IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Programación multimedia y dispositivos móviles

UD02. Fundamentos de Kotlin

Desarrollo de Aplicaciones Multiplataforma

Anna Sanchis Perales

ÍNDICE

1. ¿QUÉ ES KOTLIN?

Kotlin es un lenguaje de programación moderno y de propósito general que se ejecuta en la máquina virtual de Java (JVM) y también puede ser compilado a código nativo. Fue desarrollado por JetBrains, una compañía de software con sede en Rusia, y su primera versión estable se lanzó en 2016.

Kotlin fue diseñado para ser un lenguaje conciso, expresivo y seguro, que aborda algunas de las limitaciones y deficiencias del lenguaje Java. Aunque está estrechamente relacionado con Java y se integra perfectamente con el ecosistema de Java, Kotlin ofrece varias características adicionales y mejoras en comparación con Java.

Algunas de las características clave de Kotlin incluyen:

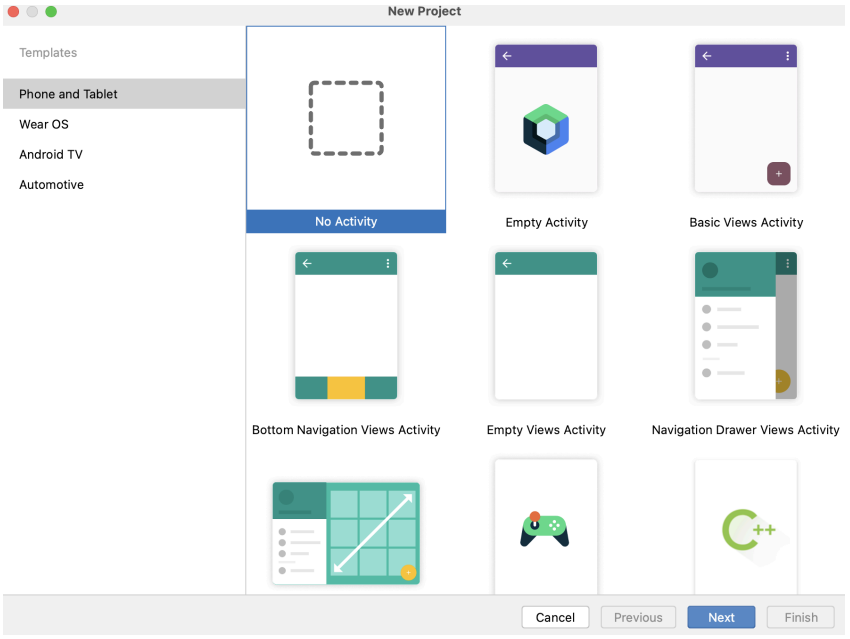
- **Seguridad del tipo**: Kotlin ofrece un sistema de tipos estático que ayuda a evitar errores comunes durante la compilación y a mejorar la estabilidad y seguridad del código.
- **Interoperabilidad**: Kotlin se puede utilizar junto con código Java existente, lo que significa que puedes aprovechar las bibliotecas y frameworks de Java sin problemas.
- **Null safety**: Kotlin aborda el problema de las referencias nulas (null) en el código, lo que reduce la posibilidad de errores de NullPointerException.
- **Extensiones de funciones**: Permite agregar funciones nuevas a clases existentes sin modificar su código fuente original.
- **Funciones de orden superior**: Kotlin admite funciones de orden superior, lo que significa que puedes pasar funciones como argumentos y devolver funciones como resultados.
- **Expresiones lambda**: Kotlin permite el uso de expresiones lambda, lo que facilita la escritura de código más conciso y legible.
- **Programación orientada a objetos**: Kotlin es un lenguaje orientado a objetos en el que todo es un objeto. También proporciona características adicionales, como clases de datos, delegación de propiedades y objetos de compañía.

