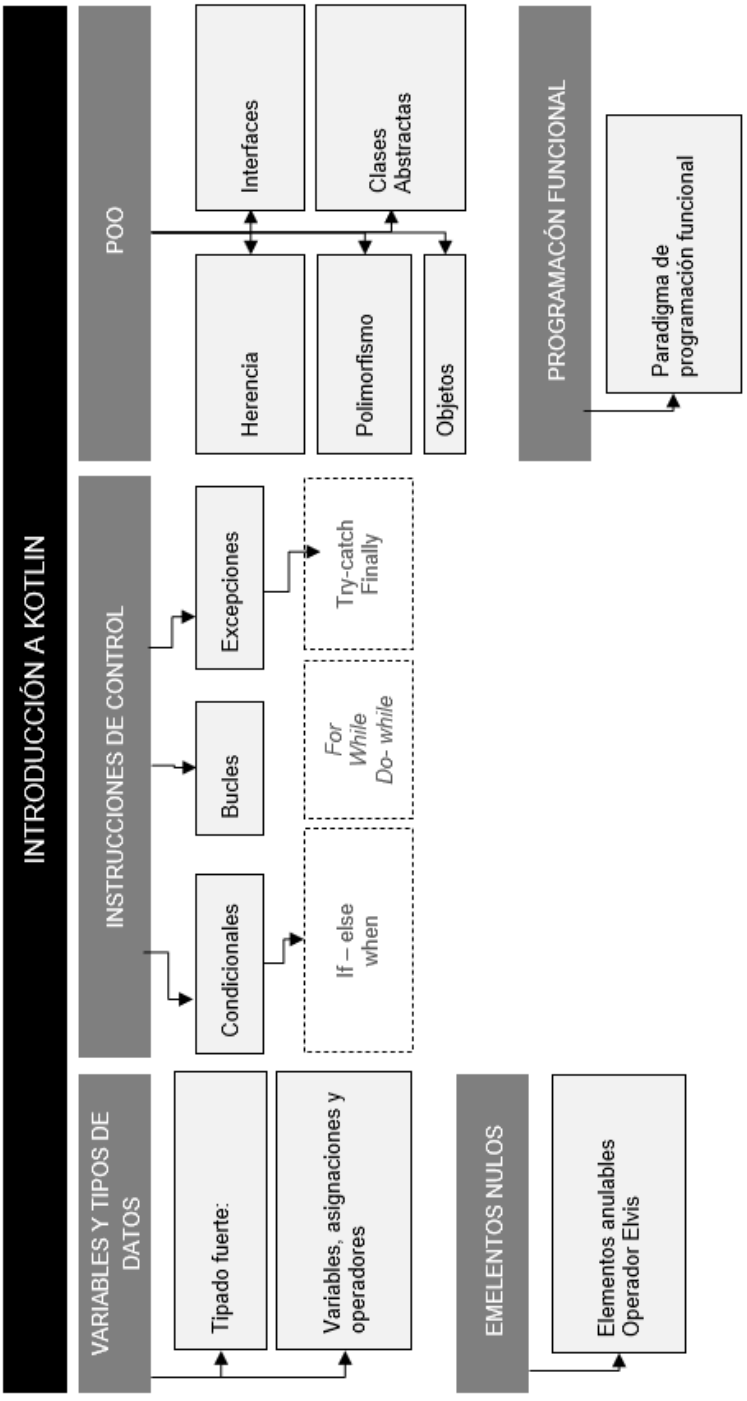
Programación multimedia y dispositivos móviles

Introducción a Kotlin

Índice

Esquema


INTRODUCCIÓN A KOTLIN
VARIABLES Y TIPOS DE DATOS
Tipado fuerte:
Variables, asignaciones y operadores
INSTRUCCIONES DE CONTROL
Condicionales
If – else
when
Bucles
For
While
Do- while
Excepciones
Try-catch
Finally
EMELENTOS NULOS
Elementos anulables
Operador Elvis
POO
Herencia
Polimorfismo
Objetos
Interfaces
Clases Abstractas
PROGRAMACÓN FUNCIONAL
Paradigma de programación funcional

Material de estudio

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.
- Uso de la programación orientada a objetos con Kotlin.

