REPASO DE KOTLIN CON EJEMPLOS PMDM

Santiago Rodenas Herráiz

Repaso de kotlin

INTRODUCCIÓN

Con este documento, pretendo ofrecer una visión rápida del lenguaje Kotlin, en comparación con el Java que ya conocéis del curso pasado. A través de sencillos ejemplos y explicaciones, espero proporcionar una introducción efectiva al lenguaje. No pretendo cubrir un módulo completo de programación de primer año de DAM, pero he observado que en años anteriores era necesario dedicar una o dos semanas para familiarizarse con este lenguaje. Kotlin, poco a poco será el lenguaje que sustituya a Java en la programación de aplicaciones con Android.

- Variables

  ```
  En esta sección, exploraremos cómo se declaran y utilizan las variables en Kotlin. Veremos la diferencia entre variables inmutables (`val`) y mutables (`var`), así como las convenciones para nombrarlas y los tipos de datos más comunes que se utilizan.
  ```

- Sentencias

  ```
  Aquí abordaremos las sentencias de control de flujo en Kotlin, como `if`, `when`, y `for`. Estas estructuras nos permiten controlar la ejecución del código según diferentes condiciones y realizar iteraciones sobre colecciones.
  ```

- Arrays

  ```
  En esta parte, aprenderemos cómo trabajar con arrays en Kotlin. Veremos cómo declararlos, inicializarlos y manipular sus elementos. También discutiremos cómo Kotlin ofrece una variedad de funciones útiles para trabajar con arrays.
  ```

- Funciones

  ```
  Exploraremos cómo definir y utilizar funciones en Kotlin. Abordaremos la sintaxis básica para declarar funciones y cómo Kotlin maneja los valores de retorno y los parámetros.
  ```

- Lambda

  ```
  Las expresiones lambda son una característica poderosa de Kotlin. En esta sección, aprenderemos cómo se definen y utilizan las lambdas, así como los casos en los que son especialmente útiles, como en funciones de orden superior. En el último punto, trataremos los callback.
  ```

- Clases

```
En esta sección, discutiremos cómo definir y utilizar clases en Kotlin. Veremos la sintaxis para crear clases, propiedades, métodos y constructores, y cómo se aplican conceptos de orientación a objetos como herencia y polimorfismo.
```

- Extensiones

```
Aquí aprenderemos sobre las funciones de extensión en Kotlin, que permiten añadir nuevas funcionalidades a clases existentes sin modificarlas. Veremos cómo definir funciones de extensión y cómo pueden mejorar la legibilidad y modularidad del código.
```

- Data Class

```
Las `data class` en Kotlin son una forma conveniente de manejar datos. En esta sección, veremos cómo definir una `data class`, así como los métodos automáticamente generados como `toString()`, `equals()`, `hashCode()`, y `copy()`.
```

- Listas

```
En esta parte, discutiremos el trabajo con listas en Kotlin, tanto inmutables (`List`) como mutables (`MutableList`). Veremos cómo declarar, inicializar y manipular listas, y cómo Kotlin ofrece funciones útiles para trabajar con ellas.
```

- Map

```
Exploraremos los mapas (`Map`) en Kotlin, que son colecciones de pares clave-valor. Veremos cómo crear, inicializar y manipular mapas, y cómo utilizar funciones como `put`, `get`, `remove`, y `forEach`.
```

- Callback

```
Finalmente, abordaremos el concepto de callbacks, que son funciones que se pasan como parámetros y se ejecutan cuando una tarea asíncrona se completa. Veremos cómo definir y utilizar callbacks en Kotlin, y cómo esto se relaciona con operaciones asíncronas y la programación basada en eventos.
```

Actualmente, soy profesor de 2º DAM en el IES VIrgen del Carmen de Jaén. Imparto la asignatura de PMDM (Programación multimedia y dispositivos móviles) y la asignatura de PSP (Programación servicios y procesos). Para cualquier consulta, puedes contactar en la siguiente dirección de correo electrónico:

- Email: srodher115@g.educaand.es

Puedes encontrar más información y ejemplos en el repositorio de GitHub:

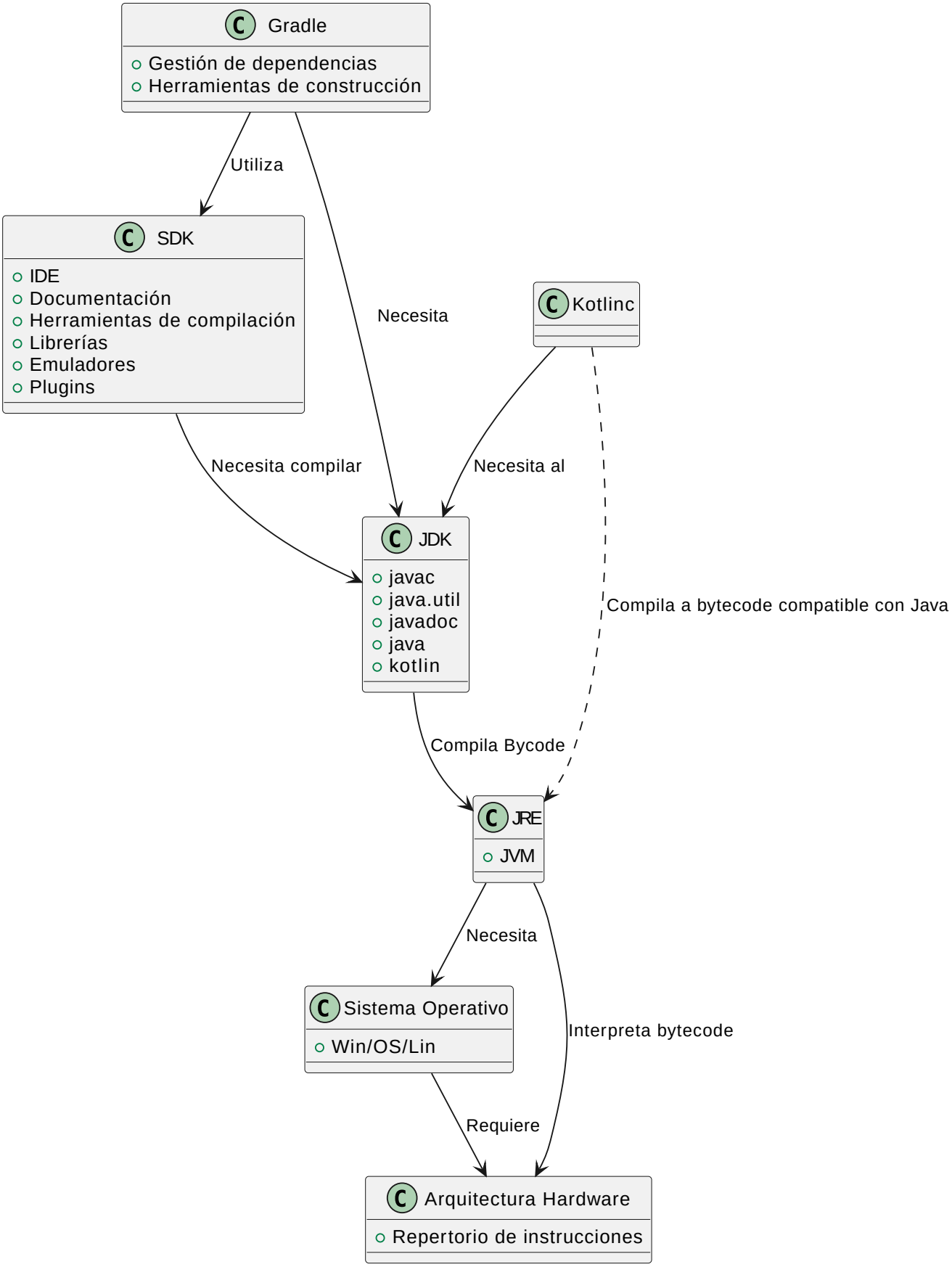
- [GitHub Repository](https://github.com/srodenas?tab=repositories)
- Curso 24/25
- Documentación oficial de Kotlin: Documentación de Kotlin
- :copyright: © 2024 Santiago Rodenas Herráiz.