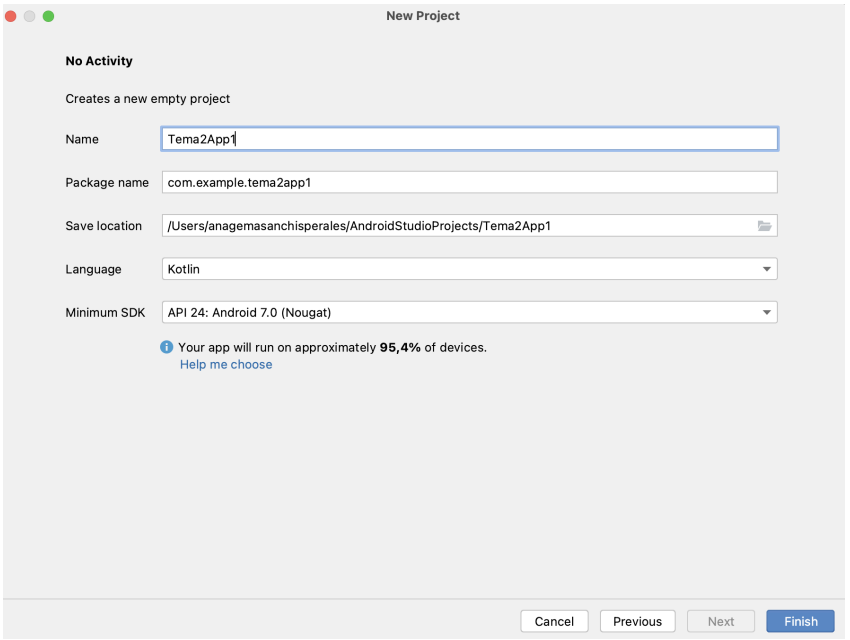
Kotlin se ha vuelto cada vez más popular en los últimos años debido a su sintaxis clara y concisa, su seguridad de tipo y su interoperabilidad con Java. Es utilizado en una variedad de aplicaciones, desde desarrollo de aplicaciones móviles hasta desarrollo de servidores y aplicaciones de escritorio.

2. NUEVO PROYECTO KOTLIN

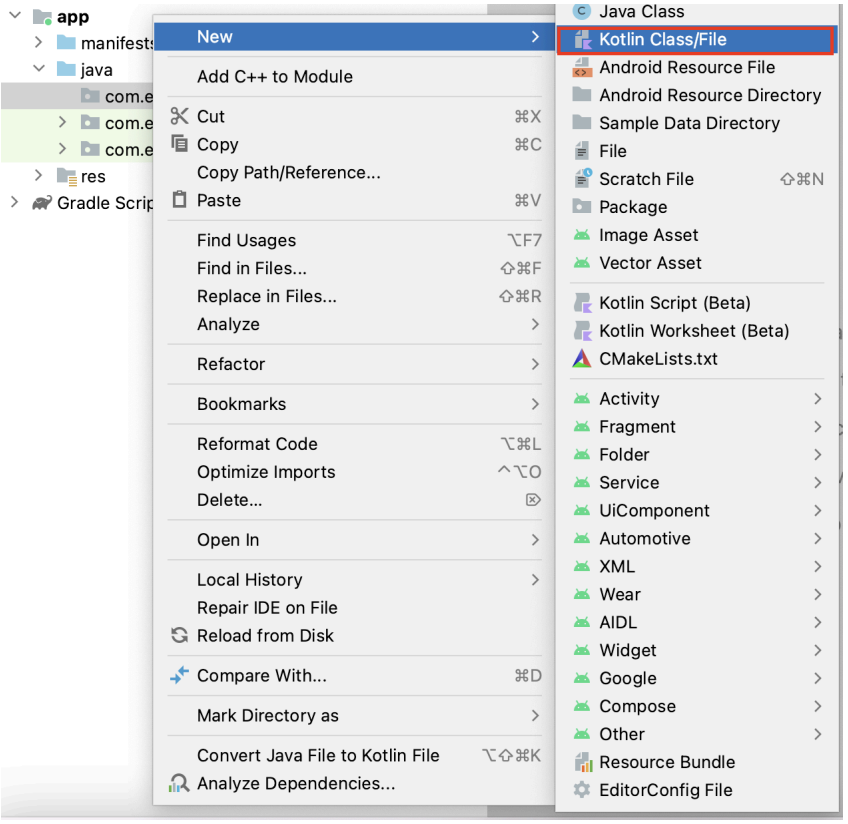
Para empezar a programar vamos a crear un nuevo proyecto en Android Studio, llámalo **Tema2App1**, para ello haz clic en "File" (Archivo) en la barra de menú y luego selecciona "New" (Nuevo) y "New Project" (Nuevo proyecto). No seleccionamos ninguna plantilla, No Activity.

