Programación Multimedia y Dispositivos Móviles

Tema 2. Introducción a Kotlin

Índice

Esquema

Material de estudio

2.1. Introducción y objetivos

2.2. ¿Qué es Kotlin?

2.3. Variables y tipos de datos

2.4. Uso de funciones

2.5. Instrucciones de control

2.6. Clases y objetos

2.7. Comparaciones y elementos nulos

2.8. Referencias bibliográficas

A fondo

Editor online de Kotlin

Kotlin docs

Kotlin en Visual Studio Code

Patrones de diseño en Kotlin

Entrenamientos

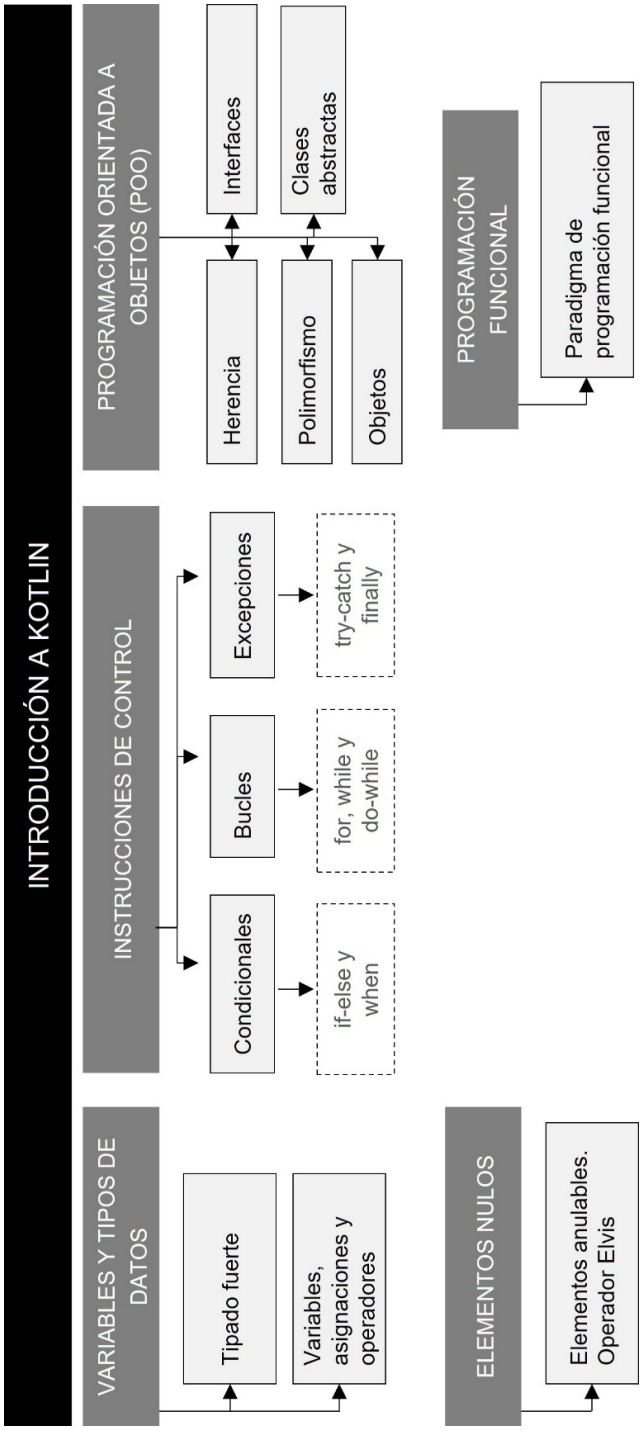
Entrenamiento 1

Entrenamiento 2

Entrenamiento 3

Entrenamiento 4

Entrenamiento 5


INTRODUCCIÓN A KOTLIN
VARIABLES Y TIPOS DE DATOS
Tipado fuerte
Variables, asignaciones y operadores
INSTRUCCIONES DE CONTROL
Condicionales
if-else y when
Bucles
for, while y do-while
Excepciones
try-catch y finally
PROGRAMACIÓN ORIENTADA A OBJETOS (POO)
Herencia
Polimorfismo
Objetos
Interfaces
Clases abstractas
ELEMENTOS NULOS
Elementos anulables. Operador Elvis
PROGRAMACIÓN FUNCIONAL
Paradigma de programación funcional

2.1. Introducción y objetivos

En el transcurso de este tema se realizará una **introducción** a **Kotlin,** el **lenguaje** de programación **recomendado** por **Google** desde el año **2017** para el **desarrollo** de **aplicaciones móviles** con **Android Studio.**

Durante el desarrollo de este tema se deben conseguir los siguientes objetivos:

- Conocer los fundamentos principales de Kotlin.
- Aprender a codificar correctamente, sabiendo qué uso podemos dar a las variables y estructuras de control.
- Aprender qué son y cómo se manejan los operadores.
- Usar la programación orientada a objetos (POO) con Kotlin.

2.2. ¿Qué es Kotlin?

Kotlin es un **lenguaje** de **programación** creado en **2010** por **JetBrains,** empresa desarrolladora de uno de los entornos de desarrollo integrado (IDE) para Java más famosos y utilizado del mundo, intelliJ IDEA. Surge como una **alternativa** al lenguaje de programación **Java** con la intención de **suplir varios** de los **problemas** más **habituales** que las personas programadoras se encontraban en él.

Ventajas

Kotlin aporta ciertas ventajas frente a lenguajes como Java, haciendo los **desarrollos más cómodos:**

- Seguro contra nulos. Como personas programadoras Java uno de los grandes problemas que nos podemos encontrar son los NullPointerException. Sin embargo, Kotlin nos obliga a gestionar en los posibles *null* en tiempo de desarrollo.
- Ahorro de código. Nos permite evitar muchas líneas de código en comparación con otros lenguajes, como Java (p. ej., podríamos realizar el equivalente a un *plain old* Java *objetic* [POJO] en una sola línea en vez de en cincuenta o cien).
- Características de programación funcional. Kotlin está diseñado para que quienes programan puedan trabajar tanto con POO como con programación funcional (e incluso combinarlas). Esto nos proporciona mucha mayor flexibilidad y la posibilidad de usar características (p. ej., *higher-order functions, function types* y lambdas).
- Fácil de usar. Kotlin está inspirado en lenguajes ya existentes, a saber, Java, C# o Scala, por lo que la curva de aprendizaje para quienes programan con estos lenguajes es bastante sencilla.

Material de estudio

En la sección de A fondo encontrarás el editor en línea oficial para programar en Kotlin, un repositorio de GitHub para compilar en Visual Studio Code y patrones de diseño para el lenguaje Kotlin.

2.3. Variables y tipos de datos

Tal como ya sabemos, cada una de las **instrucciones** que forman un **programa** se llama **sentencia;** un **programa** será, entonces, una **lista** de **sentencias.**

Cada sentencia en Kotlin no debe terminar en punto y coma (aunque es muy recomendable por continuidad con otros lenguajes) y, por mejorar la legibilidad del código (salvo excepciones), debe escribirse una sentencia por línea.