RECORDANDO CONCEPTOS.

En clase, hablaremos de los siguientes conceptos:

1. **Java** Recordemos que es el lenguaje de programación POO que fue desarrollado por Sun en 1995 y es uno de los más utilizados tanto a nivel empresarial para web, como para aplicaciones móviles (Con Android Studio). Es totalmente portable, porque puede ser ejecutado en cualquier sistema (multiplataforma).
2. **Kotlin** Es el lenguaje sustituto de Android Studio, ya que se considera mas moderno que java y es el oficial por Google. Fue desarrollado por JetBrains en 2011. Poco a poco, va sustituyendo a Java en aplicaciones para móvil. Lo más característico, es que soporta una programación funcional y las famosas corrutinas. Tenemos integridad en desarrollo software con Kotlin/Ktor, tanto en el front como en el back.
3. **Jre** Recordemos cómo integramos java en nuestro sistema. Jre (Java Runtime Enviroment), engloba a nuestra JRE (máquina virtual) encargada de interpretar y ejecutar todo el bycode, que se ha generado despues de la compilación por parte del JDK (javac). Cuando hablamos de que java es multiplataforma, tenemos que entender que la máquina virtual, debemos de instalarla en nuestro sistema dependiendo del tipo de sistema operativo que tengamos y por supuesto de la arquitectura hardware de nuestro procesador. Esto es así, porque cada S.O, entre otras muchas gestina los mapeos de memoria ppal de una forma u otra, así pues Windows no lo hace igual que OS o que Linux y viceversa. También sabemos, que el repertorio de instrucciones de un procesador es totalmente diferente, así pues un Intel y un AMD no tendrá el mismo. Pues bien, cuando queremos poner en marcha un entorno de desarrollo con Java, tendremos que instalarnos la máqina virtual específica. Cuando el jre interpreta el bycode, éste genera instrucciones directamente en la arquitectura, por eso es multiplataforma.
4. **Jdk** Jdk se conoce como el kit de desarrollo software y ¿Eso qué es? Como dice la palabra, contendrá lo necesario para poder desarrollar aplicaciones informáticas. Por ejemplo, un compilador de java (javac) y todo el conjunto de librerías que el lenguaje incorpora. La versión del jdk, está totalmente vinculada a la versión del lenguaje java, así pués os pondré un análisis sencillo. Si en un sistema utilizamos una versión 17, querrá decir que podemos utilizar todo el potencial que nos brinda la versión de java y si incluímos una librería moderna, nuestro compilador generará el bycode correspondiente a dicha versión. Recordemos que si tenemos otro jdk inferior, en el caso de que tengamos que volver a compilar nuestra aplicaciones, la librería utilizado por la 17 no será compilada en una versión inferior. En el caso de kotlin, es algo diferente porque el compilador no se encuentra dentro del jdk pero sin embargo, (kotlinc) debe generar bycode compabible con el jdk ya que hemos dicho que un programa escrito en kotlin, debe ser perfectamente compatible con la máquina virtual y por tanto con jdk. Las librerías o características que utilizemos en kotlin, deben ser compatibles con la versión de jdk, aunque el lenguaje kotlin incorpore una funcionalidad superior a la de java.
5. **Sdk** El Sdk, se conoce como el conjunto de herramientas necesarias para el desarrollo de una aplicación informácica. Aquí tenemos por ejemplo el IDE y en el caso de aplicaciones para móvil con Android studio, se relaciona con el nivel de API que deberá ser completamente compatible con la versión del sistema operativo (Android). Así pués, una aplicaciones con nivel mínima de api 34, no podrá ser ejecutada e instalada en un Android 33. Esto se conoce como API Level. Os recuerdo que la versión del sdk, no está directamente relacionado con la versión del jdk, pero si utiliza el jdk para compilar en Java/Kotlin. Cuando creemos un proyecto Android, lo primero

que nos pedirá será la versión o nivel de API (por ejemplo 33). Debemos asegurarnos, que nuestro Sdk soporte dicho nivel de API. Si aumentamos el nivel de API (por ejemplo de 32 a 34), es muy probable que tengamos también que actualizar nuestro SDK para que incluya las herramientas y bibliotecas necesarias para ese nivel.



VARIABLES PRIMITIVAS EN KOTLIN

Este apartado es el más sencillo de todos, ya que, con estos dos ejemplos, puedo resumir muy brevemente cómo se declaran las variables primitivas en Kotlin.

Declaración tipo entero.

Declaración de variables tipo entero.

e1.1.kt

```
1 fun main() {
2
3     val a = 1       //No se puede modificar
4     var b = 2       //Identificamos el tipo por el valor
5     val c = a + b
6
7     var d : Int = 0 //Inicializamos al mismo tiempo
8     var e: Int      //No es necesario inicializarlo.
9
10    //b = 2.9       //Esto no puedo hacerlo, porque ya se declaró como entero.
11    val f = 20.9
12    b = f.toInt()   //Hacemos un casting
13    print ("El valor de f es $f y el de b es $b. También puedo poner la suma: ${
  b+f.toInt() }")
14 }
```

Resultado de la ejecución

```
El valor de f es 20.9 y el de b es 20. También puedo poner la suma: 40
```

Declaración tipo reales, booleanos y cadenas.

Ejemplo para la declaración de variables reales, booleanos y cadenas.

e1.2.kt

```
1 fun main() {
2
3     val myInt : Int = 1             //definimos variable de tipo entero y lo
  inicializamos a 1
4     val myInt2 = 2                  //por defecto, kotlin entiende que es de tipo
  entero
5     val myInt3 = myInt * myInt2     //ya sabemos el resultado
6
7     val myDouble : Double = 1.0     //definimos variable de tipo real
8     val myDouble2 = 2.0
```

```
 9     val myDouble3 : Double
10     val myDouble4 = 3               //realmente es de tipo entero
11
12     myDouble3 = myDouble + myDouble4 * myDouble2  //kotlin hace las conversiones
   directamente
13
14     val myString = "Santi"
15     val myString1 : String = "soy " + myString
16
17     val myBool = true
18     var myBool1 = myBool && true
19     myBool1 = myBool1 && false
20
21     println("Estamos repasando de kotlin y $myString1")
22     println("Tipos enteros: $myInt3")
23     println("Tipos reales: $myDouble3")
24     println("Tipos booleanos: $myBool1")
25 }
```

Resultado de la ejecución

> Estamos repasando de kotlin y soy Santi Tipos enteros: 2 Tipos reales: 7.0 Tipos booleanos: false

Os recuerdo que la diferenci entre un Double y un Float está en la cantidad de memoria que se reserva para ambos tipos. Mientras que un Float se destinan 32 bits de espacio en memoria, un Double son 64 bits. En el float, es necesario añadir en el valor la letra **f**. Los Doubles se utilizan cuando queremos precisión en los resultados y los Float cuando el ahorro de memoria sea crítico.

e1.e3.kt

```
1 fun main() {
2     val myDouble: Double = 3.141592653589793
3     val myFloat: Float = 3.1415927f  // El sufijo 'f' es necesario para definir un
   Float
4
5     println("Valor de Double: $myDouble")  // Imprime: Valor de Double:
   3.141592653589793
6     println("Valor de Float: $myFloat")    // Imprime: Valor de Float: 3.1415927
7 }
```

Si quremos convertir de Double a Float, necesitamos utilizar el método **toFloat()**. Si queremos convertir de Float a Double, utilizamos la función **toDouble()**.

e1.e4.kt

```kotlin
val myDouble: Double = 9.87654321
val myFloat: Float = myDouble.toFloat()  // Convertir de Double a Float
println("Double a Float: $myFloat")  // Imprime: Double a Float: 9.876543
```

e1.e4.kt

```kotlin
val myFloat: Float = 3.14159f
val myDouble: Double = myFloat.toDouble()  // Convertir de Float a Double
println("Float a Double: $myDouble")  // Imprime: Float a Double: 3.141590118408203
```

Anulables

En Kotlin, los tipos anulables son una característica importante que permite a las variables y propiedades tomar el valor `null`. Esta característica es especialmente útil para evitar errores de referencia nula, un problema común en muchos lenguajes de programación. A continuación, se detallan los conceptos clave relacionados con los tipos anulables.