A continuación se muestran las dos pantallas:

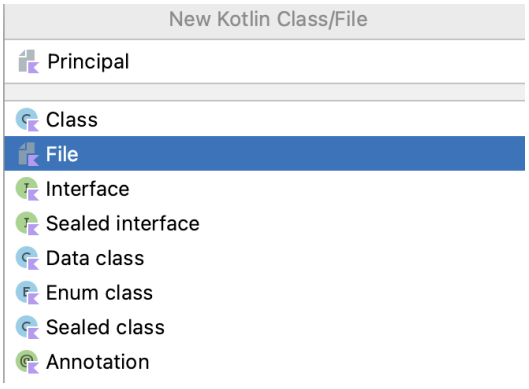




Tras ello procedemos a crear un nuevo archivo de tipo Kotlin, para ello nos situamos en el apartado Project app | java | nombre del paquete | click derecho y seleccionamos New | Kotlin Class/Fila



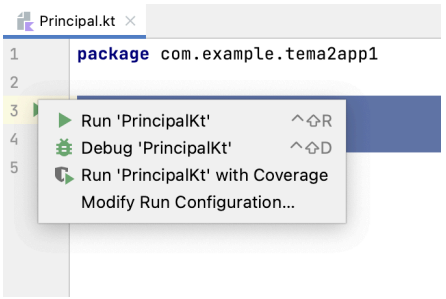
Crearemos un archivo de tipo Kotlin y le llamaremos Principal.



Y para empezar crearemos la función principal conocida como main(), ésta se ejecuta cuando se ejecuta el programa e imprimimos el mensaje “Hola Kotlin!” por consola. El código sería el siguiente:

```
fun main(){
    print("Hola Kotlin!")
}
```

Para ejecutar el código haremos clic sobre el botón de play verde que aparece en el margen izquierdo del código y luego seleccionamos Run 'Principal.kt'.



Observamos como en la consola se nos muestra el mensaje Hola Kotlin.

3. FUNCIONES

En Kotlin, puedes definir funciones utilizando la palabra clave fun. Aquí tienes algunos ejemplos de cómo puedes crear funciones en Kotlin:

- Función sin parámetros ni valor de retorno:fun sayHello() { println("Hola") }

```
fun sayHello() { println("Hola") }
```

En este caso, la función sayHello() no recibe ningún parámetro y no devuelve ningún valor. Simplemente imprime "Hola" en la consola.

- Función con parámetros:

```
fun greet(name: String) { println("Hola, $name") }
```

La función greet() recibe un parámetro de tipo String llamado name. Imprime "Hola, " seguido del nombre que se pasa como argumento.

4. VARIABLES

Una variable es un contenedor que almacena un valor en la memoria durante la ejecución de un programa. En Kotlin, se pueden declarar variables utilizando la palabra clave **var** para permitir cambios en su valor (variables mutables) o la palabra clave **val** para declarar variables con un valor único que no puede cambiarse una vez asignado (variables inmutables o constantes).

Kotlin ofrece varios tipos de variables que se pueden utilizar según el tipo de dato que se almacena. A continuación, se presentan los tipos de variables más comunes en Kotlin.

4.1 Números enteros

Los números enteros representan valores numéricos enteros sin parte decimal.

- **Byte**: Almacena números enteros de 8 bits, con valores en el rango de -128 a 127.
- **Short**: Almacena números enteros de 16 bits, con valores en el rango de -32,768 a 32,767.
- **Int**: Almacena números enteros de 32 bits, con valores en el rango de -2,147,483,648 a 2,147,483,647.
- **Long**: Almacena números enteros de 64 bits, con valores en el rango de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.

Ejemplo:

```
var age: Int = 25
val yearOfBirth: Short = 1998
```

4.2 Números reales

Representan valores numéricos con una parte decimal.

- **Float**: Almacena números de punto flotante de 32 bits, con aproximadamente 6-7 dígitos decimales de precisión.
- **Double**: Almacena números de punto flotante de 64 bits, con aproximadamente 15 dígitos decimales de precisión.

Ejemplo:

```
var pi: Float = 3.14159f
val gravity: Double = 9.8
```

4.3 Caracteres y cadenas de texto

- **Char**: Almacena un único carácter Unicode.
- **String**: Almacena una secuencia de caracteres.

Ejemplo:

```
var letter: Char = 'A'
val message: String = "Hola, mundo!"
```

4.4 Booleanos

- **Boolean**: Almacena un valor verdadero (true) o falso (false).

Ejemplo:

```
var isRaining: Boolean = false
val hasPermission: Boolean = true
```

4.5 Arrays

- **Array**: Almacena una colección de elementos del mismo tipo.

Ejemplo:

```
val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
val names: Array<String> = arrayOf("Alice", "Bob", "Charlie")
```

Estos son solo algunos ejemplos de los tipos de variables que se pueden utilizar en Kotlin. Además, Kotlin también admite tipos de datos personalizados y la capacidad de inferir el tipo de variable automáticamente utilizando la palabra clave var o val sin especificar explícitamente el tipo.