Comentarios

Al igual que cualquier lenguaje de programación, Kotlin incluye la **posibilidad** de **añadir comentarios** a su código, que son **caracteres** que **no** se **ejecutarán** y son **útiles** para **documentar** el **código, explicar funcionalidades** a futuras personas desarrolladoras u **ocultar** al intérprete de código **partes** del **programa** mientras se desarrolla.

El **código** debe ser los **suficientemente claro** como para que se pueda **explicar por sí mismo** y, si esto no es posible, se tiene que recurrir a los comentarios. Kotlin tiene **dos maneras** de escribirlos:

- Comentarios de una línea que empiezan con //.
- Comentarios de bloque que empiezan con /* y acaban con */.

```
val a = 2; // Asignación del valor 2 en la variable a

val b = 3; // Asignación del valor 3 en la variable b
```

```
/* Este código muestra la suma de dos números que se han declarado antes.

*/

println(a + b);
```

Tipos de datos

El tipo de dato es un **concepto muy común** en programación (tal como su propio nombre indica) y son las **diferentes clases** que Kotlin puede manejar. Cada uno tiene sus **características** y **operaciones permitidas** (p. ej., se pueden multiplicar dos datos que sean números, pero no las cadenas de texto).

Kotlin maneja los siguientes **tipos** de **datos:**

- Char . Es un tipo de dato alfanumérico en el que se puede guardar un carácter, ya sea de tipo literal (se ponen entre comillas simples) o carácter especial (deben empezar con una barra invertida [\]).
- In t. Tipo de dato numérico que almacena números enteros con signo de 32 bits. Al usar un numero entero por defecto, el compilador entiende que es un Int .
- Long . Tipo de dato numérico que almacena números enteros con signo de 64 bits.
- Short . Tipo de dato numérico que almacena números enteros con signo de 16 bits.
- Byte . Tipo de dato numérico que almacena números enteros con signo de 8 bits.
- Float . Tipo de dato numérico de coma flotante de 32 bits (precisión simple).
- Double . Tipo de dato numérico de coma flotante de 64 bits (doble simple). Al usar un número real o decimal por defecto, el compilador de Kotlin entiende que es un Double .
- Booleano. Pueden tener asignados dos valores true o false .

- String . Tipo de dato alfanumérico, denominado cadena de texto, en el que podemos almacenar la información que desees entre comillas dobles.
- Array. Representa una colección homogénea de elementos (mismo tipo de datos); puede ser mutable (editable) o inmutable (solo lectura).
- Objetos.

Variables

Una variable permite **almacenar valores** para poder **leerse** en otro momento. En Kotlin se puede declarar usando las **palabras reservadas** val o var , pero ¿cuál es la **diferencia** entre estas dos posibilidades?

- val . Se usa cuando se espera que el valor de la variable no cambie, es decir, que sea una constante.
- var . Se usa cuando el valor de la variable puede o debe cambiar, es decir, la variable es mutable.

Con el objetivo de **declarar** una **variable** debemos usar la **palabra clave** val o var , un **identificador** y el **tipo** de **dato.** En los siguientes ejemplos podemos ver la estructura de declaración de una variable mutable y no mutable:

```
val edad: Int = 20; //Valor no mutable

var peso: Double = 73.57; //Valor mutable
```

Sin embargo, las dos líneas de código anteriores se podrían haber escrito de la siguiente manera:

```
val edad = 20; //Valor no mutable

var peso = 73.57; //Valor mutable
```

El compilador de Kotlin sabe que deseas almacenar 20 (un número entero) en la variable edad, de modo que puede inferir que la esta es del tipo `Int` ; lo mismo ocurre con la variable peso, donde infiriere que es del tipo `Double` .

Es importante destacar que, si no se proporciona un valor inicial cuando se declara una variable, es imprescindible especificar su tipo.

```
var altura: Double;
```

E l **identificador** debe ser **único,** es decir, no puede haber dos variables con el mismo identificador/nombre. Conviene utilizar unos **cortos** pero, a la vez, **descriptivos** para hacer la labor de desarrollo más sencilla.

Si bien los *strings* y *arrays* son elementos típicos de otros lenguajes de programación, como Java, JavaScript, C#, etc., vamos a describir sus propiedades principales.

Strings

Aunque aún no hemos visto cómo manejar clases y objetos en Kotlin, el concepto de clase es algo conocido en este momento. La clase `String` es la **encargada** de **tratar** el **texto plano** en nuestros programas, de tal forma que cada uno de sus **literales** se crea como una **instancia** de esta, por lo que solo tenemos que usar las **comillas dobles** para **encerrar** el **texto deseado.**

Kotlin nos permite el **uso** de **literales** de String que tienen **más** de **una línea** de texto, interpretándolas en su forma plana. Para ello hemos de usar la **sintaxis** de **triple comillas dobles** (“””):

```
// raw string

val textoPrueba = """

¡Esto es un texto de prueba para multiples lineas!

En este tema aprenderemos las bases de Kotlin

Para posteriormente usarlo con Android Studio

y crear apps inreibles.
```

Arrays

Tal como ya conocemos de otros lenguajes de programación, es una **estructura** de **datos almacenados** de **forma contigua** donde **todos** los **elementos** se encuentran **referenciados** por un **mismo identificador** y **tipo** de **dato.** Estos se indexan empezando por el **valor 0** y el **tamaño** del ***array,*** valor que es fijo y se define al crearlo. Kotlin usa la clase genérica `Array<T>` .

Para **crear instancias** con un **tipo parametrizado** podemos usar los siguientes **métodos:**

- `arrayOf(vararg elements:T)` . Recibe un argumento variable con elementos de tipo `T` y retorna el arreglo que los contiene.
- `arrayOfNulls(size:Int)` . Crea un *array* de tamaño `size` con elementos de tipo `T` e inicializa los valores con `null` .
- `emptyArray()` . Crea un arreglo vacío con el tipo T.

Sin embargo, si queremos crear un *array* con un **tamaño especifico** y **calcular** todos sus **elementos** a partir de una **función,** debemos usar el constructor Array(size, init) .

Operadores

Los operadores permiten hacer diferentes **operaciones** con **variables.**

Operadores aritméticos

Hacen posible realizar operaciones aritméticas básicas. Algunos son estos:

- +. Suma (o concatenación para *strings*).
- -. Resta.
- *. Multiplicación.
- /. División.
- %. Resto. Devuelve el resto de una división entera.

Operadores de comparación

Estos operadores permiten **comparar dos expresiones.** Kotlin hará lo posible por compararlas y devolver un **valor booleano** (true o false).

- ==. Igual.
- !=. Distinto.
- >. Mayor que.
- <. Menor que.
- >=. Mayor o igual.
- <=. Menor o igual.

Operadores lógicos

Estos operadores permiten realizar operaciones lógicas. Kotlin hará lo posible por **comparar** los **valores** y **realizar** la **operación lógica** entre ellos:

- and: &&. Operador lógico.
- or: ||. Operador lógico.
- not: !. Operador lógico.