Declaración de Tipos Anulables

Para declarar una variable o propiedad que puede ser nula, se debe usar el operador `?` después del tipo. Esto indica que la variable puede contener un valor del tipo especificado o `null`.

e1.e5.kt

```kotlin
  var nombre: String? = null
  var edad: Int? = 25
```

En este ejemplo, nombre puede ser un String o null, y edad puede ser un Int o null.

Comprobación de Nulidad

Puedes comprobar si una variable anulable es null usando una simple condición if.

e1.e6.kt

```kotlin
if (nombre != null) {
    println("El nombre es: $nombre")
} else {
    println("El nombre es nulo")
}
```

Operador Elvis (`?:`)

El operador Elvis se usa para proporcionar un valor predeterminado cuando una expresión es `null`.

e1.e7.kt

```
val longitudNombre = nombre?.length ?: 0
println("La longitud del nombre es: $longitudNombre")
En este ejemplo, si nombre es null, longitudNombre será 0.
```

Operador de Acceso Seguro (?.)

El operador de acceso seguro se usa para llamar a un método o acceder a una propiedad solo si la variable no es null. Si la variable es null, la operación se detiene y el resultado es null.

e1.e8.kt

```
val longitudNombre = nombre?.length println("La longitud del nombre es:
$longitudNombre")
```

En este ejemplo, si `nombre` es `null`, `longitudNombre` será `null`.

Operador de Afirmación de No Nulidad (!!)

El operador `!!` se usa para afirmar que una variable no es `null`. Si la variable es `null`, se lanzará una excepción `NullPointerException`.

e1.e9.kt

```
val longitudNombre = nombre!!.length
println("La longitud del nombre es: $longitudNombre")
```

NOTE Este operador debe usarse con precaución, ya que puede causar errores en tiempo de ejecución si el valor es null.

Funciones Anulables

Kotlin permite definir funciones que pueden devolver un valor anulable. Esto se hace especificando el tipo de retorno como anulable.

e1.e10.kt

```
fun obtenerNombre(): String? {
    return null
}

fun obtenerNombreConPredeterminado(): String {
    return obtenerNombre() ?: "Devolverá siempre Nombre Predeterminado, porque
siempre retornara null"
}
```

Uso más extendido de la comprobación de nullables en kotlin.

En Kotlin, la función de extensión `let` se utiliza para ejecutar un bloque de código solo si el objeto no es `null`, cuando utilizas el operador seguro **?.**. Esto es útil para manejar valores opcionales y evitar el uso excesivo de comprobaciones `null`. A continuación se muestra un ejemplo que utiliza `let` y su alternativa sin `let`.

e1.e11.kt

```kotlin
fun procesarNombre(nombre: String?) {
    nombre?.let {
        println("El nombre es $it")
    } ?: run {
        println("El nombre es null")
    }
}

fun main() {
    val nombre1: String? = "Santi"
    val nombre2: String? = null

    procesarNombre(nombre1)  ①
    procesarNombre(nombre2)  ②
}
```

① Imprimirá el nombre Santi

② Imprimirá el nombre es null.

El uso de **let** sobre un objeto, es ejecutar un bloque de código con el objeto como argumento (it). No sólo lo podemos ver con nullables, es común encontarnos encadenamientos de múltiples operaciones con el mismo objeto:

```kotlin
fun main() {
    val yo = "Santiago Rodenas Herráiz"

    yo.let {
        it.toUpperCase()
    }.let {  nombreMayus -> //sobre ese string convertido a mayúsculas

        val partes = nombreMayus.split(" ")
        val nombre = partes[0]
        val apellido1 = partes[1]
        val apellido2 = partes[2]
        Triple(nombre, apellido2, apellido1)  //devuelvo el objeto Triple pero con los apellidos al revés.
    }.let { //sobre ese objeto Triple
        print ("Mi nombre con el apellido cambiado es ${it.first}, ${it.second}, ${it.third}")
    }
```

```
16 }
```

Analizar esto e indicar si tiene sentido y por qué:

```
1 val nombre : String? = null
2 nombre.let{
3     print ("Longitud ${it.length}")  ①
4 }?: run {                           ②
5     print ("No tiene mucho sentido")
6 }
```

① Hay una incoherencia, porque al no utilizar el operador seguro ?., el compilador no sabe bien, si nombre es null y es obligación tuya de indicar que el nombre no es nulo, utiliando el operador !!. Cláro está, si ponemos ${it!!.length}, probocaremos un error de NullPointerException.

② El uso de **?:run** dentro de **objeto?.let**, tiene sentido cuando queremos verificar con el operador elvis qué hacer cuando es nulo. En el caso de utilizar let sin nullables como en el ejemplo de antes, que concatena acciones sobre el mismo objeto, no tiene ningún sentido utilizar **?:run**.

```
Resumen
```

- **Declaración de Tipos Anulables:** Usa `?` después del tipo para permitir que una variable o propiedad sea `null`.
- **Comprobación de Nulidad:** Usa condiciones `if` para verificar si una variable es `null`.
- **Operador Elvis (`?:`):** Proporciona un valor predeterminado si una expresión es `null`.
- **Operador de Acceso Seguro (`?.`):** Accede a métodos y propiedades solo si la variable no es `null`.
- **Operador de Afirmación de No Nulidad (`!!`):** Asegura que una variable no sea `null`, lanzando una excepción si lo es.
- **Funciones Anulables:** Las funciones pueden devolver valores anulables especificando el tipo de retorno como anulable.

Esta documentación te ayudará a entender y manejar adecuadamente los tipos anulables en Kotlin, evitando errores comunes relacionados con valores `null`.

ACTIVIDADES

1. **Declaración de variables enteras**: Escribe un programa que declare varias variables enteras (`val` y `var`), realiza operaciones básicas con ellas y muestra el resultado en la consola.
2. **Declaración de variables reales, booleanas y cadenas**: Crea un programa que declare variables de tipo `Double`, `Float`, `Boolean` y `String`. Realiza operaciones con estas variables y muestra los resultados en la consola.
3. **Diferencia entre `Double` y `Float`**: Escribe un programa que declare una variable de tipo `Double` y una de tipo `Float`. Muestra ambos valores en la consola para observar la diferencia de precisión.
4. **Conversión de `Double` a `Float`**: Crea un programa que convierta un valor de tipo `Double` a `Float` y muestra el resultado en la consola.
5. **Conversión de `Float` a `Double`**: Escribe un programa que convierta un valor de tipo `Float` a `Double` y muestra el resultado en la consola.
6. **Inicialización de variables sin valor**: Desarrolla un programa en el que declares una variable sin inicializarla y luego le asignes un valor. Usa esta variable en una operación simple.
7. **Casting de tipos numéricos**: Realiza un programa que convierta un valor de tipo `Double` a `Int` usando casting y muestra el valor convertido junto con el original.
8. **Operaciones con cadenas**: Escribe un programa que concatene dos cadenas de texto y muestre el resultado en la consola.
9. **Uso de `Boolean` en condiciones**: Crea un programa que use una variable `Boolean` en una expresión condicional para mostrar mensajes diferentes según el valor de la variable.
10. **Declaración y uso de `var` y `val`**: Escribe un programa que declare variables utilizando tanto `var` como `val`. Modifica el valor de las variables `var` y muestra cómo cambian en comparación con las variables `val` que son inmutables.
11. **Valor Predeterminado con Elvis**: Define una función que recibe un parámetro de tipo `String?` y devuelve un `String` que es el valor del parámetro si no es `null`, o un valor predeterminado si es `null`. Utiliza el operador Elvis (`?:`) para proporcionar el valor predeterminado.
12. **Uso de `let` para Procesar Valores No Nulos**: Crea una función que reciba un parámetro de tipo `Int?`. Si el parámetro no es `null`, utiliza `let` para imprimir el doble del valor. Si el parámetro es `null`, imprime un mensaje que indique que el valor es `null`.
13. **Uso de ?:run sólo cuando existe un objeto?.let**
14. **Uso de let para Encadenar operaciones** sobre el mismo objeto y poder realizar una serie de acciones antes de devolver el resultado antes del cierre **}** del bloque let.
15. **Filtrado de Valores Nulos en Listas**: Define una función que recibe una lista de `String?` y devuelve una lista de `String` filtrada que contiene solo los elementos no nulos. Utiliza la función `filterNotNull()` para obtener la lista de valores no nulos.
16. **Operaciones con Valores Nulos en Cadenas**: Crea una función que recibe una cadena de tipo `String?` y devuelve el número de caracteres de la cadena si no es `null`. Si la cadena es `null`, devuelve `0`. Utiliza el operador `?.let` para realizar la operación si la cadena no es `null`.
17. **Manejo de Nulos en Funciones de Cálculo**: Define una función que reciba dos parámetros de tipo `Int?`. Si ambos parámetros no son `null`, devuelve la suma de los dos valores. Si al menos

