    println(Color.YELLOW.containsRed()) //7
}
```

1. *Este uso es solo de **Kotlin**. Podemos definir enumerados como si fuesen una clase, con sus **métodos** y **atributos**.*

Ejercicio

Crea un fichero donde declares estas dos clases. Crea también un método main donde pruebes los diferentes métodos.



7. Colecciones

KOTLIN

- Puedes ver las colecciones en el siguiente enlace:

  https://play.kotlinlang.org/byExample/05_Collections/01_List

7. Colecciones - Listas inmutables

KOTLIN

```
data class Punto2D(val x: Int, val y: Int)

data class Poligono(val puntos:
List<Punto2D>)
{
    init {
        require(puntos.size >= 3) {
            "Un polígono debe tener al menos 3
puntos."
        }
    }

    fun calcularPerimetro(): Double {
        //Calcula el perímetro
    }

    private fun calcularDistancia(punto1:
Punto2D, punto2: Punto2D): Double {
        //Calcula la distancia entre 2 puntos}
```

```
fun main()
{
    val puntosPoligono = listOf(Punto2D(0,0),
Punto2D(0,1), Punto2D(1,0))

    val poligono = Poligono(puntosPoligono)

    println("Polígono: $poligono")
    println("Perimetro:
${poligono.calcularPerimetro()}")

}
```

7. Colecciones - Listas Mutables

KOTLIN

```
data class Punto2D(val x: Int, val y: Int)

data class Poligono(val puntos:
MutableList<Punto2D>)
{
    init {
        require(puntos.size >= 3) {
            "Un polígono debe tener al menos 3
puntos."
        }
    }

    fun calcularPerimetro(): Double {
        //Calcula el perímetro
    }

    private fun calcularDistancia(punto1:
Punto2D, punto2: Punto2D): Double {
        //Calcula la distancia entre 2 puntos}
```

```
fun main()
{
    val puntosPoligono =
mutableListOf(Punto2D(0,0), Punto2D(0,1),
Punto2D(1,0))

    val poligono = Poligono(puntosPoligono)

    //Añadir un elemento
    puntosPoligono.add(Punto2D(1,1))

    println("Polígono: $poligono")
    println("Perimetro:
${poligono.calcularPerimetro()}")

}
```

8. Herencia

KOTLIN

```
//Superclase
open class Figura
{
    open fun area(): Double {
        return 0.0
    }
}

//Subclase
class Circulo(val radio: Double) : Figura()
{
    override fun area(): Double {
        return Math.PI * radio * radio
    }
}
```

1. En la *superclase*, las funciones o propiedades que se pueden sobrescribir deben ser marcadas como **open**. En las *subclases*, los métodos o propiedades que redefinen deben ser marcados como **override**.

2. Cuando una clase hereda de otra, se especifica colocando **:** y a continuación el nombre de la *superclase*.

8. Herencia

KOTLIN

Fuente: https://www.develou.com/

Constructor Primario En Herencia

Si la superclase tiene constructor primario, debes inicializarlo pasando los parámetros en la llamada de la sintaxis de herencia.

```
open class Ancestro(val propiedad:Boolean)
class Descendiente(propiedad: Boolean) : Ancestro(propiedad)
```

Constructor Secundario En Herencia

Si la clase base no tiene constructor primario o deseas realizar la llamada del mismo, desde un constructor secundario de la subclase, entonces usa `constructor` junto a la palabra reservada `super` para la llamada.

Por ejemplo:

```
open class Weapon(val damage: Int, val speed: Double)

class Bow : Weapon {
    constructor(damage: Int, speed: Double) : super(damage, speed)
}
```

La subclase `Bow` inicializa las propiedades de `Weapon` a través de la llamada del constructor primario con `super(damage, speed)`.

Sobrescribir Métodos

Aplicar polimorfismo con la sobrescritura de métodos en Kotlin, requiere habilitar el método con el modificador `open`.

Luego usa el modificador `override` desde el método polifórmico la subclase.

Por ejemplo:

```
open class Character(val name: String) {
    open fun die() = println("Morir")
}

class Mage(name: String) : Character(name) {
    override fun die() = println("Mago muriendo")
}

fun main() {
    val jaina = Mage("Jaina")
    // Sucesos desafortunados...
    jaina.die()
}
```

La clase base `Character` tiene un método llamado `die()` que imprime un mensaje sobre la muerte del personaje.

En la subclase `Mage`, diseñada para los personajes tipos magos, sobrescribimos `die()` con el fin de personalizar el mensaje.

Al crear una instancia de `Mage` y ejecutar el método la salida imprimirá:

```
Mago muriendo
```

8. Herencia

KOTLIN

Sobrescribir Propiedades

Similar a la sobrecritura de métodos, sobrescribir una propiedad requiere usar el modificador `override` sobre la propiedad en la subclase.

Si la propiedad está declarada con `val` en la clase padre, es posible reescribirla con `var` en la clase hija. Sin embargo, lo contrario no es posible.

Por ejemplo:

```
open class BaseItem(val name: String) {
    open var quantity = 1
}

class PopularItem(name: String) : BaseItem(name) {
    override var quantity = 6
}

fun main() {
    // Añadir item regular de orden
    val notebook = BaseItem("Cuaderno")

    // Añadir item popular
    val pencil = PopularItem("Lapicero")

    // Mostrar factura
    println("${notebook.name} x ${notebook.quantity}")
    println("${pencil.name} x ${pencil.quantity}")
}
```

En el anterior código tenemos a la superclase `BaseItem` que especifica los ítems por defecto añadidos a una factura. De ella hereda `PopularItem`, con el fin de dar un tratamiento adicional a los ítems populares.

Como ves, se sobrescribe la propiedad `quantity` en `PopularItem`, donde toma una inicialización de `6`.

Luego creamos dos ítems, `notebook` y `pencil`. Al imprimir sus nombres y valores se muestra el cambio de la herencia:

```
Cuaderno x 1
Lapicero x 6
```

8. Herencia - Clases abstractas

KOTLIN

El Modificador abstract

Una clase abstracta es aquella que está marcada con el modificador `abstract` en su declaración. Esto evita que se creen instancias de la clase, pero no impide que se creen subclases a partir de ella.

```
abstract class ClaseAbstracta{
    // Propiedades regulares
    // Propiedades abstractas
    // Métodos abstractos
    // Métodos regulares
}
```

Los miembros de una clase abstracta también pueden ser marcados como `abstract`, lo que significa que no tendrán implementación, debido a que esta será exigida para las subclases.

```
abstract class ClaseAbstracta {
    abstract val propiedadAbstracta: Int

    abstract fun metodoAbstracto()

    fun metodoNoAbstracto() {
        // Cuerpo
    }
}
```

Las clases que deriven de una clase abstracta deben sobrescribir a los miembros abstractos con el modificador `override`:

```
class Subclase : ClaseAbstracta() {
    override val propiedadAbstracta: Int = 10

    override fun metodoAbstracto() {
        print(propiedadAbstracta)
    }
}
```

Como ves, una clase abstracta no debe ser marcada con `open` para permitir herencia al igual que sus miembros abstractos.

Úsalo solo con los miembros no abstractos si deseas sobrescribirlos.

8. Herencia múltiple - Clase + Interfaz

KOTLIN

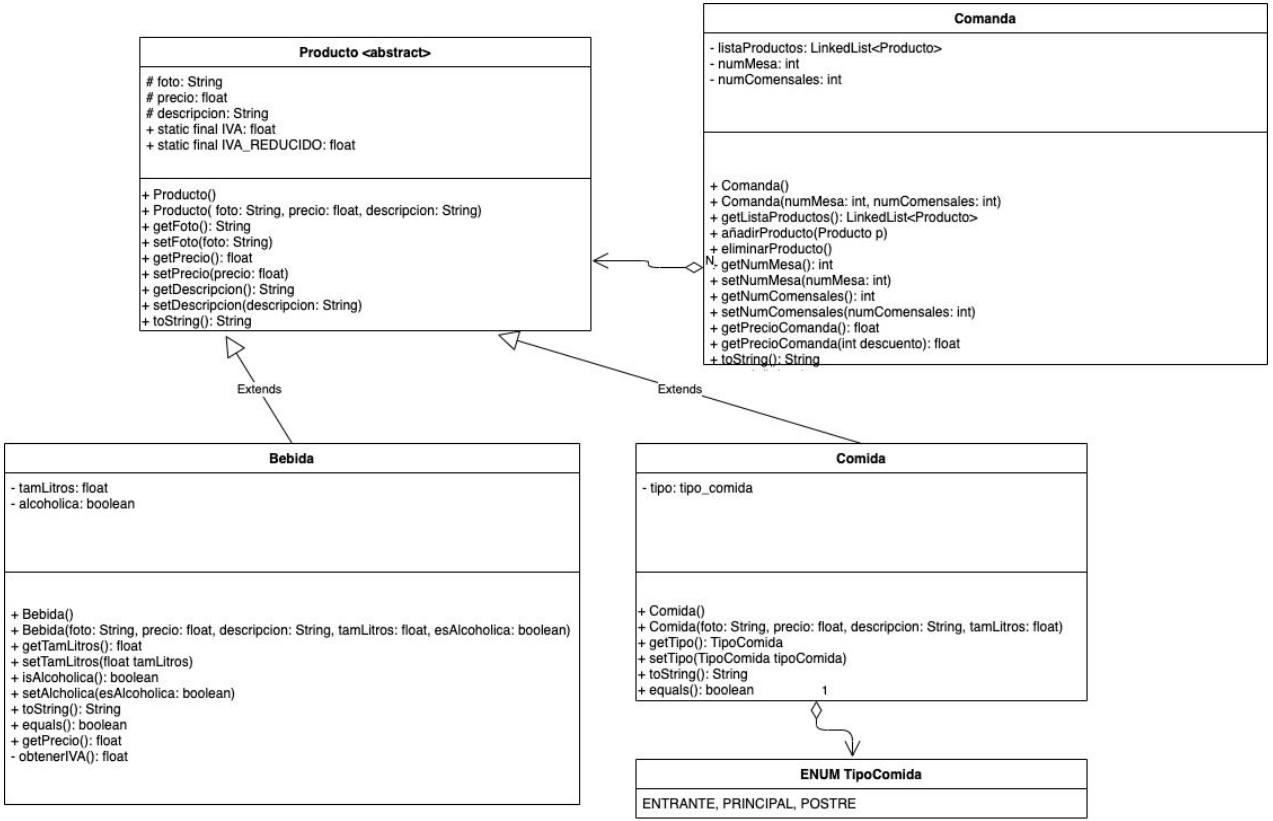
```kotlin
// Clase base
open class Animal(val nombre: String) {
    fun comer() {
        println("$nombre está comiendo.")
    }
}