Operadores de asignación

Con un operador de asignación se **asigna** el **valor** de la **derecha** en el de la **izquierda** (p. ej., x = 3 se asigna el valor 3 a x). Son los siguientes:

- =:. Asignación simple (x = 3).
- +=:. Asignación de suma (x += 3 es lo mismo que x = x + 3).
- -=:. Asignación de resta (x -= 3 es lo mismo que x = x – 3).

En la sección de A fondo localizarás documentación sobre Kotlin para manejar datos.

2.4. Uso de funciones

Kotlin es un lenguaje de programación que **implementa** el **estilo** de **programación funcional** (y POO); por lo tanto, la función es una herramienta fundamental y muy importante en él.

Una función no es más que un conjunto de instrucciones que realizan una tarea o funcionalidad específica, agrupadas en un bloque de programación.

Se puede **llamar** a una función desde **cualquier zona** del **programa** donde sea necesario. Para definirla debemos usar la **palabra clave** `fun`, seguido del **identificador** y los **parámetros.** Veamos el siguiente ejemplo:

```
fun square (x: Int): Int {

    return x * x;

}
```

- Identificador de la función. Es el nombre que escogemos y debe ser único.
- Lista de parámetros. Son los datos que recibe la función; es necesario definir su tipo y se separan por comas.
- Tipo de retorno. Es el tipo de salida de la función.
- Cuerpo de la función. Son las instrucciones necesarias para realizar la funcionalidad que tiene definida.

Si una **función devuelve** únicamente una **expresión,** podemos **reducir** su **sintaxis** a una **función** con **cuerpo** de **expresión.** La expresión se coloca después del operador igual en la misma cabecera de la función.

```
fun equation(x:Int, y:Int, z:Int) = 5 * x – 3 * y + 7 * z;
```

Si nos fijamos bien, no hemos definido su tipo de dato de salida. Esto se debe a que, al usar **funciones** con **cuerpo** de **expresión,** el **compilador** de Kotlin puede **inferir** los **tipos directamente.** En caso de querer definirlos, sería así:

```
fun equation(x:Int, y:Int, z:Int): Int = 5 * x – 3 * y + 7 * z;
```

Retorno tipo Unit

Al igual que ocurre en Java con la **palabra reservada** `void` , Kotlin tiene `Unit` , con la que **describimos** que una **función no retorna valor.** En la siguiente simplemente escribimos por pantalla un saludo.

```
fun saludar (nombre: String): Unit

{

println("Hola "+nombre);

}
```

Cabe destacar que el compilador de Kotlin no puede inferir el tipo del valor de retorno en funciones con cuerpo de bloques de código. Sin embargo, el tipo `Unit` es una excepción y se puede omitir de la declaración debido a su naturaleza.

```
fun saludar (nombre: String)

{

println("Hola "+nombre);

}
```

Argumentos por defecto y argumentos nombrados

Kotlin nos permite definir argumentos con valores por defecto y nombrados. Con tal fin vamos a partir de la siguiente función:

```
fun suma (a: Int, b: Int) = a + b;
```

No obstante, podríamos querer configurar que `b` siempre tu viese un valor por defecto definido (p. ej., 0):

```
fun suma (a: Int, b: Int = 0) = a + b;
```

Al llamarla ya no necesitamos pasar el argumento `b` y podemos hacerlo como `suma(13)`, obteniendo como resultado 13. Sin embargo, ahora también nos interesa que el valor `a` pueda tener uno por defecto, por lo que definimos la función de esta manera:

```
fun suma (a: Int = 3, b: Int = 4) = a + b;
```

Al llamar a la función ya no necesitamos pasar un argumento y podemos hacerlo como `suma()`, obteniendo como resultado 7, pero ¿qué ocurre si lo que queremos es no escribir uno de los argumentos? Pues, al llamarla, siempre **entenderá** que el **único argumento** es el **primero definido** en la función, es decir, en nuestro caso, `a`.

Para solventar este problema Kotlin ha implementado los **argumentos nombrados,** de tal forma que cuando vayamos a llamarla podemos **definir** su **nombre:**

```
fun suma (a: Int = 3, b: Int = 4) = a + b;

fun main() {

println("La suma es: "+sumar(b = 20));

}
```

Así, obtenemos como resultado 23, pero también tenemos la posibilidad de nombrar a los dos argumentos:

```
fun suma (a: Int = 3, b: Int = 4) = a + b;

fun main() {

println("La suma es: "+sumar(a= 5, b = 20));

}
```

Y el resultado es 25. Incluso es posible cambiar su orden cuando están nombrados:

```
fun suma (a: Int = 3, b: Int = 4) = a + b;

fun main() {

println("La suma es: "+sumar(b= 5, a = 20));

}
```

2.5. Instrucciones de control

El **flujo** del **programa** es la **línea** que sigue el **dispositivo** al **ejecutar código.** Si no ocurre nada que lo altere, empezará en la **primera sentencia** e irá ejecutando de manera individual cada una, de arriba abajo, hasta llegar al final.

Sin embargo, es complicado entender un programa sin que el flujo pueda variar. (P. ej., en una web de la previsión meteorológica habrá condiciones en el flujo, a saber, si hace buen tiempo ocurre una cosa, y si hace mal tiempo ocurre, otra. De esta manera, el flujo tiene caminos diferentes que recorre hasta llegar al final de la ejecución).

Instrucciones de control condicional

if / **if-else**

La **estructura** de control condicional if es la **más sencilla** y con ella, mediante una **condición booleana,** el flujo puede tomar un **camino específico** para esa condición:

```
// Suponiendo la variable edad declarada con un int que representa la edad de una persona

if (edad >= 18) {

println("mayor de edad");

}
```

La condición booleana aparece entre paréntesis () y será evaluada por Kotlin. Si el resultado es *true,* el flujo entrará a **ejecutar** las **sentencias** que hay **dentro** de las **llaves** ({}), y en caso contrario, se las saltará.

Se puede complicar un poco más añadiendo otro camino para el flujo mediante la **estructura** if-else . Si se cumple la condición, el flujo seguirá un camino y, si no, se irá a otro:

```
// Suponiendo la variable edad declarada con un int que representa la edad de una persona

if (edad >= 18) {

println("mayor de edad");

} else {

println("menor de edad");

}
```

when

La **expresión** condicional `when` nos permite **comparar** el **valor** de un **argumento** con una **lista** de **entradas.** Estas últimas tienen **condiciones asociadas** al cuerpo que se ejecutará y pueden ser las siguientes:

- Expresiones.
- Comprobaciones de rangos.
- Comprobaciones de tipos.

Cuando ocurre la coincidencia se **ejecuta** la **rama correspondiente.** Veamos un ejemplo:

```
fun main() {

val entrada = 'c';

when (entrada) {

'c' -> print("Continuando...");

'p' -> print("Parar y cerrar...");

else -> print("Entrada inválida");

}

}
```

Del mismo modo que ocurre con la expresión `if` , se usa `else` en caso de que ninguna de las condiciones se cumpla. Además de literales, también se pueden **usar expresiones variadas,** como conjunciones, comparaciones, operaciones, etc.

La expresión `when` se puede considerar análoga a la sentencia `switch` de Java, solo que `when` no requiere sentencias `break` para determinar la terminación de las ramas de condición.