2.2. ¿Qué es Kotlin?

Kotlin es un lenguaje de programación creado en 2010 por Jetbrains, empresa desarrolladora de uno de los IDE para Java más famosos y utilizado del mundo, intellij.

Kotlin surge como una alternativa al lenguaje de programación Java, con la intención de suplir varios de los problemas más habituales que los programadores se encontraban en Java.

Kotlin aporta ciertas ventajas frente a lenguajes como Java haciendo los desarrollos más cómodos.

Seguro contra nulos

Como programadores 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

Kotlin nos permite evitar muchas líneas de código en comparación con otros lenguajes como Java. Por ejemplo, podríamos realizar el equivalente a un POJO (Plain Old Java Objects) en una sola línea en vez de 50-100.

Características de programación funcional

Kotlin está diseñado para que los programadores puedan trabajar tanto con programación orientada a objetos, como con programación funcional (e incluso combinarlas). Esto nos proporciona mucha mayor y la posibilidad de usar características como higher-order functions, function types y lambdas.

Fácil de usar

Kotlin está inspirado en lenguajes ya existentes como Java, C# o Scala, por lo que la curva de aprendizaje para programadores de estos lenguajes es bastante sencilla.

2.3. Variables y tipos de datos

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 ser terminada con punto y coma (pero es muy recomendable por continuidad con otros lenguajes) y por mejorar la legibilidad del código (salvo excepciones) debe escribirse una sentencia por línea.

Comentarios

Como cualquier lenguaje de programación, Kotlin incluye la posibilidad de añadir comentarios a su código.

Los comentarios son caracteres que no se ejecutarán y son útiles para documentar el código, explicar funcionalidades a futuros desarrolladores u ocultar al intérprete de código partes del programa mientras se está desarrollando. El código debe ser los suficientemente claro para que se pueda explicar por sí mismo, si esto no es posible se debe recurrir a los comentarios.

Kotlin tiene dos maneras de escribir comentarios: comentarios de **una línea** que empiezan con // o 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, como su propio nombre indica, son las diferentes clases de datos que Kotlin puede manejar. Cada tipo de dato tiene sus características y operaciones permitidas, por ejemplo, se pueden multiplicar dos datos que sean números, sin embargo, no se pueden multiplicar 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 del tipo literal (se ponen entre comillas simples) o carácter especial (deben de empezar con una barra invertida "\").

- Int: 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: es un 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 ser leídos en otro momento. Una variable 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 esperar 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.

Para declarar una variable debemos de 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 pueda inferir que la variable *edad* es del tipo Int. Del mismo modo ocurre con la variable peso, infiriendo que la variable es del tipo Double.

Es importante destacar, que si no se proporciona un valor inicial cuando se declare una variable es imprescindible especificar el tipo de esta.

```
var altura: Double;
```

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

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

Strings

Aunque aun no hemos visto como 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 literal de la clase String se crea como una instancia de esta, 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 debemos 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

Como ya conocemos de otros lenguajes de programación un array es una estructura de datos almacenados de forma contigua donde todos los elementos son referenciados por un mismo identificador y tipo de dato. Estos elementos están indexados 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(): crear 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

Estos operadores permiten realizar operaciones aritméticas básicas, algunos de ellos son:

- 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, JavaScript hará lo posible por comparar los valores y realizar la operación lógica entre ellos.

- Operador lógico and: &&.

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

Operadores de asignación

Con un operador de asignación se asigna el valor de la derecha en el operador de la izquierda. Por ejemplo: x = 3 se asigna el valor 3 a x.

Algunos operadores de asignación:

- =: 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.

2.4. Uso de funciones

Kotlin es un lenguaje de programación que implementa el estilo de programación funcional (y programación orientada a objetos). Por lo tanto, la función es una herramienta fundamental y muy importante en Kotlin.

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

A una función se le puede llamar desde cualquier zona del programa donde sea necesario. Para definir una función 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 para la función, este debe de 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 definida de la función.