// Interfaz
interface Nadador {
    fun nadar()
}

// Clase que hereda de una clase base y también implementa una interfaz
class Delfin(nombre: String) : Animal(nombre), Nadador {
    override fun nadar() {
        println("$nombre está nadando en el agua.")
    }
}

fun main() {
    val flipper = Delfin("Flipper")

    flipper.comer()
    flipper.nadar()
}
```

1. En este ejemplo, tenemos una clase base llamada **Animal** que tiene un método *comer()*. Luego, tenemos una interfaz llamada **Nadador** con un método *nadar()*. La clase **Delfin** hereda de la superclase **Animal** y también implementa la interfaz **Nadador**.

2. La clase **Delfin** tiene acceso a los miembros de la superclase Animal, como el *constructor* y el método *comer()*. Además, implementa el método *nadar()* de la interfaz **Nadador**. Esto nos permite combinar una superclase y una interfaz en la misma clase **Delfin**.

3. En el *main()*, hemos creado una *instancia* de **Delfin** llamada flipper y hemos llamado tanto al método heredado *comer()* como al método de la interfaz *nadar()*, demostrando así la combinación de herencia y uso de interfaces en una misma clase.

Ejercicio 2


Producto <abstract>
foto: String
precio: float
descripcion: String
+ static final IVA: float
+ static final IVA_REDUCIDO: float
+ Producto()
+ Producto(foto: String, precio: float, descripcion: String)
+ getFoto(): String
+ setFoto(foto: String)
+ getPrecio(): float
+ setPrecio(precio: float)
+ getDescripcion(): String
+ setDescripcion(descripcion: String)
+ toString(): String
Comanda
- listaProductos: LinkedList<Producto>
- numMesa: int
- numComensales: int
+ Comanda()
+ Comanda(numMesa: int, numComensales: int)
+ getListaProductos(): LinkedList<Producto>
+ añadirProducto(Producto p)
+ eliminarProducto()
+ getNumMesa(): int
+ setNumMesa(numMesa: int)
+ getNumComensales(): int
+ setNumComensales(numComensales: int)
+ getPrecioComanda(): float
+ getPrecioComanda(int descuento): float
+ toString(): String
N
Extends
Extends
Bebida
- tamLitros: float
- alcoholica: boolean
+ Bebida()
+ Bebida(foto: String, precio: float, descripcion: String, tamLitros: float, esAlcoholica: boolean)
+ getTamLitros(): float
+ setTamLitros(float tamLitros)
+ isAlcoholica(): boolean
+ setAlcholica(esAlcoholica: boolean)
+ toString(): String
+ equals(): boolean
+ getPrecio(): float
- obtenerIVA(): float
Comida
- tipo: tipo_comida
+ Comida()
+ Comida(foto: String, precio: float, descripcion: String, tamLitros: float)
+ getTipo(): TipoComida
+ setTipo(TipoComida tipoComida)
+ toString(): String
+ equals(): boolean
1
ENUM TipoComida
ENTRANTE, PRINCIPAL, POSTRE