Podemos **comprobar varios valores** en una **entrada** y para ello se debe de **pasar** la **lista,** separada por comas, a la **condición** de la **rama:**

```
fun main() {

val entrada = 2

when (entrada) {

1, 2, 3 -> println("Te toca turno nocturno")

4, 5, 6 -> println("Te toca turno diurno")

}
```

Otra de las características que nos permite la expresión `when` es **usar rangos** como **condiciones** de las **entradas.** Denota la coincidencia con el **operador** `in` o `!in`. En el siguiente ejemplo generamos un numero entero aleatorio entre 0 y 200, donde determinamos si pertenece a los rangos [0,49] o [50,100]. Vamos a pedir que nos inserte el número por teclado:

```
fun main() {

val entrada: Int = (Math.random() * 200).toInt()

when (entrada) {

in 0..49 -> print("$entrada pertenece a [0..49]")

in 50..100 -> print("$entrada pertence a [50..100]")

else -> print("$entrada: Fuera de los rangos contemplados")

}

}
```

También podemos comparar tipos con el **operador** `is` . Veamos un ejemplo:

```
fun main() {

val respuesta: Any = 12

when (respuesta) {

is Int -> print("Respuesta Entera")

is String -> print("Respuesta String")

}

}
```

A diferencia de Java, **no** es **necesario castear** la **variable** para determinar si es el tipo correcto; a esto se le llama ***smart cast*** en Kotlin.

Por otro lado, podemos **comprobar expresiones booleanas sin** tener un **parámetro** de **comparación.** En estos casos, la instrucción `when` se puede escribir sin argumento:

```
fun main() {

val a = -5

when {

a > 0 -> print("Es positivo")

a == 0 -> print("Es cero")

else -> print("Es negativo")

}

}
```

Cabe la posibilidad de **pasar** una **variable** en el **argumento** de `when` y con tal fin se tiene que **crear** una **expresión** de **asignación** a una **variable no mutable:**

```
fun main() {

val factorSuerte = 0.2

val bonus = 0.3

when (val damage: Double = factorSuerte + bonus) {

in 0.0..0.3 -> print("Daño recibido:${damage * 10}")

in 0.3..0.6 -> print("Daño recibido:${damage * 20}")

in 0.6..1.0 -> print("Daño recibido:${damage * 30}")

}

}
```

Podemos ver que la variable `damage` se declara e inicializa en la cabecera de la instrucción `when`, lo que permite usarla dentro de su bloque de estructura.

Finalmente, podemos usar la instrucción `when` como una **expresión** en **retornos** y en **asignaciones** de **variable** de la **misma forma** que `if`. En este caso, es imprescindible **definir** la **rama** `else`, a menos que las otras cubran todas las opciones posibles.

```
fun main() {

val nota = 4

val calificacion = when (nota) {

1 -> "Insuficiente"

2 -> "Deficiente"

3 -> "Aceptable"

4 -> "Notable"

5 -> "Excelente"

else -> "No permitido"

}

}
```

Instrucciones de control iterativas

for

El bucle `for` en Kotlin se **asemeja** a las **sentencias** `foreach` de otros lenguajes; tenemos un **iterador** para **recorrer** los **elementos.** La **sentencia** se compone de estos **elementos:**

- Declaración de variables.
- Expresión contenedora. Se compone del operador `in` y los datos que hay que recorrer (rangos, *arrays,* listas, etc.).
- Cuerpo del bucle. Se marca entre llaves.

```
fun main(){

for(i in 1..5){

println("Contando $i")

}

}
```

En el ejemplo anterior la variable declarada es `i`, y la estructura de datos, un rango del 1 al 5; así, el cuerpo se ejecutará 5 veces y el valor de la variable `i` es accesible dentro de este, por lo que se puede imprimir su contenido, por ejemplo.

Este bucle en Kotlin nos permite **recorrer rangos** de **valores** y **modificar** sus **límites,** su **orden** o el **ritmo** de las **repeticiones:**

```
fun main() {

// iteración normal

for (char in 'a'..'k') print(char)

// iteración con avance de 2

println()

for (char in 'a'..'k' step 2) print(char)

println()

// iteración en orden inverso

for (char in 'k' downTo 'a') print(char)
```

```
// iteración excluyendo el límite superior

println()

for (char in 'a' until 'f') print(char)

}
```

Si queremos **recorrer** un ***array,*** debemos usar como referencia sus **índices**. Por lo tanto, tenemos que realizar **dos tareas:**

- Crear una variable donde almacenemos el índice.
- Usar la propiedad `.indices` del objeto `array` .

```
fun main() {

val nombres = arrayOf("Carlos", "Sergio", "Juan", "Gerardo", "Manuel")

for (i in nombres.indices) {

println("[$i, ${nombres[i]}]")

}

}
```

Por otro lado, Kotlin permite **usar** la instrucción `for` con el método `withIndex()` , el cual **devuelve** una **dupla** en `IndexedValue` , que contiene el índice y el valor. Se pueden **reescribir** mediante una **desestructuración** del **objeto** en la **forma** (índice, valor):

```
fun main() {

val nombres = arrayOf("Carlos", "Sergio", "Juan", "Gerardo", "Manuel")

for ((i, v) in nombres.withIndex()) {

println("[$i, v]")

}

}
```

De manera adicional, existe la posibilidad de **recorrer** un ***string*** usando la **sentencia** for , que **interpretará** la **posición** y el **valor** de cada **carácter:**

```
fun main(){

for(c in "Kotlin developer"){

println(c)

}

}
```

Finalmente, podemos usar el for en Kotlin de una forma análoga al foreach en Java si disponemos de una **colección.** Por ejemplo, en un *array* de *strings* podemos recorrer cada elemento de la colección de la siguiente forma:

```
val languages = arrayOf("Java", "Kotlin", "JavaScript", "Typescript")

for (language:String in languages){

println("Estoy programando en $language")

}
```

Esta forma de recorrer un *array* es extrapolable para otros tipos de colecciones.

while / **do-while**

La **estructura** while, tal como su nombre indica, realiza un **bucle** mientras una **condición** se **cumpla** y, en el momento que se evalúe como falsa, acabará.

```
var i = 0;

while (i < 4) {

println(i)

i++

}

// Se mostrará por pantalla

// 0 1 2 3

var i = 10;

while (i < 4) {

println(i);

i++;

}

// No se mostrará nada por pantalla porque en la primera vuelta del bucle se evalúa ( 10 < 4) por lo que no entra en el bucle
```

Similar al while, tenemos la estructura do-while, que es igual con una salvedad, a saber, la **primera vuelta** se **ejecutará siempre:**

```
var i = 1;

val n = 5;

do {

println(i);

i++;

} while(i <= n);

// Se mostrará por consola

// 1 2 3 4 5
```

Interrumpir o abandonar un bucle

La **sentencia** break sirve para **terminar** un **bucle,** hacer que el **flujo no** lo **respete** y **salte** como si **hubiese acabado,** aunque su utilización se reserva para casos muy determinados.

Siempre es más recomendable diseñar correctamente un bucle antes de tener que recurrir a ella con excepción de la sentencia switch , donde se requiere utilizar break .