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 el tipo de dato de salida de la función, esto es debido a que al usar funciones con cuerpo de expresión el compilador de Kotlin puede inferir los tipos directamente. En caso de querer defirnlos sería:

```
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 la palabra reservada *Unit*. Por lo tanto, con *Unit* describimos que una función no retorna ningún 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, es posible omitirlo 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 defectos y argumentos nombrados, para ello vamos a partir de la siguiente función:

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

Pero nosotros podríamos querer configurar que b siempre tu viese un valor por defecto definido, por ejemplo 0

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

Al llamar a la función ya no necesitamos pasar el argumento b y podemos llamarla `sum(13)` obteniendo como resultado 13.

Pero ahora también nos interesa que el valor a puede tener un valor por defecto y definimos la función de la siguiente manera.

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

Al llamar a la función ya no necesitamos pasar ningún argumento y podemos llamarla `sum()` obteniendo como resultado 7. Pero que ocurre si solo queremos no escribir uno de los argumentos. Pues al llamar a la función 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 llamar la función podemos definir el nombre del argumento:

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

fun main() {
	println("La suma es: "+sumar(b = 20));
}
```

Obteniendo como resultado 23. Pero también podemos 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));
}
```

Obteniendo como resultado 25. Incluso podemos cambiar el orden de los argumentos 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 ejecutando código. Si no ocurre nada que lo altere, el flujo empezará en la primera sentencia e irá ejecutando una a una cada sentencia de arriba a abajo hasta llegar al final.

Sin embargo, es complicado entender un programa sin que el flujo pueda variar. Por ejemplo, en una web de la previsión meteorológica habrá condiciones en el flujo: si hace buen tiempo ocurre una cosa y si hace mal tiempo ocurre otra cosa. 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 estructura más sencilla, 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 (). Esta condición será evaluada por Kotlin y si el resultado es true, el flujo entrará a ejecutar las sentencias que hay dentro de las llaves {}, en caso contrario, saltará dichas sentencias.

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 camino:

```
  // 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");
  }
```

Instrucción de control condicional: when

La expresión condicional *when* nos permite comparar el valor de un argumento con una lista de entradas.

Las entradas tienen condiciones asociadas al cuerpo que se ejecutará. Dichas condiciones pueden ser:

- 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 en 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, para ello se debe de pasar la lista separa 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, determinando si número pertenece a los rangos [0,49] o [50,100]. Vamos a pedir que nos inserten 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 parametro de comparación, en estos casos la instrucción *when* se puede escribirse sin argumento.

```
fun main() {
    val a = -5

    when {
        a > 0 -> print("Es positivo")
        a == 0 -> print("Es cero")
        else -> print("Es negativo")
    }
}
```

Podemos pasar una variable en el argumento del *when*, para ello se debe de 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 es declarada e inicializada 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 la instrucción *if*. En este caso es imprescindible definir la rama else, a menos que las otras ramas 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.

Para definir el *for* la sentencia está compuesta de una declaración de variables, una expresión contenedora, compuesta por el operador in y los datos a recorrer (rangos, arrays, listas, etc.). Después entre llaves marcamos el cuerpo del bucle.

```
fun main(){
    for(i in 1..5){
        println("Contando $i")
    }
}
```

En el ejemplo anterior, la variable declarada es i y la estructura de datos es un rango del 1 al 5. Por lo que el cuerpo se ejecutará cinco veces y el valor de la variable i es accesible dentro del cuerpo, por lo que se puedes imprimir su contenido, por ejemplo. El bucle for en Kotlin nos permite recorrer rangos de valores y modificar sus límites, orden o 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 los índices de este. Por lo tanto, debemos de crear una variable donde almacenemos el índice y usaremos 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 contienen el índice y el valor. Estos podemos reescribirlos mediante una desestructuración del objeto en la forma (indice, valor)