uno de los parámetros es `null`, devuelve un valor predeterminado que indique que uno o ambos valores eran nulos.

ESTRUCTURAS DE CONTROL Y REPETICIÓN

Mostraremos algunos ejemplos sencillos, para repasar la sintáxis de las siguientes estructuras de control:

Sentencias de control

1. Condicionales
    a. Condicional simple
    b. Condicional doble
    c. Múltiple
2. Repetitivas
    a. While
    b. Do-While
    c. For

Condicionales compuesta

Sentencias condicionales if-else-if-else

e2.1.kt

```
fun main() {
    val myInt = 9

    if (myInt < 0)
        println ("Numero negativo , es $myInt")
    else
        if (myInt <=10 && myInt != 5)
            println ("Numero entre 0 y 10 y distinto de 5 es, $myInt")
        else
            if (myInt == 5)
                println ("Número igual a 5")
            else
                println "Número mayor que 10 es, $myInt"
}
```

Condicionales múltiles When

En java, lo conocemos por medio del switch mientras que kotlin, utiliza un when mucho mas expresivo.

e2.2.kt

```
fun main() {
    val pais : String = "España"
    var moneda = ""

    when(pais) {        ①
        "España" -> {
            moneda ="Euro"
        }
        "Francia" -> {
            moneda ="Euro"
        }
        "Alemania" -> {
            moneda ="Euro"
        }
        "EEUU" -> {
            moneda ="Dolar"
        }
        "Italia" -> {
            moneda ="Euro"
        }
        "Venezuela" -> {
            moneda ="Bolibar"
        }
        else -> {
            moneda = "N.I."
        }
    }

    println("La moneda del pais $pais es $moneda")

    when(pais) {        ②
        "España", "Francia", "Alemania" , "Italia" ->
            moneda ="Euro"
        "EEUU" ->
            moneda ="Dolar"
        "Venezuela" ->
            moneda ="Bolibar"
        else ->
            moneda = "N.I."
    }

    val sueldo = 1000
    when (sueldo) {     ③
        in 700 .. 900 ->
```

```
45              println("Sueldo de 700 a 900")
46          in 901 ..1200 ->
47              println("Sueldo de 901 a 1200")
48          in 1201 .. 2000 ->
49              println("Sueldo de menos de 2000")
50          else ->
51              println("Otro sueldo")
52      }
53  }
```

① Evaluamos por un valor

② Evaluamos por un conjunto de valores

③ Evaluamos por rango.

Bubles

En el while y do-while, no existen diferencias significativas con respecto a java. Lo recordaremos con un par de ejemplos.

e2.3.kt (while y do-while)

```
1  fun main() {
2      var x=0
3      while (x<10){
4          print (" $x ")
5          x+=2
6      }
7      println("\nAhora do-while")
8
9      //Bucle do while. Igual que en java.
10     x=0
11     do{
12         print (" $x ")
13         x+=2
14     }while(x<10)
15 }
```

e2.4.kt (for incremental)

```
1  fun main() {
2      var suma = 0
3
4      for(i in 1..10) {
5          print("Ingrese un valor:")
6          val valor = readLine()!!.toInt()
7          suma += valor
8      }
9
10     println("La suma de los valores ingresados es $suma")
```

```
11     val promedio = suma / 10
12     println("Su promedio es $promedio")
13 }
```

e2.5.kt (for incremental en dos pasos)

```
1 fun main() {
2     var suma = 0
3     println("Contando números pares de 0 a 10 y calculando su suma:")
4
5     for (i in 0..10 step 2) {       ①
6         println("Número par: $i")
7         suma += i
8     }
9
10    println("La suma de los números pares es: $suma")
11 }
```

① A diferencia de java, tenemos que utilizar la palabra reservada **step** junto al valor del paso. Vemos también, como se utilizan los rangos de datos *inicio..fin*.

e2.6.kt (for decremental en dos pasos)

```
1 fun main() {
2     println("Tiempo para la explosión de dos en dos:")
3
4     for (i in 10 downTo 0 step 2) {     ①
5         println("Contamos... Estado actual: $i")
6     }
7
8     println("¡BUMMMMMMM!")
9 }
```

① Hay diferencia en comparación con java. Podemos dar a un error, intenando hacer algo parecido a esto: **for (i in 10..0 step 2)**. Cuidado, porque es totalmente contradictorio. Mientras que *10..0* es un rango decremental e iría hacia abajo, *step 2* indica que se le suma 2. Por tanto, no haría nada.

ACTIVIDADES

1. **Condicional simple**: Escribe un programa que pida al usuario un número y que imprima si es **positivo**, **negativo** o **cero**.
2. **Condicional doble**: Crea un programa que pregunte al usuario su edad y le indique si es **mayor de edad** o **menor de edad**.
3. **Condicional múltiple**: Escribe un programa que reciba una calificación de 0 a 10 y muestre si es **suspenso**, **aprobado**, **notable** o **sobresaliente**, según el valor de la calificación.
4. **Uso de `when`**: Crea un programa que reciba el nombre de un día de la semana e imprima si es un **día laboral** o un **día de descanso** (sábado o domingo) usando `when`.
5. **Bucle `while`**: Realiza un programa que imprima los **primeros 10 números pares** utilizando un bucle `while`.
6. **Bucle `do-while`**: Crea un programa que pida al usuario un número y sume todos los números desde 1 hasta ese número, mostrando el resultado final. El programa debe continuar pidiendo números hasta que el usuario introduzca un número negativo.
7. **Bucle `for`**: Escribe un programa que pida al usuario 5 números y luego imprima el **número mayor** de todos ellos.
8. **Bucle `for` con `step`**: Haz un programa que cuente desde el número **20** hasta el número **0**, de **dos en dos**, imprimiendo cada número por pantalla.
9. **Uso de rangos y `in` con `when`**: Realiza un programa que reciba un número de 1 a 12 y muestre el nombre del **mes** correspondiente (por ejemplo, 1 = Enero, 2 = Febrero, etc.).
10. **Bucle `for` con condición**: Escribe un programa que pida al usuario un número y determine cuántos de los números entre 1 y ese número son **divisibles por 3**.