4.6 Ejemplos

Crear un programa que defina dos variables inmutables de tipo Int. Luego definir una tercera variable mutable que almacene la suma de las dos primeras variables y las muestre. Seguidamente almacenar en la variable el producto de las dos primeras variables y mostrar el resultado.

El código asociado sería:

```
fun main() {
    val num1: Int = 5
    val num2: Int = 10
    var result: Int

    result = num1 + num2
    println("La suma de num1 y num2 es: $result")

    result = num1 * num2
    println("El producto de num1 y num2 es: $result")
}
```

Al ejecutar el programa, obtendrás la siguiente salida:

```
La suma de num1 y num2 es: 15
El producto de num1 y num2 es: 50
```

Si entramos a la página oficial de Kotlin podemos ver que una de sus premisas es que un programa en Kotlin sea "CONCISO" (es decir que se exprese un algoritmo en la forma más breve posible).

Una versión más concisa del programa que cumple con los mismos requisitos:

```kotlin
fun main() {
    val num1 = 5
    val num2 = 10
    var result: Int

    result = num1 + num2
    println("La suma de num1 y num2 es: $result")

    result = num1 * num2
    println("El producto de num1 y num2 es: $result")
}
```

En esta versión, utilizamos la inferencia de tipos de Kotlin para evitar especificar explícitamente el tipo de las variables num1 y num2. Kotlin puede determinar automáticamente que son de tipo Int según los valores asignados.

5. DATOS NULOS Y CUALQUIER VALOR

Datos nulos

En Kotlin, la gestión de datos nulos se realiza de manera segura y explícita mediante el uso de tipos anulables y el operador null seguro (?.). Algunos ejemplos son:

Tipos anulables: En Kotlin, puedes definir un tipo de dato anulable utilizando el operador de interrogación ? después del tipo de dato. Por ejemplo:

```
var name: String? = null
```

En este caso, la variable name es de tipo String?, lo que significa que puede contener un valor String o el valor nulo (null).

Cualquier valor

En Kotlin, el tipo Any representa cualquier tipo de valor. Es similar al tipo Object en Java o Any en otros lenguajes de programación. Básicamente, Any es la raíz de la jerarquía de tipos en Kotlin.

Cuando declaramos una variable con el tipo Any, podemos asignarle cualquier valor, ya sea un tipo primitivo, un objeto de una clase definida por el usuario o incluso una función. Por ejemplo:

```
fun main() {
    val anyValue: Any = 42
    println(anyValue)

    val anyObject: Any = Person("John", 30)
    println(anyObject)
}

data class Person(val name: String, val age: Int)
```

6. INFIX

En Kotlin, la palabra clave infix se utiliza para definir funciones o extensiones de funciones con una notación especial conocida como "llamada infix". Esto permite llamar a la función sin utilizar la notación de punto ni paréntesis alrededor del argumento. En su lugar, se utiliza el operador infix entre el receptor y el argumento.

Para declarar una función como infix, debes seguir los siguientes pasos:

1. Declara una función o una extensión de función.
2. Utiliza la palabra clave infix antes de la definición de la función.

Un ejemplo sería:

```
infix fun Int.sumar(numero: Int): Int {
    return this + numero
}
fun main() {
    val resultado = 5 sumar 3
    println(resultado) // Imprime: 8
}
```

En el ejemplo anterior, hemos definido una función sumar como infix que toma un parámetro número y devuelve la suma de this (el receptor) y número. Dentro de la función, podemos utilizar this para referirse al objeto en el que se llama la función sumar.

Luego, en la función main(), hemos utilizado la función sumar de forma infix para sumar los números 5 y 3. En lugar de llamar a la función como 5.sumar(3), podemos utilizar el operador infix entre los números para obtener un código más legible y conciso.

Es importante tener en cuenta que no todas las funciones se pueden declarar como infix. Hay ciertas restricciones que se deben seguir:

- La función debe ser miembro de una clase o una extensión de una clase.
- La función debe tener un solo parámetro.

- El parámetro debe tener un tipo concreto (no puede ser un tipo genérico).

7. SOBRECARGA DE MÉTODOS

En Kotlin, la sobrecarga de métodos (o funciones) te permite definir múltiples funciones con el mismo nombre pero con diferentes parámetros. Esto significa que puedes tener varias versiones de una función con comportamientos diferentes dependiendo de los argumentos que se le pasen.