```
fun main() {
    val nombres = arrayOf("Carlos", "Sergio", "Juan", "Gerardo", "Manuel")

    for ((i, v) in nombres.withIndex()) {
        println("[$i, v]")
```

```
    }
}
```

Adicionalmente podemos recorrer un string usando la sentencia *for*, la cual 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, 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.

Instrucciones de control iterativas: while / do-while

La estructura while como su nombre indica realiza un bucle mientras una condición se cumpla, en el momento que esa condición se evalúe como falsa, el bucle 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: 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 bucles, hacer que el flujo no respete el bucle y salte como si el bucle hubiese acabado. La utilización de break debe hacerse en muy determinados casos, siempre es más recomendable diseñar correctamente un bucle antes de tener que recurrir a esta sentencia, a excepción de la sentencia switch donde la utilización de break es requerida.

La expresión continue es una expresión de salto que 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 esa excepción, la ejecución del programa no puede seguir.

Para manejar excepciones y que, aunque ocurra una el programa pueda seguir ejecutándose, se utiliza la estructura try/catch. Esta estructura es 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 ese bloque try será manejado por el bloque catch, por donde continuará la ejecución (saltando el resto de las sentencias que pudiese haber en el bloque try).

Como complemento al try-catch existe finally, gracias a finally se pueden agrupar sentencias de manera que se **ejecutarán tanto ocurra una excepción como si no**.

2.6. Clases y objetos

Kotlin es un lenguaje de programación que implementa el paradigma de programación orientada a objetos (POO). En POO tenemos dos elementos básicos clases y objetos. En este momento ya hemos visto en otros módulos el estudio conceptual de este paradigma y de sus componentes, por lo que en este apartado nos vamos a centrar como usar POO en Kotlin.

Clases

En Kotlin la declaración de una clase está compuesta por:

- Nombre o identificador de la clase
- Cabecera.
- Cuerpo.

Para definir una clase debemos de usar la palabra clave *class* seguido de su nombre o identificador:

```
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:

- Modificadores.
- Declaración de parámetros.
- Constructor primario.
- Herencia.
- Restricciones de tipo.

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: solo visible en la clase y clases 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, 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). En Kotlin no es necesario el uso de palabras clave para crear el objeto, simplemente se llama al constructor como si se tratará de una función normal.

```
class Programador{
```

```
//Cuerpo de la clase
}
val p1 = Programador ()
```

Si no definimos explícitamente un constructor el compilador de Kotlin genera el constructor por defecto sin ningún parámetro.

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

El constructor primario o principal, se define como parte de la cabecera de la propia clase, pudiendo 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 aun contraer 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 un constructor publico de la clase coche que guarda la matricula y la marca en las propiedades de la clase. Para cambiar la visibilidad de los constructores debemos de 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, siendo esta 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 son declaras en el interior de la clase y al constructor solo se le pasan los argumentos (no tienen las palabras clave var o val), que posteriormente copiamos sus valores en las propiedades de la clase.

Los constructores secundarios nos dan la posibilidad de crear mecanismos de creación de objetos con otros parámetros que nos pueden ser útiles en diferentes situaciones. Para definirlos debemos de usar la palabra clave constructor y se definen en el interior de la clase y nos permite delegar parámetros al constructor 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 aleatorio del numeroSerie (utilizando unas funciones de la clase UUID). Posteriormente hemos creado un constructor secundario que se le pasan como argumentos el numeroSerie, la marca y el modelo, en este constructor usando `:this(matricula, marca)` hemos delegado al constructor primario la gestión de estos parámetros y propiedades. Finalmente, el constructor secundario asigna el valor del argumento a la propiedad numeroSerie.

Podemos definir tantos constructores secundarios como necesitemos, incluso sin definir ninguno constructor primario.

Para instanciar una clase tan solo debemos de escribir desde queramos usar nuestro objeto:

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

Pero como el compilador de Kotlin realiza inferencia de datos podemos simplificarlo a:

```
val tuccson = Coche(“7777MWT”, “Hyundai”)
```

Es importante resaltar que cuando defines la variable con la palabra clave *val* para hacer referencia al objeto, la variable 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, pero puedes cambiar el estado del objeto cuando actualices los valores de sus propiedades.