La **expresión** de **salto** continue nos permite **parar** la **iteración** de un bucle y **pasar** a la **siguiente.**

Gestión de excepciones: try-catch

Una **excepción** es un **error** que se produce (por la razón que sea) cuando un **programa** se está **ejecutando** y, salvo que se gestione, no puede seguir. Para manejarlas y que, aunque ocurra una, el programa tenga la capacidad de seguir ejecutándose, se utiliza la estructura try/catch, totalmente **análoga** a como se gestiona en **Java.**

En el anterior ejemplo el flujo de ejecución entra en el try. Cualquier excepción que ocurra dentro de este la manejará el bloque catch, por donde continuará la ejecución (saltando el resto de las sentencias que pudiese haber en el bloque try).

Como complemento a dicha estructura existe finally y, gracias a ella, se pueden **agrupar sentencias,** de manera que se ejecutarán tanto si ocurre una excepción como si no.

2.6. Clases y objetos

Kotlin es un lenguaje de programación que implementa el **paradigma** d e **POO,** donde tenemos **dos elementos básicos** (clases y objetos). En este momento ya hemos visto dentro de otros módulos el estudio conceptual de este paradigma y de sus componentes, por lo que aquí nos centraremos en la manera de usar POO en Kotlin.

Clases

En Kotlin la **declaración** de una **clase** se compone de varios **elementos** (véase la Figura 1) y para definir una debemos usar la **palabra clave** `class`, seguido de su **nombre** o **identificador:**

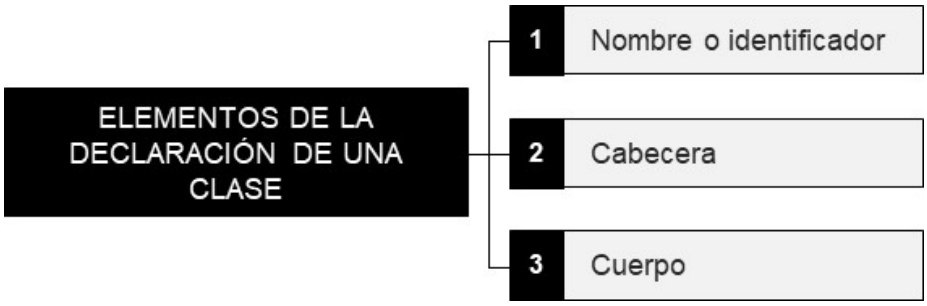


Figura 1. Elementos de la declaración de una clase en Kotlin. Fuente: elaboración propia.

```
class MiPrimeraClase{

//Cuerpo de la clase

}
```

Este ejemplo es un caso reducido de declaración, pero la **cabecera** de una **clase** puede tener **más elementos,** como estos (véase la Figura 2):

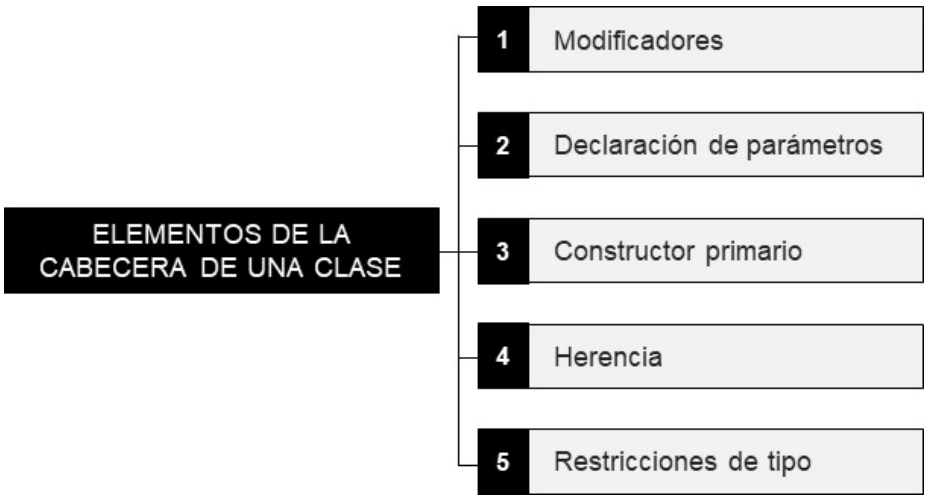


Figura 2. Elementos de la cabecera de una clase. Fuente: elaboración propia.

Modificadores

En Kotlin podemos encontrar una serie de **palabras claves** para **restringir** la **visibilidad** de **clases, constructores, propiedades** y **funciones,** que reciben el nombre de **modificadores:**

- private . Solo visible en ese archivo o en la clase actual.
- protected . Únicamente visible en la clase y las hijas.
- internal . Solo visible en el módulo actual.
- public . Visible en todas partes.

El modificador protected es equivalente a private si la clase no está marcada como open para herencia, tal como veremos más adelante.

Constructores

Un **constructor** en Kotlin es una **función especial** que se usa para **inicializar** los **atributos** de las **instancias** de la clase **(objetos).** Aquí no es necesario emplear palabras clave para crear el objeto, simplemente se **llama** al **constructor** como si se tratara de una **función normal.**

```
class Programador{

//Cuerpo de la clase

}

val p1 = Programador ()
```

Si no lo definimos de manera explícita, el **compilador** de Kotlin **genera** el **constructor** por **defecto sin parámetros**.

En Kotlin disponemos de **dos tipos** de **constructores,** a saber, primarios y secundarios. Vamos a ver sus diferencias.

Constructor primario o principal

Se define como **parte** de la **cabecera** de la propia **clase,** que puede recibir como **argumentos** aquellos **datos** que sean **necesarios** o **imprescindibles** para **inicializar** las **propiedades** en la creación del objeto. Se debe usar la **palabra clave** constructor .

```
class Coche constructor(matricula: String, marca: String)
```

Si directamente declaramos con `val` o `var` antes del parámetro de un constructor primario, crearemos una propiedad de la clase automáticamente:

```
class Coche constructor(val matricula: String, val marca: String)
```

Además, si no tiene anotaciones o modificadores, podemos **contraer** aún más la **sintaxis** omitiendo la palabra constructor :

```
class Coche (val matricula: String, val marca: String)
```

En Kotlin, **por defecto,** todos los **constructores** son **públicos** a menos que se indique explícitamente lo contrario, por lo que en el ejemplo anterior hemos creado uno público de la clase `Coche`, que guarda la matrícula y la marca en sus propiedades. Para **cambiar** la **visibilidad** de los constructores tenemos que **escribir** el **modificador** deseado **entre** el **nombre** de la clase y la **palabra clave** constructor :

```
class Coche internal constructor(val matricula: String, val marca: String)
```

Kotlin nos proporciona la posibilidad de **expandir** la **inicialización** de las **propiedades** usando el bloque init en la clase. Se trata de una **lógica extra** para el **constructor primario.**

```
class Coche (matricula: String, marca: String){

val matricula: String;

val marca: String;

init{

this.matricula = matricula; //Podriamos añadir validacion p.e.

this.marca = marca;

}

}
```