La sobrecarga de métodos se utiliza para mejorar la legibilidad y la reutilización del código al proporcionar diferentes formas de llamar a una función con diferentes combinaciones de argumentos. Kotlin permite sobrecargar tanto las funciones miembro de una clase como las funciones de nivel superior.

Aquí tienes un ejemplo que muestra cómo se puede utilizar la sobrecarga de métodos en Kotlin:

```
class Calculadora {
    fun sumar(a: Int, b: Int): Int {
        return a + b
    }

    fun sumar(a: Double, b: Double): Double {
        return a + b
    }

    fun sumar(a: Int, b: Int, c: Int): Int {
        return a + b + c
    }
}

fun main() {
    val calculadora = Calculadora()

    val resultado1 = calculadora.sumar(2, 3)
    println(resultado1) // Imprime: 5

    val resultado2 = calculadora.sumar(2.5, 3.7)
    println(resultado2) // Imprime: 6.2
```

```
    val resultado3 = calculadora.sumar(2, 3, 4)
    println(resultado3) // Imprime: 9
}
```

En el ejemplo anterior, tenemos una clase Calculadora con tres funciones sumar sobrecargadas. La primera función sumar toma dos parámetros enteros y devuelve un entero. La segunda función sumar toma dos parámetros de tipo Double y devuelve un Double. La tercera función sumar toma tres parámetros enteros y devuelve un entero.

En la función main(), creamos una instancia de la clase Calculadora y llamamos a las diferentes versiones de la función sumar con diferentes combinaciones de argumentos. Dependiendo de los tipos de argumentos que pasemos, se ejecutará la versión correspondiente de la función sumar.

Es importante tener en cuenta que la sobrecarga de métodos se basa en los tipos y el número de parámetros de las funciones, por lo que no se puede sobrecargar una función solo cambiando su valor de retorno.

8. ARRAY DE ARGUMENTOS

En Kotlin, puedes pasar un array de argumentos utilizando la sintaxis de desestructuración. Por ejemplo:

```
fun myFunction(vararg args: String) {
    // Código de la función
    for (arg in args) {
        println(arg)
    }
}
fun main() {
    val myArray = arrayOf("Hola", "Mundo", "en", "Kotlin")
    myFunction(*myArray)
}
```

En el ejemplo anterior, myFunction es una función que recibe un número variable de argumentos de tipo String. El modificador vararg indica que los argumentos se pueden pasar como un array o como argumentos separados por comas.

En la función main(), se crea un array de tipo String llamado myArray. Para pasar este array como argumentos a myFunction, se utiliza el operador de expansión * antes del nombre del array (*myArray). Esto permite desestructurar el array y pasar sus elementos como argumentos separados.

Dentro de la función myFunction, puedes iterar sobre los argumentos utilizando un bucle for, como se muestra en el ejemplo.

Al ejecutar el código, se imprimirán los elementos del array "Hola", "Mundo", "en" y "Kotlin".

9. BUCLES

FOR

El bucle for lo podemos utilizar de los siguientes modos:

1: Bucle for para recorrer un rango de números y mostrarlos en la consola:

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

2: Bucle for para recorrer un array y calcular la suma de sus elementos:

```
fun main() {
    val numeros = arrayOf(10, 20, 30, 40, 50)
    var suma = 0

    for (numero in numeros) {
        suma += numero
    }

    println("La suma de los números es: $suma")
}
```

WHILE

El bucle while lo utilizamos para repetir un bloque de código mientras se cumpla una condición específica. La sintaxis básica del bucle while es la siguiente:

```
while (condición) {
    // Código a ejecutar mientras se cumpla la condición
}
```

El bloque de código dentro del bucle while se repetirá mientras la condición especificada sea verdadera. Si la condición es falsa desde el principio, el código dentro del bucle nunca se

ejecutará.

Ejemplo de imprimir los números del 1 al 5 usando un bucle while:

```
fun main() {
    var i = 1
    while (i <= 5) {
        println(i)
        i++
    }
}
```

10. SENTENCIA WHEN

La sentencia when es una estructura de control que se utiliza para reemplazar múltiples declaraciones if-else if-else cuando necesitas tomar diferentes acciones según el valor de una expresión. Es similar a un switch en otros lenguajes de programación, pero en Kotlin es más poderoso y versátil.