Propiedades

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

println(tuccson.modelo);

Al igual que las variables, si usas *val*, declaras una propiedad de solo lectura y si usas *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 cambiar 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 hemos creado una clase Persona, en la que se han creado las propiedades nombre y edad en el constructor primario. Posteriormente hemos creado la propiedad *isMayorEdad* modificando su *get* por defecto, de tal manera que comprueba el valor de la edad y nos devuelve un booleano con el valor correspondiente. Posteriormente, se han creado las propiedades sobrePeso y peso,

modificando 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 programación orientada a objetos, la cual nos permite que entidades diferentes tengan patrones y propiedades comunes manteniendo su independencia. Para ello debemos de definir una clase padre que contenga las características comunes, posteriormente podemos definir tantas clases hijas como queramos obteniendo automáticamente las características de la clase padre, pero una clase hija solo puede tener una clase padre, esto recibe el nombre de herencia simple.

Análogamente a Java, donde todas clases heredan de la clase *Object*, en Kotlin todas las clases heredan de la clase *Any*. Esta clase Any implementa 3 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.

Para poder utilizar la herencia en Kotlin debemos de tener en cuenta los siguientes aspectos de sintaxis:

- Se debe de añadir el modificador open al inicio de la clase padre.
- Se debe de añadir dos puntos en la cabecera de la clase hija para indicar que hereda de otra.
- Se debe de 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 un constructor secundario de la clase hija debemos de 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)
}
```

Kotlin nos permite el polimorfismo, para aplicarlo con la sobreescritura de métodos es necesario habilitar el método de la clase padre con el modificador *open* y posteriormente 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 propiedad de la clase hija. Si la propiedad está declarada como *val* en la clase padre, es posible reescribirla con *var* en la clase hija. Sin embargo, lo contrario no es posible.

Como hemos indicado antes Kotlin como muchos lenguajes basados en POO solo permiten herencia simple, es decir una clase hija solo puede tener una clase padre. Sin embargo, este problema podemos afrontarlo mediante el uso de las Interfaces. **Las interfaces** nos permiten definir comportamientos y características que pueden ser compartidos por diferentes clases sin que tengan una jerarquía de herencia.

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

- Pueden contener tanto métodos abstractos como métodos con implementación.
- Pueden contener propiedades abstractas y regulares, pero sin campos de respaldo.

- Las propiedades y métodos regulares de una interface pueden ser sobreescritos con el modificador *override*, sin la necesidad de marcarlos con el modificador 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")
    }
}
```