ARRAYS

Existen diferentes formas de trabajar con los arrays y en la mayoría de los casos, en su inicialización de valores nos decantaremos por las **expresiones lambda**. Recordemos como java declara un array, con respecto a como lo puede hacer kotlin:

```
1 int[] arr = new int[5];     ①
2 String[] names = {"Santi", "Sonia", "Guille", "Diego"};
3
4
5 //En kotlin
6 val arr = IntArray(5)        ②
7 val names = arrayOf("Santi", "Sonia", "Guille", "Diego")
```

① Declaración de un array en java

② Declaración de un array en kotlin. arrayOf tendrá un significado especial, ya que es un array inmutable.

Pregunta: ¿Qué diferencia hay entre utilizar var o val en un array?

Declaración e inicialización de arrays.

Como hemos indicado anteriormente, la mayoría utilizaremos arrays inicializados a valor 0 o inicializados con valores prestablecidos. A continuación mostramos un ejemplo muy sencillo de varias.

e3.1.kt

```
1 fun main(){
2     val myArray = arrayOf("lunes", "Martes", "Miercoles", "jueves", "Viernes",
  "Sabado", "Domingo")                                    ①
3     val martes = myArray[1]                                ②
4     val miercoles = myArray.get(2)                         ③
5     myArray[3] = "Jueves"                                  ④
6     myArray.set(0, "Lunes")
7
8     myArray.forEach {                                      ⑤
9         if (it == "Sabado")
10            println("Sabado, el mejor día de la semana")
11        else
12            println(it)
13    }
14
15    for (a in myArray)
16        if (a == "Domingo")
17            println("El domingo, día de reunirse con la familia")
18        else
19            println (a)
```

```
20
21      var myArray2 = 0..10                                    ⑥
22
23      for (x in myArray2)
24          println(x)
25
26      for (i in 0..myArray.size -1)                           ⑦
27          print(myArray[i]+" ")
28
29      for (i in 0..myArray.size -1 step 2){                   ⑧
30          print(myArray[i]+" ")
31      }
32
33      for (i in 2 until  myArray.size -1){                    ⑨
34          print(myArray[i]+" ")
35      }
36
37      for (i in myArray.size -1 downTo 0 )                    ⑩
38          print(myArray[i]+" ")
39
40      for (pos in myArray.indices)                            ⑪
41          println(myArray.get(pos)+" ")
42
43
44      for ( (pos, valor) in myArray.withIndex())              ⑫
45          println("La posicion $pos tiene de valor $valor")
46
47  }
```

① Los arrays son de tamaño fijo. No pueden añadirse más elementos una vez definidos. Si podemos cambiar sus valores en posición.

② Manera clásica de acceder a un elemento del arrays, igual que en java. Internamente, el acceder a **[]** se invoca al método get indicado en el punto 3.

③ Accedemos al valor igual que en el punto 2, pero se hace de manera más explícito. Podemos incluso sobrecargar el método get, por tanto también cambiará la forma de acceso del punto 2. Utilizado en las clases.

④ De la misma forma que accedemos a los valores según posición o índice, también es posible modificar dichos valores, aplicando los mismos conceptos del punto 2 y 3.

⑤ forEach, es la función incorpora los arrays, por excelencia para recorrer cada uno de sus valores, de los cuales pasamos como argumento una expresión o función, que será invocada dentro del forEach, elemento por elemento. Es el claro ejemlo de una programación funcional. Más adelante lo trabajaremos. El for que aparece debajo del forEach, es más conocido y sencillo de interpretar.

⑥ Me declaro y defino un rango de 11 elementos. No confundir un rango de datos con un array. Se pueden recorrer mediante un **for a in Rango**, pero no intentéis hacer un acceso por **[]**, no posee el método get.

⑦ Lo más parecido a otros lenguajes imperativos. for i=0; i⇐7; i++

⑧ Lo más parecido a for i=0; i⇐ 7; i+2

⑨ Lo más parecido a for i =2; i⇐7; i++

⑩ Lo más parecido a for i=7; i>=0; i--

⑪ Bucle for. (Recorremos por índice)

⑫ Bucle for. (Recorremos por índice→valor). Muy utilizado, cuando queremos sonsacar tanto su índice como el valor que encierra.

Funciones integradas en Kotlin

Kotlin, incorpora una cantidad de funciones integradas para trabajar con los arrays de manera sencilla y funcional. En el mundo de la programación imperativa, cuando queremos realizar un filtro de valores en un array, necesitamos indicar el cómo hacerlo, mientras que la programación funcional es justamente lo contrario, indicar qué es lo que queremos sin indicar como hacerlo.

Kotlin cuenta con un gran número de funciones de los cuales sólo mostraré una pequeña parte de ellas:

- Funciones para crear arrays
- Funciones para acceder a sus valores
- Funciones básicas de iteración
- Funciones de búsqueda y filtrado
- Funciones de agregación
- Funciones de ordenación
- Funciones de conversión
- Funciones de transformación

Funciones para Crear Arrays

Ya hemos visto algunas formas de crear un array, pero las más comunes las encontrados con:

e3.2.kt

```
1 val myArray1 = arrayOf(1, 2, 3.3, "santi")   //Creamos e inmicializamos un array de
  elementos genéricos, de cualquier tipo. Any
2 val myArray2 = intArrayOf(1, 2, 3, 4)   //Lo mismo que antes, pero especificamos un
  array de sólo enteros.
3 val myArray3 = doubleArrayOf(1.4, 2.6)  //Sólo reales
4 val myArray4 = Array(5) { it*2 }        //Objeto Array de 5 elementos y los
  inicializamos multiplicando su índice por 2. Sólo inicializamos con lambda.
5 val myArray5 = Array(5) { index->       //Detro de su lambda, podemos realizar una
  serie de comprobaciones.
6         when (index){
7             0 -> 1.4
8             1 -> 2.5
9             3 -> 5.3
```

```
10            else -> 0.0
11        }
12    }
13
14 myArray5.forEach { elemento -> println (elemento) }
```

La diferencia entre **arrayOf** y **intArrayOf/doubleArrayOf**, es que el primero es genérico de elementos de tipo **Any**. En kotlin, el object de java es Any, por tanto los elementos pueden ser de diferente tipo ya que Any es la superclase de todos. La diferencia entre Array y intArrayOf/doubleArrayOf/arrayOf/etc, está en la forma que tenemos de inicializar sus elementos, ya que lo podemos hacer mediante una **lambda**. Es el caso de myArray4 y myArray5.

Pregunta: ¿Qué diferencia hay entre estas dos formas de crearnos e inicializar un array?

```
1 val myArray = IntArray(5) {it}
2 val myArray1 = intArrayOf( 0,1,2,3,4 )
```

Funciones para acceder a sus valores

Ya hemos visto como acceder mediante **[]** o utilizando el método **get** de la variable de tio array. También hemos visto como podemos modificar el valor en una posición determinada del array con **[]** o utilizando el método **set(indice, valor)**

Funciones básicas de iteración

Las funciones más utilizadas para su iteración son el **forEach** y el **forEachIndexed**. Son funciones en los que pasaremos expresiones lambda. Ya podemos intuir, que cuando la expresión lambda utiliza un parámetro, podemos no redefinir su nombre utilizando **it**, mientras que se existen varios parámetros, necesitamos redefinir las variables. Este es el caso de forEachIndexed, en el que la llamada de orden superior, utiliza dos parámetros que redefinimos en la expresión con los nombres indice y valor. Para ello, anteponemos la →.

e3.3.kt

```
1 myArray.forEach { println(it) }
2 myArray2.forEachIndexed { indice, valor -> println("Elemento en $indice: $valor") }
```

Funciones básicas de búsqueda y filtrado

Con filter, seleccionamos aquellos elementos de acuerdo a un criterio. En el caso de las llamadas a orden superior (más adelante), veremos como a partir de un parámetro booleano, podemos seleccionar aquellos elementos que deseemos. Os mostraré un ejemplo de filtrado muy sencillo.

e3.4.kt

```
1 val myArray = IntArray(20){ it }              ①
2 val pares = myArray.filter { it % 2 == 0 }    ②
```

```
3 val index = myArray.indexOf(3)                    ③
4 val exists = myArray.contains(4)                  ④
```

① Creamos un array de 20 posiciones y lo inicializamos según el indice con valores de 0 a 19.