La sintaxis básica de la sentencia when es la siguiente:

```
when (expresion) {
    valor1 -> {
        // Código a ejecutar si la expresión coincide con valor1
    }
    valor2 -> {
        // Código a ejecutar si la expresión coincide con valor2
    }
    // Puedes añadir más casos...
    else -> {
        // Código a ejecutar si la expresión no coincide con ninguno de
los valores anteriores
    }
}
```

La expresión es la variable o valor que se evalúa en cada caso. Cuando se encuentra una coincidencia con uno de los valorX, el código correspondiente a ese caso se ejecuta. Si no hay coincidencia con ninguno de los valores, se ejecutará el bloque de código dentro de else (opcional).

A continuación se muestra un ejemplo:

```
fun evaluarNumero(numero: Int) {
    when (numero) {
        1 -> println("El número es uno")
        2 -> println("El número es dos")
        3, 4 -> println("El número es tres o cuatro")
        in 5..10 -> println("El número está entre 5 y 10")
        else -> println("El número no es reconocido")
    }
}
```

```
fun main() {
    evaluarNumero(3)
    evaluarNumero(7)
    evaluarNumero(11)
}
```

11. PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos (POO) en Kotlin es una forma de estructurar y organizar el código en la que los conceptos del mundo real se modelan como "objetos" que tienen atributos (propiedades) y comportamientos (métodos). Kotlin es un lenguaje de programación moderno que está diseñado para admitir y fomentar la programación orientada a objetos de manera sencilla y expresiva.

11.1 Creación de clases y objetos

Una clase es una plantilla que define cómo se crearán los objetos. Describe las propiedades y los métodos que tendrán los objetos de esa clase. Un objeto es una instancia específica de una clase. Se puede crear múltiples objetos a partir de una sola clase.

```
// Definición de una clase
class Persona {
    // Propiedades de la clase
    var nombre: String = ""
    var edad: Int = 0
    // Función miembro de la clase
    fun saludar() {
        println("Hola, mi nombre es $nombre y tengo $edad años.")
    }
}
fun main() {
    // Crear un objeto de la clase Persona
    val persona1 = Persona()
    persona1.nombre = "Juan"
    persona1.edad = 30
    // Llamar a la función saludar()
    persona1.saludar()
}
```

Constructores: Los constructores son funciones especiales utilizadas para inicializar los objetos cuando se crean. Kotlin admite dos tipos de constructores: el constructor primario y los constructores secundarios.

```
class Persona(val nombre: String, val edad: Int) {
    // Función miembro de la clase
    fun saludar() {
        println("Hola, mi nombre es $nombre y tengo $edad años.")
    }
}
fun main() {
    // Crear un objeto de la clase Persona utilizando el constructor
primario
    val persona1 = Persona("Juan", 30)

    // Llamar a la función saludar()
    persona1.saludar()
}
```

11.2 Herencia

Kotlin permite la herencia, lo que significa que una clase puede heredar propiedades y métodos de otra clase (superclase). La herencia se utiliza para crear una jerarquía de clases y reutilizar el código.

Un ejemplo:

```
fun main() {
    val perro = Perro("Firulais")
    perro.hacerSonido()
    perro.ladrar()
}
```

11.3 Polimorfismo

El polimorfismo permite que las clases derivadas (subclases) se comporten de manera diferente pero coherente con sus clases base (superclases). Kotlin admite el polimorfismo a través de la herencia y la sobrescritura de métodos.

```
open class Figura {
    open fun dibujar() {
        println("Dibujando una figura genérica.")
    }
}
class Circulo : Figura() {
    override fun dibujar() {
        println("Dibujando un círculo.")
    }
}
fun main() {
    val figura1: Figura = Figura()
    val figura2: Figura = Circulo()
    figura1.dibujar()
    figura2.dibujar()
}
```

11.4 Encapsulación

La encapsulación es un principio de la POO que permite ocultar los detalles internos de una clase y exponer solo la interfaz pública necesaria para interactuar con el objeto. En Kotlin, se utilizan modificadores de acceso como private, protected, internal y public para controlar la visibilidad de las propiedades y métodos.

11.5 Data Class

La palabra clave data class es una característica especial que se utiliza para crear clases que se utilizan principalmente para almacenar datos. Estas clases son conocidas como "data classes". La declaración de una data class proporciona automáticamente una serie de funcionalidades útiles y convenientes relacionadas con la manipulación de datos, como la generación automática de métodos como equals(), hashCode(), toString(), copy(), etc.

La sintaxis para crear una data class es muy simple:

```
data class NombreDeLaClase(val propiedad1: Tipo, val propiedad2: Tipo,
...)
```

Donde:

- NombreDeLaClase: Es el nombre de la data class que deseas crear.
- propiedad1, propiedad2, ...: Son las propiedades o atributos que deseas que tenga la data class, junto con sus tipos de datos.