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

Los miembros de una clase abstracta también pueden estar definidos como abstractos, lo que hace no tengan implementación, sino que deben ser implementados 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 de utilizar la palabra clave override en ellos. A diferencia de las clases normales las clases abstractas no necesitan el modficador open para poder ser heredadas.

```
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. Sin embargo, 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 declararse el tipo, el compilador le inferirá el tipo 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 (.?), de tal forma que si el elemento existe y tiene valor nos devuelve su valor 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 variables anulables solo puede ser asignado a otras variables anulables.

Para poder manejar la anulabilidad de una forma sencilla y rápida, Kotlin nos proporciona el operador (?:), comúnmente conocido como operador Elvis. Pero en que consiste este operador, 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 definimos dos variables, a y c. La variable c es de tipo anulable por lo que debemos de manejar si es null o no cuando se utilice. Al realizar la primera división hemos usado el operador Elvis (?:) el cual nos permite dar un valor

predeterminado en caso de que la variable sea null, como ocurre en este caso, por lo tanto, la salida de la primera división sería 21 / 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 que la división en este caso sería 21 / 7 = 3.

2.8. Referencias bibliográficas

Eixarch, R. P. (2023). *Kotlin y Jetpack Compose. Desarrollo de aplicaciones Android (Profesional).* Ra-Ma, S.A.

Google. (11 de 07 de 2024). *Android Developers*. Obtenido de https://developer.android.com/

JetBrains. (11 de 07 de 2024). *Kotlin*. Obtenido de https://kotlinlang.org/

Leiva, A. (2016). *Kotlin for Android Developers: Learn Kotlin the easy way while developing and Android App.* CreateSpace Independent Publishing Platform.

Trivedi, H. (2020). *Android application development with Kotlin: Build Your First Android App In No Time.* BPB Publications.

A fondo

Editor online de Kotlin

Kotlin playground. (2024). https://play.kotlinlang.org/

Editor online oficial para programar código Kotlin.

Kotlin docs

Kotlin documentation (2024) [documentación en línea].
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). GitHub Repository [documentación en línea].
https://github.com/desaextremo/kotlinvscode?tab=readme-ov-file

Repositorio de github con los archivos y configuraciones necesarias para poder usar y compilar Kotlin en Visual Studio Code.

Patrones de diseño en Kotlin

Ortiz, J. (2023). Articulo online
https://medium.com/@joseortizfuenzalida/patrones-de-dise%C3%B1o-en-kotlin-3c51b61c733f

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

Entrenamientos

Entrenamiento 1

- Escribe un programa en Kotlin que dada una calificación numérica entre 0 y 10, la transforme en calificación alfabética, escribiendo 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.
  - Desarrollo paso a paso:
    - Leer la nota numérica por teclado
    - Realizar la conversión a calificación de texto (recomendación usar instrucción *when*)
    - Mostrar la calificación por pantalla.
  - Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_1

Entrenamiento 2

- Escribe un programa en Kotlin que dada una hora expresada en horas, minutos y segundos que nos calcula y escribe la hora, minutos y segundos que serán, transcurrido un segundo.
  - Desarrollo paso a paso:
    - Solicitamos la hora, minuto y segundo por teclado
    - Aumentamos en uno el valor de los segundos
    - Realizamos comprobaciones por si tenemos que modificar los minutos y las horas.
    - Mostramos la nueva hora por pantalla.

- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_2

Entrenamiento 3