② Filtramos según la expresión booleana que indicamos. Creará un array con los valores según la expresión.

③ Retorna el índice del array o en caso de no encontrarlo, un -1.

④ Devuelve un booleano, dependiendo de si encuentra un valor o no.

Funciones de agregación

Estas funciones, nos ahorra el tiempo en aspectos como sumar todos los elementos, la media, encontrar el máximo valor o mínimo valor.

e3.5.kt

```
1 val total = myArray.sum()
2 val avg = myArray.average()
3 val max = myArray.maxOrNull()
4 val min = myArray.minOrNull()
```

Funciones de ordenación

En el paradigma imperativo, uno de los aspectos más complejos es la ordenación de un array. Existen diferentes métodos que hoy en día ya no se utilizan porque existen funciones agregadas que te ahorran su lógica. Los dos ejemplos más sencillos son los de ordenar de manera incremental o decremental.

e3.6.kt

```
1 fun main() {
2     val myArray = intArrayOf(5, 2, 9, 1, 7)
3     val sortedArray = myArray.sorted()                          ①
4     println("Ordenado de menor a mayor: $sortedArray")
5     val sortedArrayDescending = myArray.sortedDescending()      ②
6     println("Ordenado de mayor a menor: $sortedArrayDescending")
7 }
```

① Ordenamos de menor a mayor. 1,2,5,7,9

② Ordenamos de mayor a menor. 9,7,5,2,1

Funciones de conversión

Este tipo de funciones, podemos utilizarlo para pasar de un array a una lista con los valores iniciales pero con la diferencia de que en una lista, ésta puede o no crecer ya que es totalmente dinámica. Algunos de los métodos más utilizados, será **.toList()**, **.toMutableList()**. La diferencia es que toList es inmutable mientras que toMutableList es una lista mutable, es decir que podemos

insertar o eliminar elementos dinámicamente. Existen otras conversiones como a conjunto **.toSet()** o **.toMap()**

e3.7.kt

```
fun main() {
    val myArray = intArrayOf(5, 2, 9, 1, 7)
    val listFromArray = myArray.toList()                    ①
    println("Convertido a List: $listFromArray")

    val mutableListFromArray = myArray.toMutableList()      ②
    println("Convertido a MutableList: $mutableListFromArray")

    val setFromArray = myArray.toSet()                      ③
    println("Convertido a Set: $setFromArray")
}
```

① Convertimos a una lista inmutable, con sus típicos métodos get, put, size, etc.

② Se convierte a una lista mutable.

③ Se eliminan los elementos duplicados.

e3.8.kt

```
fun main() {
    val pairsArray = arrayOf(
        "Santi" to 25,
        "Sonia" to 30,
        "Guille" to 15,
        "Diego" to 10
    )                                                       ①

    val mapFromArray = pairsArray.toMap()
    println("Convertido a Map: $mapFromArray")

    val anotherMap = mapOf(
        "Santi" to 25,
        "Sonia" to 30,
        "Guille" to 15,
        "Diego" to 10
    )                                                       ②
    println("Otro Map creado directamente: $anotherMap")
}
```

① clave **to** valor, representa una estructura de datos llamada Pair(clave, valor). Con **to**, creamos un **Pair**. En este punto, lo que hacemos es devolver un mapa compuesto con valores (clave, valor), especificados en los creados mediante arrayOf. Recordemos que arrayOf se declara del tipo **Array<Pair<String,Int>>**. Por tanto, se devuelve un Map a partir de un array de tipo Pair.

② Es otra forma de crearnos un mapa a partir de un array de pares.

e3.9.kt

```
val myArray : Array<Pair<String, Int>> = arrayOf(
        "Santi" to 25,
        "Sonia" to 30,
        "Guille" to 15,
        "Diego" to 10
    )
val myMap : Map<String, Int> = myArray.toMap()
myMap.forEach{ clave, valor ->  println("Clave: $clave, Valor: $valor")}
```

Funciones de Transformación

Los casos mas sencillos, es transformar un array en otro con valores cambiados. Lo que hacemos con map, es aplicar una expresión por cada elemento del array pero devolvemos una colección **List**. Podemos entender, que al hacer un map sobre un array, todos sus valores se cambian y NO, ya que no es lo mismo un List() que un Array(). Si queremos devolver nuestro map en un array de nuevo, necesitaríamos utilizar el método **toTypedArray()**.

e3.10.kt

```
var myArray = Array(5){ it + 1 }
val myList = myArray.map { it * 2 }
myArray = myList.toTypedArray()
val final = myArray.mapIndexed { index, value -> index * value }.toTypedArray()
final.forEach{ println (it)}
```

El ejemplo siguiente, utiliza map para aumentar en un 20% el valor de cada uno de los precios de una serie de productos:

e3.11.kt

```
fun main() {
    val originalPrices = doubleArrayOf(100.0, 150.0, 200.0, 250.0)
    val increasedPrices = originalPrices.map { price -> price * 1.20 }

    println("Precios originales: ${originalPrices.joinToString(", ")}")
    println("Precios incrementados en un 20%: $increasedPrices")
}
```

Dos cosas a apreciar. En kotlin, puedo imprimir todos los elementos de una **lista** con un print, pero un **array** NO. Para ello, utilizamos el método **jointToString(", ")** para imprimir con la separación que queramos, elemento a elemento. <<< === ACTIVIDADES

1. **Declaración de arrays en Kotlin**: Escribe un programa que declare un array de enteros y un array de cadenas en Kotlin. Inicializalos con algunos valores y muestra estos valores en la consola.
2. **Acceso y modificación de elementos en un array**: Crea un programa que declare un array de

cadenas, acceda a algunos de sus elementos utilizando índices, y modifique algunos de esos elementos. Muestra el array completo después de las modificaciones.

3. **Recorrido de arrays con `forEach` y `forEachIndexed`**: Desarrolla un programa que use `forEach` para recorrer un array de enteros e imprimir cada elemento. Luego, utiliza `forEachIndexed` para recorrer el mismo array e imprimir cada elemento junto con su índice.
4. **Uso de `for` para recorrer arrays**: Crea un programa que recorra un array de enteros usando un bucle `for` tradicional y un bucle `for` con `step`. Imprime los valores del array en la consola.
5. **Declaración de arrays con rango**: Escribe un programa que declare un rango de números del 0 al 10 y lo recorra usando un bucle `for`. Imprime cada número en la consola.
6. **Funciones básicas de búsqueda en arrays**: Desarrolla un programa que declare un array de enteros, y luego use las funciones `filter`, `indexOf`, y `contains` para filtrar pares, encontrar el índice de un elemento y verificar la existencia de un valor, respectivamente.
7. **Funciones de agregación en arrays**: Crea un programa que declare un array de enteros y utilice las funciones `sum`, `average`, `maxOrNull`, y `minOrNull` para calcular la suma, la media, el valor máximo y el valor mínimo del array.
8. **Ordenación de arrays**: Desarrolla un programa que declare un array de enteros y lo ordene en orden ascendente y descendente utilizando las funciones `sorted` y `sortedDescending`. Imprime los resultados en la consola.
9. **Conversión de arrays a listas y conjuntos**: Escribe un programa que declare un array de enteros y lo convierta en una lista inmutable, una lista mutable y un conjunto. Imprime cada resultado en la consola.
10. **Transformación de arrays**: Crea un programa que use `map` para transformar un array de enteros multiplicando cada valor por 2. Luego, usa `mapIndexed` para crear un nuevo array donde cada valor es el producto del índice por el valor original. Imprime los resultados. Recordar que map, devuelve una lista, no un array.