Ejemplo

```
// Declaración de una data class llamada "Persona"
data class Persona(val nombre: String, val edad: Int)

fun main() {
    // Crear objetos de la data class
    val persona1 = Persona("Juan", 30)
    val persona2 = Persona("María", 25)

    // Acceder a las propiedades de la data class
    println("Nombre de persona1: ${persona1.nombre}")
    println("Edad de persona2: ${persona2.edad}")

    // Método toString() automáticamente generado
    println("Datos de persona1: $persona1")

    // Método equals() automáticamente generado
    println("¿persona1 es igual a persona2?: ${persona1 == persona2}")

    // Método copy() automáticamente generado
```

```
    val persona3 = persona1.copy()
    println("Datos de persona3 (copia de persona1): $persona3")

    // Modificar propiedad usando copy()
    val persona4 = persona1.copy(edad = 40)
    println("Datos de persona4 (copia de persona1 con edad modificada):
$persona4")
}
```

11.6 Enum

En Kotlin, se utiliza la palabra clave enum para crear enumeraciones. Las enumeraciones son un tipo de datos especial que consiste en un conjunto fijo de constantes con nombres descriptivos y valores específicos. Las enumeraciones son útiles cuando se necesita representar un conjunto limitado y conocido de opciones o estados.

La sintaxis para crear una enumeración es la siguiente:

```
enum NombreEnum {
    OPCION1,
    OPCION2,
    OPCION3,
    // Otras opciones...
}
```

Cada constante dentro de la enumeración se separa por coma y no necesita un valor asignado explícitamente. Si no se asignan valores a las constantes, Kotlin les asignará automáticamente valores enteros consecutivos comenzando desde 0.

Un ejemplo sería:

```
enum class DiaSemana {
    LUNES,
    MARTES,
    MIERCOLES,
    JUEVES,
    VIERNES,
```

```
    SABADO,
    DOMINGO
}

fun main() {
    // Utilizar una constante de la enumeración
    val diaHoy = DiaSemana.MARTES

    // Comparar valores de la enumeración
    if (diaHoy == DiaSemana.MARTES) {
        println("Hoy es martes.")
    } else {
        println("Hoy no es martes.")
    }

    // Recorrer todas las constantes de la enumeración
    for (dia in DiaSemana.values()) {
        println(dia)
    }
}
```

11.7 Funciones de alcance

Las "funciones de alcance" son funciones especiales que permiten operar sobre un objeto de manera más concisa y fácilmente legible. Estas funciones son extensiones de la clase que se aplican a un objeto y permiten realizar operaciones con ese objeto dentro del contexto de la función. Las funciones de alcance son muy útiles cuando se trabaja con objetos y se quiere realizar una serie de operaciones en ellos sin tener que repetir constantemente el nombre del objeto.

En Kotlin hay diversas, dos de ellas son:

1. **apply** se utiliza para realizar operaciones en el objeto y devolver el objeto mismo. Es útil cuando se quiere realizar múltiples cambios en el objeto sin necesidad de asignarlo a una variable nuevamente.

```
val persona = Persona("Juan", 30)
persona.apply {
    edad = 31
    // Otras modificaciones...
}
```

2. **with** no es una extensión de la clase, pero te permite acceder a las propiedades y métodos de un objeto sin tener que repetir el nombre del objeto. Dentro de with, puedes acceder a las propiedades y métodos del objeto directamente.

```
val persona = Persona("Juan", 30)
with(persona) {
    println("Nombre: $nombre, Edad: $edad")
}
```

3. **find** es una función de extensión en Kotlin que se utiliza para buscar un elemento en una colección, como una lista, que cumpla con ciertos criterios. En este caso, el criterio de búsqueda se define dentro de las llaves.

11.8 Listas dinámicas

Las listas dinámicas son colecciones que pueden cambiar su tamaño durante la ejecución del programa. Kotlin proporciona una implementación estándar de listas dinámicas a través de la interfaz MutableList y su implementación concreta ArrayList. Con las listas dinámicas, se puede agregar, eliminar y modificar elementos en cualquier momento.