En este caso, las **propiedades** se **declaran** en el **interior** de la **clase** y al **constructor** solo se le **pasan** los **argumentos** (no tienen las palabras clave `var` o `val`); posteriormente, **copiamos** sus **valores** en las **propiedades** de la **clase.**

Constructor secundario

Nos da la posibilidad de **construir mecanismos** de **creación** de **objetos** con otros **parámetros** que nos pueden ser **útiles** en **diferentes situaciones.** Para definirlo debemos usar la **palabra clave** `constructor` en el interior de la clase. Además, nos permite **delegar parámetros** al **primario** mediante la **palabra clave** `this`.

```
class Coche (val matricula: String, val marca: String){

var numeroSerie: String;

init{

numeroSerie = UUID.randomUUID().toString();

}

constructor(numeroSerie: String, matricula: String, marca: String) :this(matriula, marca){

this.numeroSerie = numeroSerie;

}

}
```

Hemos ampliado nuestra clase `Coche` anterior. Al constructor primario le hemos añadido un bloque `init` para gestionar la creación de forma aleatoria del `numeroSerie` (utilizando unas funciones de la clase `UUID`). Posteriormente, hemos creado un constructor secundario al que se le pasan como argumentos el `numeroSerie`, la marca y el modelo. En este, usando `:this(matricula, marca),` hemos delegado al primario la gestión de tales parámetros y propiedades. Finalmente, el secundario asigna el valor del argumento a la propiedad `numeroSerie`.

Podemos definir tantos constructores secundarios como necesitemos, incluso sin determinar uno primario.

Para **instanciar** una **clase** tan solo hemos de escribir desde donde queramos usar nuestro objeto:

```
val tuccson: Coche = Coche("7777MWT", "Hyundai")
```

Ahora bien, como el compilador de Kotlin realiza una inferencia de datos, podemos **simplificarlo** así:

```
val tuccson = Coche("7777MWT", "Hyundai")
```

Es importante resaltar que, cuando defines la variable con la palabra clave `val` para hacer referencia al objeto, esta en sí es de solo lectura, pero el objeto de la clase permanece mutable. Esto significa que no puedes reasignar otro objeto a la variable, aunque sí cambiar su estado en el momento de actualizar los valores de sus propiedades.

Propiedades

Al igual que en otros lenguajes modernos, Kotlin implementa el **uso** de **propiedades,** pero, a diferencia de algunos, este **maneja directamente** el `get` y `set` de una forma **transparente** cuando se **accede** o **modifica** la propiedad. Esto es, se usa de forma directa la **notación** de **punto** en su nombre.

```
println(tuccson.modelo);
```

De la misma forma que pasa con las variables, si usas `val`, declaras una propiedad de solo lectura y, si empleas `var`, será mutable; es decir, cuando se **declara** una **propiedad** con `val`, internamente **solo** se **crea** el `get`, mientras que, si se usa `var`, se crea el `get` y el `set`.

Si deseamos **cambiar** la **lógica** por defecto de los **métodos** `get` y `set`, Kotlin nos permite **llamarlos junto** a la **propiedad** y **definir** el **comportamiento deseado.** Además, podemos hacer lo mismo con la **visibilidad** de ambos usando los **modificadores** del **lenguaje** justo antes de las palabras clave `get` o `set`.

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

val isMayorEdad

get() = this.edad >= 18

var sobrePeso = false

var peso = 0.0

set(value) {

field = value

sobrePeso = value > 100

}
```

```
}

fun main() {

val pp = Persona("Pepe", 40)

pp.peso = 101.0

println("¿Sobrepeso?:${if (pp.sobrePeso) "SI" else "NO"}")

println(pp.peso)

}
```

En el ejemplo anterior se han **creado** los siguientes **elementos:**

- Clase `Persona` . Se han creado las propiedades `nombre` y `edad` en el constructor primario.
- Propiedad `isMayorEdad` . Se ha modificado su `get` por defecto, de tal manera que comprueba el valor de la edad y nos devuelve un booleano con el valor correspondiente.
- Propiedades `sobrePeso` y `peso` . Se ha modificado el `set` de esta última para que automáticamente calcule si la persona tiene sobrepeso al cambiar el valor de la propiedad `peso` .

Herencia

La **herencia** es una de las **funcionalidades base** de la **POO** que hace posible que **entidades diferentes** tengan **patrones** y **propiedades comunes manteniendo** su **independencia.** Para ello debemos realizar estos **pasos:**

- Definir una clase padre que contenga las características comunes.

- Determinar tantas clases hijas como queramos obteniendo automáticamente las características de la clase padre. No obstante, una clase hija solo puede tener una padre (esto recibe el nombre de herencia simple).

De manera análoga a Java, donde todas clases heredan de la clase `Object`, en Kotlin lo hacen de la clase `Any`, que implementa **tres métodos:**

- `equals`. Nos indica si otro objeto es igual al actual.
- `hasCode()`. Devuelve el código `hash` asociado al objeto.
- `toString()`. Devuelve un `string` con la representación del objeto.

Con el objetivo de poder utilizar la herencia en Kotlin debemos **tener** en **cuenta** los **siguientes aspectos** de **sintaxis:**

- Se ha de añadir el modificador `open` al inicio de la clase padre.
- Se deben incluir dos puntos en la cabecera de la clase hija para indicar que hereda de otra.
- Se tiene que especificar y usar el constructor de la clase padre.

```
open class Padre(val nombre: String)

class Hija(nombre: String) : Padre(nombre)
```

Si la clase base **no** tiene **constructor primario** o queremos realizar la **llamada** desde uno **secundario** de la **clase hija,** tenemos que usar la **palabra clave** `constructor` junto a la **palabra clave** `super`.

```
open class Vehiculo (val marca:String, val modelo: String)

class Moto : Vehiculo{

constructor(marca:String, modelo: String): super(marca, modelo)

}
```

Polimorfismo

Kotlin nos permite el polimorfismo y, para **aplicarlo** con la **sobreescritura** de **métodos,** es necesario realizar **dos tareas:**

- Habilitar el método de la clase padre con el modificador `open` .
- Usar el modificador `override` desde el método de la clase hija que se quiere sobrescribir.

Análogamente podemos sobrescribir propiedades marcando con el modificador `open` en la propiedad de la clase padre, y con `override` , en la de la hija. Si la propiedad está declarada como `val` en la clase padre, se puede reescribir con `var` en la clase hija. Sin embargo, lo contrario no es posible.

Interfaces

Tal como hemos indicado antes, **Kotlin** —al igual que muchos lenguajes basados en POO— **solo permite herencia simple,** es decir, que una clase hija únicamente puede tener una padre. Sin embargo, cabe la posibilidad de afrontar este problema mediante el uso de las ***interfaces,*** que nos permiten **definir comportamientos** y **características** que pueden **compartir diferentes clases sin** que tengan una **jerarquía** de **herencia.**

Debemos tener en cuenta las siguientes **consideraciones** a la hora de **declarar** una interface :

- Pueden contener tanto métodos abstractos como con implementación.
- Es posible que contengan propiedades abstractas y regulares, pero sin campos de respaldo.
- Sus propiedades y métodos regulares pueden sobrescribirse con el modificador override sin la necesidad de marcarlos con open .