FUNCIONES

Las funciones son una parte esencial de cualquier lenguaje de programación, y en Kotlin no es diferente. Definir funciones en Kotlin es sencillo y puede realizarse de manera concisa gracias a la sintaxis clara que ofrece el lenguaje.

Una función en Kotlin se define con la palabra clave `fun`, seguida del nombre de la función, los parámetros (si los tiene), el tipo de retorno (si corresponde) y el cuerpo de la función.

Ejemplo básico de función

```
fun saludar(nombre: String): String {
    return "Hola, $nombre!"
}
```

En este ejemplo, la función `saludar` toma un parámetro de tipo `String` y devuelve una cadena de texto que contiene el saludo.

Funciones con cuerpo de expresión

En Kotlin, si una función tiene un cuerpo simple, puedes simplificar su definición usando la sintaxis de cuerpo de expresión.

Ejemplo con cuerpo de expresión

```
fun saludar(nombre: String) = "Hola, $nombre!"
```

Esta versión más concisa de la función anterior hace lo mismo, pero sin usar la palabra `return`, ya que el valor devuelto es implícito.

Funciones sin parámetros

No todas las funciones requieren parámetros. Es posible definir funciones que no necesiten entradas y simplemente ejecuten un bloque de código.

Ejemplo de función sin parámetros

```
fun decirHola() {
    println("Hola mundo!")
}
```

Funciones con parámetros opcionales

En Kotlin, es posible proporcionar valores predeterminados a los parámetros de las funciones. Si no se pasa un valor para un parámetro con un valor predeterminado, se utilizará dicho valor por defecto.

Ejemplo de parámetros opcionales

```
fun mostrarMensaje(mensaje: String = "Mensaje por defecto") {
    println(mensaje)
}
```

La salvedad más significativas en kotlin con respecto a java, es que podemos pasar como argumento a una función, otra función. Es la base para entender las llamadas de orden superior que estudiaremos en el siguiente tema.

Paso de argumentos a un función.

En algunos lenguajes de programación, tenemos las funciones y los procedimientos. Sabemos que los procedimientos no devuelven nada, a diferencia de las funciones. En kotlin, el concepto función es la misma que en cualquier lenguaje, sólo que si no devuelve nada, incluímos la palabra Unit en la declaración de la función.

Pondremos algunos ejemplos sencillos.

e4.1.kt

```
1 fun realizaSuma(a: Int, b: Int): Int{
2     return a+b
3 }
4
5 fun realizaSuma2 (a: Int, b: Int) = a+b        ①
6
7 fun main(){
8     println("La suma de 2 y 3 es ${realizaSuma(2,3)}")
9     println("La suma de 2 y 3 es ${realizaSuma2(2,3)}")
10 }
```

① Simplificamos cuando sólo hay una línea en el cuerpo. La primera opción realizaSuma es la tradicional.

Cuando queramos pasar parámetros como objetos o arrays, símplemente indicamos el tipo en su declaración. Recordemos que al igual que en java, lo que se pasan son las referencias a los objetos por tanto hay que tener cuidado si no queremos modificarlos.

e4.2.kt

```
1 fun devuelveSumaArray(myArray: Array<Int>):Int{     ①
2     var suma=0
3     for (i in myArray.indices)
4         suma+= myArray[i]
5
6     return suma
7 }
8
```

```
fun devuelveSumaArray1(myArray: IntRange):Int{      ②
    var suma=0
    for (x in myArray)
        suma+=x

    return suma
}

fun main(){
    val myRange = 1..50
    val myArray = myRange.toList().toTypedArray()   ③
    println ("La suma del array completo es ${devuelveSumaArray(myArray)}")
    println ("La suma del array completo es ${devuelveSumaArray1(myRange)}")
}
```

① El paso de un array por argumento, es mediante referencia. Pasamos un array de tipo entero.

② Pasamos el tipo Rango a la función. No confundir con un array.

③ El método **toList()**, convierte a lista y **toTypedArray()** convierte de lista a Array.

Modificar un parámetro pasado por valor a una función.

Cuando pasamos un parámetro primitivo a una función del tipo 0 Int, Double, Float, Boolean, etc, éstos son pasados por valor, no por referencia. Esto significa que existe una copia tal cual. Veamos el tipo de ejemplo de una variable de tipo entero que intentamos modificarla dentro de una función.

e4.3.kt

```
fun modificarNumero(numero: Int) {
    var copiaNumero = numero
    copiaNumero += 5                                ①
    println("Dentro de la función: $copiaNumero")
}

fun main() {
    val numero = 10
    modificarNumero(numero)
    println("Después de la función: $numero")       ②
}
```

① Modificamos el valor del parámetro pasado. En este caso al recibir un 10 y sumar 5, imprime 15.

② Al pasar el parámetro por valor, aunque se modifique dentro de la función, no tiene visibilidad fuera de ésta, por tanto imprimirá un 10.

Modificar un parámetro pasado como referencia a una función.

Recordemos que al igual que java, cualquier objeto pasado como argumento a una función, se considera el paso por referencia que a diferencia de paso por valor **(copia)**, cualquier modificación de un parámetro por referencia tendrá visibilidad desde fuera de la función. Veamos este ejemplo simple:

e4.4.kt

```
1  import kotlin.random.Random
2
3  fun loadData():  IntArray{                    ①
4      var myArr = IntArray(10)                  ②
5      for (i in myArr.indices)
6          myArr[i] = Random.nextInt(0, 99)
7      return myArr                              ③
8  }
9
10 fun printValuesOfArr (arr: IntArray){         ④
11     for (i in arr.indices)
12         if ( i < arr.size -1)
13             print ("${arr[i]}, ")
14         else
15             print ("${arr[i]}")
16 }
17
18 fun main(){
19     var myArr = loadData()                    ⑤
20     printValuesOfArr(myArr)
21 }
```

① Declaramos el tipo del argumento a devolver, que es un array. Recordar que no hay que especificar el tamaño, sólo el tipo.

② Nos creamos un array local dentro de la función y cargamos sus datos.

③ Recordemos que al retornar una referencia dentro de la función, si ésta es recogida desde fuera el recolector no interviene, por tanto su ciclo de vida sería global.

④ Pasamos como referencia un tipo IntArray sin especificar tamaño.

⑤ Recogemos la referencia del punto 3.

ACTIVIDADES

1. **Definición de funciones**: Crea una función en Kotlin que tome un parámetro de tipo `String` y devuelva un saludo personalizado. Utiliza tanto la forma completa como la forma concisa con cuerpo de expresión.
2. **Funciones sin parámetros**: Escribe una función que no tome ningún parámetro y que imprima "Hola mundo!" en la consola. Llama a esta función desde la función `main`.
3. **Parámetros opcionales**: Define una función que tome un parámetro de tipo `String` con un valor predeterminado. Si no se pasa ningún argumento al llamar a esta función, debe imprimir un mensaje predeterminado. Llama a esta función tanto con un argumento como sin él.
4. **Paso de argumentos a funciones**: Crea dos funciones que reciban un array de enteros y un rango de enteros, respectivamente, y devuelvan la suma de sus elementos. Convierte un rango en un array y usa ambas funciones para calcular la suma de los elementos. Muestra los resultados en la consola.
5. **Modificación de parámetros primitivos**: Desarrolla una función que reciba un parámetro de tipo `Int`, lo modifique dentro de la función y luego imprima el valor modificado. Llama a esta función desde `main` y muestra que el valor original no ha cambiado.
6. **Modificación de parámetros de tipo referencia**: Escribe una función que cree un array de enteros con valores aleatorios, devuelva el array y otra función que reciba el array como parámetro y lo imprima. Llama a estas funciones desde `main` y muestra que el array se modifica fuera de la función que lo crea.