- Escribe un programa en Kotlin para crear el juego de "Piedra, papel o tijera", el programa debe de seguir la siguiente estructura.
  - Explicarle al jugador cómo se juega.
  - Generar la jugada aleatoria de cada jugador (usar una función).
  - Decidir quién ha ganado.
- Desarrollo paso a paso:
  - Importar librería random
  - Explicar por pantalla las reglas
  - Pedir que el jugador introduzca su jugada
  - Verificar que es correcta (entre las 3 posibles)
  - Generar la jugada aleatoria del ordenador
  - Mostrar la jugada de cada uno en formato texto
  - Comprobar las jugadas
  - Mostrar el ganador
- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_3

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 para crear una cuenta bancaria. Si son válidos, creará la cuenta y mostrará el menú principal para permitir actuar sobre la cuenta. Tras cada acción se volverá a mostrar el menú.
  - Datos de la cuenta. Mostrará el IBAN, el titular y el saldo.
  - Saldo. Mostrará el saldo disponible.
  - Ingreso. Pedirá la cantidad a ingresar y realizará el ingreso si es posible.

- Retirada. Pedirá la cantidad a retirar y realizará la retirada si es posible.
- Movimientos. Mostrará una lista con el historial de movimientos.
- Salir. Termina el programa.

Clase CuentaBancaria

Una cuenta bancaria tiene como datos asociados el iban (international bank acount number, formado por dos letras y 22 números, por ejemplo, ES6621000418401234567891), el titular (un nombre completo), el saldo (dinero en euros) y los movimientos (histórico de los movimientos realizados en la cuenta, un máximo de 100(*) para simplificar). Cuando se crea una cuenta es obligatorio que tenga un iban en el formato correcto y un titular (que no podrán cambiar nunca). El saldo será de 0 euros y la cuenta no tendrá movimientos asociados.

El saldo solo puede variar cuando se produce un ingreso (entra dinero en la cuenta) o una retirada (sale dinero de la cuenta). En ambos casos se deberá registrar la operación en los movimientos. Los ingresos y retiradas solo pueden ser de valores superiores a cero. El saldo de una cuenta nunca podrá ser inferior a -50(*) euros. 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

La clase Movimiento deberá tener los siguientes atributos:

- ID: identificador único numérico del movimiento
- Fecha: fecha en el siguiente formado dd/mm/aaaa
- Tipo (Ingreso o retirada)
- Cantidad

Clase DawBank

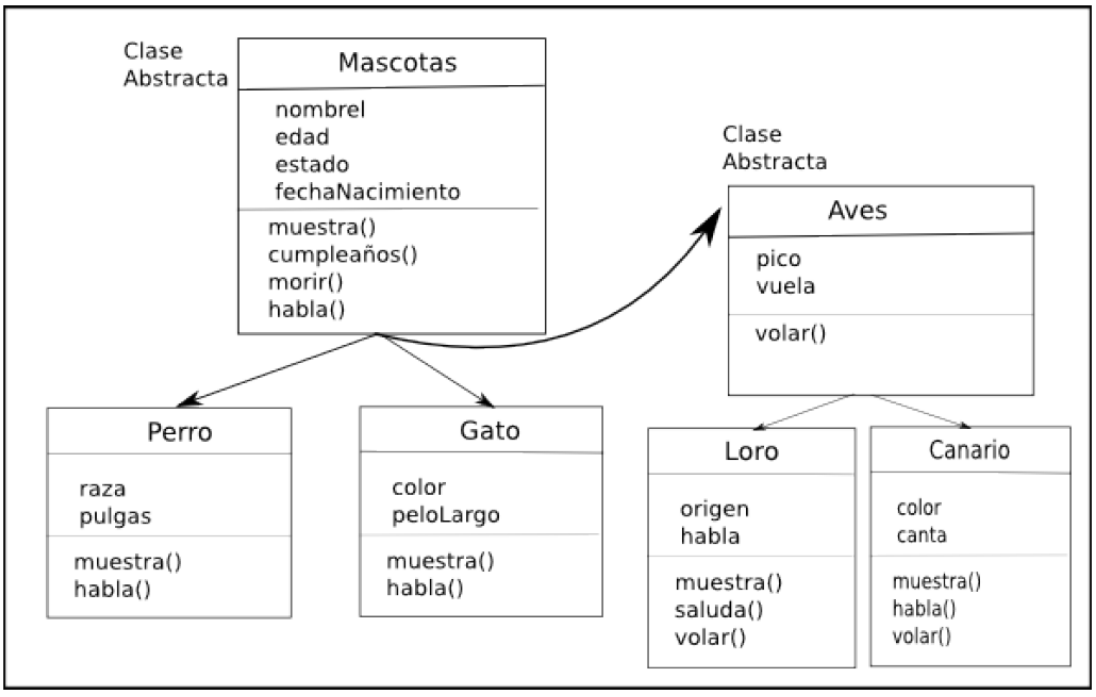
Clase principal con función main. Encargada de interactuar con el usuario, mostrar el menú principal, dar feedback y/o mensajes de error, etc. Utilizará la clase CuentaBancaria.

- Desarrollo paso a paso
- Solución: https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_4/Entrenamiento_4

Entrenamiento 5

- Desarrolla una clase llamada Inventario para almacenar referencia a todos los animales existentes en una tienda de mascotas. Esta clase debe cumplir con los siguientes requisitos:
    - En la tienda existirán 4 tipos de animales: perros, gatos, loros y canarios.
    - Los animales deben almacenarse en un Array privado dentro de la clase Inventario.
    - La clase debe permitir realizar las siguientes acciones:
    - Mostrar la lista de animales (solo tipo y nombre, 1 línea por animal).
    - Mostrar todos los datos de un animal concreto.
    - Mostrar todos los datos de todos los animales.
    - Insertar animales en el inventario.
    - Eliminar animales del inventario.
    - Vaciar el inventario.

Implementa las demás clases necesarias para la clase Inventario.



- Desarrollo paso a paso:

  Clases Mascota, Perro, Gato, Loro, 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, hereda 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 para 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 el inventario mostrando la lista de animales, datos específicos, eliminando animales y vaciando el inventario

- Solución:

https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%202/Entrenamiento_5/Mascotas