```
interface Interfaz {

val p1: Int // Propiedad abstracta

val p2: Boolean // Propiedad regular con accesor

get() = p1 > 0

fun m1() // Método abstracto

fun m2() { // Método regular

print("Método implementado")

}

}
```

```
class Ejemplo : Interfaz {

override val p1: Int = 0

override fun m1() {

print("Sobrescibiendo método de Interfaz")

}

}
```

Clases abstractas

Una **clase abstracta** es aquella de la que **no** se **pueden crear instancias** (objetos) y, además, está **marcada** con el **modificador** `abstract` en su **declaración.** No obstante, sí podemos **crear clases hijas** a partir de ella.

Los **miembros** de una clase abstracta también pueden **definirse** como **abstractos,** lo que hace no tengan implementación, sino que deben **implementarse** en las **clases hijas** que **no** estén **marcadas** como **abstractas.**

```
abstract class MiClaseAbstracta {

abstract val campoAbstracto: Int

abstract fun funcionAbstracta()

fun funcionNoAbstracta() {

// Cuerpo

}

}
```

Las clases hijas que sobrescriban a los miembros abstractos deben utilizar la **palabra clave** override en ellos. A diferencia de las clases normales, las abstractas **no necesitan** el **modificador** open para poder heredarse.

```
class Hija : MiClaseAbstracta () {

override val campoAbstracto: Int = 10

override fun funcionAbstracta () {

print(campoAbstracto)

}

}
```

2.7. Comparaciones y elementos nulos

Kotlin, de forma predeterminada, **evita** que sus **tipos** de **datos acepten valores nulos,** es decir, el literal *null*. Ahora bien, si necesitamos trabajar con tipos de datos que los acepten, debemos **definir** nuestras **variables** como **anulables;** esto lo hacemos añadiendo el signo de interrogación (?) al final de la declaración del tipo.

El compilador de Kotlin es capaz de inferir si el tipo será anulable o no, dependiendo del contexto, de tal forma que, si inicializamos una variable con un valor sin tipo declarado, el **compilador** lo **inferirá** y la **asignará** como **no anulable.**

```
fun main() {

var textoNoNulable: String

textoNoNulable = null

//Error: Null can not be a value of a non-null type String

var textoNulable: String?

textoNulable = null

//No hay error

}
```

Para hacer un **uso seguro** de la **anulabilidad** podemos usar el **operador** de **acceso seguro** (`.?`); así, si el elemento existe y tiene valor, nos lo devuelve y, en caso contrario, obtendremos un valor `null` .

```
cocheRojo?.matricula //Si cocheRojo no es null, accedemos al valor de matricula
```

Es importante destacar que en Kotlin el resultado de las variables anulables solo puede ser asignarse a otras anulables.

A fin de poder manejar la anulabilidad de una forma sencilla y rápida Kotlin nos proporciona el **operador** (?:), comúnmente conocido como **operador Elvis**, pero ¿en qué consiste? Veámoslo con un ejemplo sencillo:

```
fun main() {

var a = 21

var c: Int? = null

var division = a / (c ?: 1)

println("Primera division: "+division)

c = 7

division = a / (c ?: 1)

println("Segunda division: "+division)