Un ejemplo es:

1. Crear una lista dinámica:

```
// Declarar e inicializar una lista dinámica vacía
val lista: MutableList<String> = mutableListOf()

// Declarar e inicializar una lista dinámica con elementos
val numeros: MutableList<Int> = mutableListOf(1, 2, 3, 4, 5)
```

2. Agregar elementos a la lista:

```
val colores: MutableList<String> = mutableListOf()
colores.add("Rojo")
colores.add("Verde")
colores.add("Azul")
```

3. Acceder a elementos de la lista:

```
val nombres: MutableList<String> = mutableListOf("Juan", "María",
"Pedro")
val nombre1: String = nombres[0]
val nombre2: String? = nombres.getOrNull(1) // Obtener el segundo
elemento o null si no existe
```

4. Modificar elementos de la lista:

```
val frutas: MutableList<String> = mutableListOf("Manzana", "Plátano",
"Naranja")
frutas[1] = "Pera" // Modificar el segundo elemento a "Pera"
```

5. Eliminar elementos de la lista:

```
val numeros: MutableList<Int> = mutableListOf(1, 2, 3, 4, 5)
numeros.removeAt(2) // Eliminar el elemento en la posición 2 (3)
numeros.remove(4) // Eliminar el elemento "4" de la lista
```

6. Recorrer la lista:

```
val animales: MutableList<String> = mutableListOf("Perro", "Gato",
"Elefante")
for (animal in animales) {
    println(animal)
}

// Recorrer la lista con índice
for (index in animales.indices) {
    println("Elemento en la posición $index: ${animales[index]}")
}
```

MutableMapOf

MutableMapOf es una función que se utiliza para crear mapas mutables (mapas que pueden modificarse). Los mapas son colecciones de pares clave-valor, donde cada clave es única y se utiliza para acceder a su valor correspondiente. La función mutableMapOf() permite crear un mapa mutable con la capacidad de agregar, eliminar y modificar pares clave-valor después de su creación.

La sintaxis para crear un mapa mutable con mutableMapOf() es la siguiente:

```
val mapaMutable: MutableMap<TipoDeClave, TipoDeValor> = mutableMapOf(
    clave1 to valor1,
    clave2 to valor2,
    clave3 to valor3,
    ...
)
```

Donde:

- mapaMutable: Es el nombre de la variable que contendrá el mapa mutable.
- TipoDeClave: Es el tipo de datos de las claves del mapa.
- TipoDeValor: Es el tipo de datos de los valores del mapa.
- clave1, clave2, clave3, ...: Son las claves que se utilizarán para acceder a los valores correspondientes.
- valor1, valor2, valor3, ...: Son los valores asociados a las claves.

Un ejemplo podría ser:

```
fun main() {
    // Crear un mapa mutable de estudiantes y sus calificaciones
    val calificaciones: MutableMap<String, Double> = mutableMapOf(
        "Juan" to 8.5,
        "María" to 9.0,
        "Pedro" to 7.2
    )
```

```
    // Agregar un nuevo estudiante y su calificación al mapa
    calificaciones["Ana"] = 9.5

    // Modificar la calificación de un estudiante existente
    calificaciones["Pedro"] = 8.0

    // Eliminar un estudiante del mapa
    calificaciones.remove("María")

    // Imprimir las calificaciones de los estudiantes restantes
    for ((estudiante, calificacion) in calificaciones) {
        println("$estudiante: $calificacion")
    }

    // Verificar si un estudiante está en el mapa y obtener su
calificación
    val nombreEstudiante = "Juan"
    if (nombreEstudiante in calificaciones) {
        val calificacionJuan = calificaciones[nombreEstudiante]
        println("La calificación de $nombreEstudiante es
$calificacionJuan")
    } else {
        println("$nombreEstudiante no se encuentra en el mapa")
    }
}
```

12.ENTRADA DE DATOS POR TECLADO

Para leer datos de entrada por consola en Kotlin, puedes utilizar la clase Scanner de la biblioteca estándar de Java. Aquí tienes un ejemplo de cómo hacerlo:

```
import java.util.Scanner

fun main() {
    val scanner = Scanner(System.`in`)

    println("Ingrese su nombre:")
    val nombre = scanner.nextLine()

    println("Ingrese su edad:")
    val edad = scanner.nextInt()

    println("Ingrese su salario:")
    val salario = scanner.nextDouble()

    println("Nombre: $nombre")
    println("Edad: $edad")
    println("Salario: $salario")

    scanner.close()
}
```

En este ejemplo, primero creamos una instancia de la clase Scanner y la asociamos con la entrada estándar (System.in). Luego, solicitamos al usuario que ingrese su nombre utilizando el método nextLine() y lo almacenamos en la variable nombre. Luego, solicitamos la edad usando nextInt() y el salario utilizando nextDouble().

Finalmente, imprimimos los valores ingresados por el usuario. Recuerda cerrar el scanner llamando al método close() al finalizar la lectura de entrada.