}
```

En el ejemplo anterior se han definido dos variables (a y c). La variable c es de tipo anulable, por lo que debemos manejar si es *null* o no cuando se utilice. Al realizar la primera división hemos usado el **operador Elvis** (?:), que nos permite **dar** un **valor predeterminado** en caso de que la variable sea *null* (tal como ocurre en este caso). Por lo tanto, la salida de la primera división sería:

$$21 \div 1 = 21$$

Sin embargo, en la segunda operación c tiene un valor de 7, por lo que, al realizar la división, el operador Elvis encuentra un valor no nulo y no asigna su valor predeterminado 1. Así las cosas, la división en este caso sería esta:

$$21 \div 7 = 3$$

2.8. Referencias bibliográficas

Anuar-UNIR. (2024a). *PMDM_2024-2025/Tema 2/Entrenamiento_1*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_1

Anuar-UNIR. (2024b). *PMDM_2024-2025/Tema 2/Entrenamiento_2*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_2

Anuar-UNIR. (2024c). *PMDM_2024-2025/Tema 2/Entrenamiento_3*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_3

Anuar-UNIR. (2024d). *PMDM_2024-2025/Tema 2/Entrenamiento_4*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_4/Entrenamiento_4

Anuar-UNIR. (2024e). *PMDM_2024-2025/Tema 2/Entrenamiento_5/Mascotas*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_5/Mascotas

Editor online de Kotlin

Página de Kotlin Playground.

Editor en línea oficial para programar código en Kotlin.

Kotlin docs

JetBrains. (s. f.). *Kotlin docs*. https://kotlinlang.org/docs/home.html

`Set` es un tipo de dato que trae JavaScript, similar a un *array,* pero con métodos interesantes para manejar los datos.

Kotlin en Visual Studio Code

Ovalle, C. (2021). *Instalación y configuración del compilador Kotlin, en Visual Studio Code* [documentación en línea]. https://github.com/desaextremo/kotlinvscode?tab=readme-ov-file

Repositorio de github con los archivos y las configuraciones necesarias para poder usar a la par que compilar Kotlin en Visual Studio Code.

Patrones de diseño en Kotlin

Ortiz, J. (2023, junio 1). Patrones de diseño en kotlin. *Medium*. https://medium.com/@joseortizfuenzalida/patrones-de-dise%C3%B1o-en-kotlin-3c51b61c733f

Artículo en línea donde se describen los patrones de diseño más importantes de la programación actual y, en especial, sobre el lenguaje Kotlin.

Entrenamiento 1

- Escribe un **programa** en **Kotlin** que, dada una **calificación numérica** entre **0** y **10,** la **transforme** en **alfabética** y **escriba** el **resultado:**
  - De 0 a <3. Muy deficiente.
  - De 3 a <5. Insuficiente.
  - De 5 a <6. Bien.
  - De 6 a <9. Notable
  - De 9 a 10. Sobresaliente.
- El desarrollo paso a paso es este:
  - Lee la nota numérica por teclado.
  - Realiza la conversión a calificación de texto (se recomienda usar la instrucción `when`).
  - Muestra la calificación por pantalla.
- La solución se encuentra en Anuar-UNIR (2024a).

Entrenamiento 2

- Escribe un **programa** en **Kotlin** que, en función de una hora expresada en horas, minutos y segundos, **calcule** y **escriba** la **hora,** los **minutos** y **segundos** que serán **transcurrido** un **segundo.**
- El desarrollo paso a paso es el siguiente:
  - Solicita la hora, el minuto y segundo por teclado.
  - Aumenta en uno el valor de los segundos.
  - Realiza las comprobaciones por si tienes que modificar los minutos y las horas.
  - Muestra la nueva hora por pantalla.
- La solución se encuentra en Anuar-UNIR (2024b).

Entrenamiento 3

- Escribe un **programa** en **Kotlin** para **crear** el juego de **piedra, papel** o **tijera.** Debe seguir la siguiente **estructura:**
  - Explicar las reglas a la persona jugadora.
  - Generar la jugada aleatoria de cada persona jugadora (usar una función).
  - Decidir quién ha ganado.
- El desarrollo paso a paso es este:
  - Importa la librería `random` .
  - Explica por pantalla las reglas.
  - Pide a la persona jugadora que introduzca su jugada.
  - Verifica que es correcta (entre las tres posibles).
  - Genera la jugada aleatoria del ordenador.
  - Muestra la jugada de cada uno en formato texto.
  - Comprueba las jugadas.
  - Muestra quién ha ganado.
- La solución se encuentra en Anuar-UNIR (2024c).

Entrenamiento 4

- Desarrolla una **aplicación** en Kotlin formada por una clase principal, `DamBank` ; otra, llamada `CuentaBancaria` , y otra, `Movimiento` . El programa pedirá los **datos necesarios** a fin de **crear** una **cuenta bancaria.** Si son válidos, la creará y mostrará el **menú principal** para permitir actuar sobre la cuenta, que volverá a mostrar tras cada acción (véase la Tabla 1):

Datos de la cuenta	Mostrará el *international bank acount number* (IBAN), la persona titular y el saldo.
Saldo	Enseñará el saldo disponible.
Ingreso	Pedirá la cantidad que se va a ingresar y realizará el ingreso si es posible.
Retirada	Solicitará el importe que se va a retirar y hará la retirada si es posible.
Movimientos	Mostrará una lista con el historial de movimientos.
Salir	Termina el programa.

Tabla 1. Menú de la aplicación. Fuente: elaboración propia.

-

- Clase `CuentaBancaria` . Una cuenta bancaria tiene como datos asociados el IBAN (formado por 2 letras y 22 números [p. ej., ES6621000418401234567891]), la persona titular (un nombre completo), el saldo (dinero en euros [€]) y los movimientos (histórico de las acciones realizadas en la cuenta, un máximo de 100[*] para simplificar). Cuando se crea una es obligatorio que tenga un IBAN en el formato

correcto y una persona titular (que no podrán cambiar nunca). El saldo será de 0 € y la cuenta no tendrá movimientos asociados. Este solo puede variar cuando se produce un ingreso (entra dinero en la cuenta) o una retirada (sale dinero), y en ambos casos se deberá registrar la operación en los movimientos. Tanto los ingresos como las retiradas solo pueden ser de valores superiores a 0. El saldo de una cuenta nunca podrá ser inferior a -50(*) €; si se produce un movimiento que deje la cuenta con un saldo negativo (no inferior a -50), habrá que mostrar el mensaje «AVISO: Saldo negativo».

- Clase `Movimiento`. Debe tener los siguientes atributos, a saber, identificador único numérico del movimiento (ID), fecha en el formato dd/mm/aaaa, tipo (ingreso o retirada) y cantidad.

- Clase `DawBank`. Clase principal con función `main`, encargada de interactuar con la persona usuaria, mostrar el menú principal, dar *feedback* o mensajes de error, etc. Utilizará la clase `CuentaBancaria`.

▶ El desarrollo paso a paso es el siguiente:

- Genera la clase `Movimiento` con sus propiedades y el método `toString()`.

- Genera la clase `CuentaBancaria`, en la que se debe incluir un *array* de `Movimiento` de dimensión `100`.

- Genera la clase `DawBank` con el método principal que gestionara el menú principal y el objeto de `CuentaBancaria`.

▶ La solución se encuentra en Anuar-UNIR (2024d).

Entrenamiento 5

- Desarrolla una **clase,** llamada Inventario , para **almacenar** la **referencia** a todos los **animales existentes** en una **tienda** de **mascotas** y que debe cumplir con una serie de **requisitos** (véanse la Tabla 1 y la Figura 3). Implementa las demás clases necesarias para la clase Inventario.

Requisitos de la clase Inventario
En la tienda existirán cuatro tipos de animales (perros, gatos, loros y canarios).
Los animales deben almacenarse en un *array* privado dentro de la clase Inventario.
La clase ha de permitir realizar las siguientes acciones, a saber, mostrar la lista de animales (solo tipo y nombre [una línea por animal]), todos los datos de uno concreto y los de la totalidad de animales, insertar animales en el inventario, eliminarlos y vaciar el inventario.

Tabla 1. Requisitos de la clase Inventario. Fuente: elaboración propia.

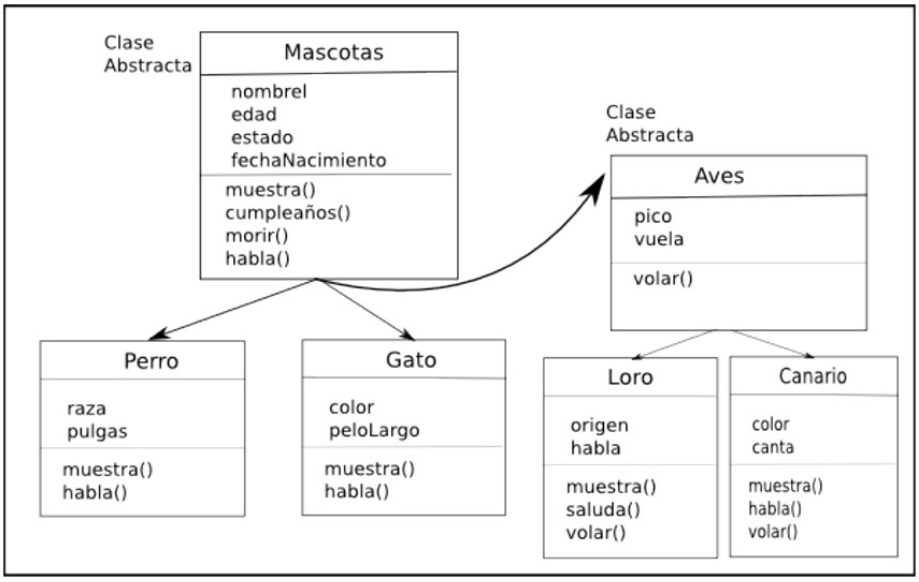


Figura 3. Clases necesarias. Fuente: elaboración propia.

- El desarrollo paso a paso es este (véase la Tabla 2):

Elemento	Descripción
Clases Mascota, Perro, Gato, Loro y Canario	▸ Mascota es una clase abstracta que tiene atributos y métodos comunes para todos los animales. ▸ Perro y Gato heredan de Mascota, y añaden atributos específicos. ▸ Loro y Canario heredan de la clase abstracta Aves que, a su vez, lo hace de Mascota. Esto les permite tener atributos y métodos adicionales, como pico, vuela y volar().
Clase Inventario	▸ Usa una lista privada mutable para almacenar todos los animales. ▸ Implementa métodos a fin de mostrar los animales, añadir, eliminar y vaciar el inventario.
Función principal (main)	▸ Se insertan algunos animales de prueba en el inventario. ▸ Luego, se interactúa con él mostrando la lista de animales y los datos específicos, eliminando animales y vaciando el inventario.

Tabla 2. Desarrollo del entrenamiento. Fuente: elaboración propia.

- La solución se encuentra en Anuar-UNIR (2024e).