FUNCIONES LAMBDA

Este punto, es quizás la parte más interesante del lenguaje kotlin y en la que nos centraremos mayormente, ya que en el mundo de la programación Android, es impensable no utilizar las expresiones lambda y llamadas a funciones de orden superior, sobre todo para el manejo de eventos, en los cuales en java sólo lo podemos hacer por medio de las interfaces y en nuestro caso tendremos además, la posibilidad de utilizar las llamadas a funciones de orden superior.

Estructura

1. Concepto de referencia a una función.
2. Invocar a una función, mediante una variable.
3. Concepto de expresión lambda.
4. Pasar como parámetro a una función, la referencia de otra función.
5. Llamadas a funciones de orden superior.
    a. Ejemplos.

Concepto de referencia a una función.

Quizás es algo complejo de entender, porque la programación imperativa de siempre se ha compuesto de una declaracion/definición de función **(se aportan los parámetros, cuerpo y retorno)** y ésta es invocada las veces que se quiera desde cualquier parte de tu aplicación. La idea ahora, es que podemos almacenar en una variable, no sólo un valor primitivo, o la referencia de cualquier objeto, sino la referencia a una función ya definida. De esta forma, una variable puede tomar la forma de invocación a dicha función en el momento que se desee. Pongamos un ejemplo muy sencillo.

e5.1.kt

```
1 fun imprimeTuNombre (nombre : String){
2     println ("Tu nombre es $nombre")
3 }
4
5 fun main(){
6     val myFun = ::imprimeTuNombre           ①
7
8     myFun("Santiago Rodenas Herraiz")       ②
9     myFun("Sonia Mena Delgado")             ③
10 }
```

① myFun será una variable del tipo **(String) → Unit**. Lo que significa que recibirá un argumento de tipo entero y no devolverá nada. myFun, recibirá una referencia o dirección de memoria de la función imprimeTuNombre. Para ser algo brutos, es como si la variable *recibiera un código* y en el momento que quisiéramos y por mediación a dicha variable, invocáramos la ejecución de su código. Se sobreentiende que la variable myFun es declarada de la siguiente forma **myFun : (String) → Unit = ::imprimeTuNombre**

② Invocamos el código de la función almacenada por referencia, como si dicha varible fuera a hora la función.

③ Volvemos a invocar a dicha función, cambiando de valor el parámetro.

Invocar a una función, mediante una variable.

En el punto anterior, hemos visto que significa que una variable tome como valor la referencia de una función. También hemos visto como podemos invocar a dicha función a partir de dicha variable de tipo función. Lo que haremos ahora, es profundizar un poco más y ver como una variable de tipo función, puede tomar como valor la referencia de diferentes funciones que representen el mismo **tipo**, sino tendríamos que volver a decararla. Pongamos el ejemplo de una variable llamada operación, que almacene la referencia de una función, que recibe dos parámetros de tipo entero y devuelve el resultado de dicha operación.

e5.2.kt

```
1  fun suma (a: Int, b: Int) : Int {                                    ①
2      return a + b
3  }
4
5  fun resta (a: Int, b: Int) : Int {
6      return a - b
7  }
8
9  fun multi (a: Int, b: Int) : Int {
10     return a * b
11 }
12
13 fun main() {
14     var operacion : (Int, Int) -> Int                                ②
15     var res = 0
16
17     operacion = ::suma                                               ③
18     println("La operación suma de 2 y 3, es ${operacion(2,3)}")      ④
19     operacion = ::resta                                              ⑤
20     println("La operación resta de 2 y 3, es ${operacion(2,3)}")
21     operacion = ::multi
22     println("La operación multiplicacion de 2 y 3, es ${operacion(2,3)}")
23 }
```

① Más adelante, simplificaremos este tipo de funciones de la forma = **a**+**b**. Cuidado con las { }, cuando sólo es una sentencia ya que no es necesario.

② Nos declaramos una variable de tipo funcion. Esa función recibe dos argumentos de tipo Entero y devuelve también un entero.

③ Pasamos como valor la referencia de la funcion suma.

④ Invocamos a la función por medio de su variable.

⑤ Modificamos el valor de la variable operación con otra referencia distinta.

Concepto de expresión lambda

Si ya tenemos claro que existen variables que almacenan referencia a funciones, ya es hora de pasarnos a las expresiones lambda. **¿Qué son?** Podemos decir, que es una forma sencilla de definir una función con toda su lógica y de que una variable referencie dicha función, pero de un sólo paso y sin especificar el nombre de dicha función, es decir **(como si fuera ANÓNIMAAAAA)**. Pongamos el mismo ejemplo que en el punto anterior.

e5.3.kt

```
1 fun main(){
2     var operacion : (Int, Int) -> Int = {a, b -> a + b }                    ①
3     println("La operación suma de 2 y 3, es ${operacion(2,3)}")              ②
4     operacion = { a, b -> a - b }                                           ③
5     println("La operación resta de 2 y 3, es ${operacion(2,3)}")
6     operacion = { a, b -> a * b }
7     println("La operación multiplicacion de 2 y 3, es ${operacion(2,3)}")
8 }
```

① Nos declaramos nuestra variable de tipo función, pero al mismo tiempo y mediante una lambda, definimos el código de una función anónima a la que es referenciada.

② Invocamos a la función a partir de su variable.

③ Volvemos a cambiar la referencia de la variable, con otra lambda diferente.

Por tanto, una función lambda es una función anónima de la forma **{ parámetros → cuerpo }**, que es muy fácil, sencilla de utilizar cuando la función es pequeña o tiene poca lógica y por tanto se la podemos asignar a una variable de un sólo paso. No debemos de olvidar, que se necesitan parámetros y éstos están separados por comas. La siguiente cuestión es: **¿Podemos pasar como referencia a una función, otra función?**, o lo que es lo mismo, **¿Podemos pasar como parámetro a una función, la referencia de una función anónima** o la pregunta a la que os quería llevar, **¿Podemos pasar como parámetro a una función, una lambda?** La respuesta es que **Por supuesto....**

Ya no tenemos a obligación de definirnos todas las funciones que necesitemos y símplemente y en el momento que necesitemos una función, podemos crearnos una lambda.

En el siguiente punto sobre llamadas a funciones de orden superior, veremos cómo las lambdas pueden simplificar nuestro código. En lugar de declarar una función separada (función1) y luego pasar su referencia como parámetro a otra función (función2), podemos directamente pasar una lambda a función2, actuando como una función anónima. Dentro del cuerpo de función2, esta lambda será invocada, y su lógica se ejecutará en el contexto y momento específico en que se llame a función2. Es un avance que exploraremos más a fondo en el siguiente punto.

Llamadas a funciones de orden superior.

Este será quizás el punto más importante que le doy a este lenguaje de programación ya que define

claramente la diferencia entre una programación imperativa y otra funcional. Intentemos aclarar que significa llamada a funciones de orden superior, antes viendo qué sucede si pasamos como parámetro a una función otra función.

Imaginemos que tenemos ahora una función llamada **operación** y ésta acepta dos argumentos de tipo entero y una función que representa la lógica a efectuarse dentro de la función operación.

e5.4.kt

```
1 fun suma (a: Int, b: Int): Int =  a + b                                    ①
2 fun resta (a: Int, b: Int): Int =  a - b                                   ②
3 fun operacion (a: Int, b: Int, fn: (Int, Int)-> Int): Int =  fn (a,b)      ③
4
5 fun main(){
6     println("La operación suma de 2 y 3, es ${operacion(2,3, ::suma)}")    ④
7     println("La operación resta de 2 y 3, es ${operacion(2,3, ::resta)}")  ⑤
8 }
```

① Nos definimos la función suma.

② Nos definimos la función resta.

③ Nos definimos una función operación, que recibe tres parámetros. Los dos operandos y otro que será una referencia a una función del tipo **Int, Int → Int**. Dicha función, recibirá dos argumentos de tipo entero y devolverá como resultado un entero. La lógica de dicha función, es invocar a la pasada como argumento y sus parámetros serán los mismos que recibe la función operación.

④ Invocamos a la función operación y como tercer parámetro pasamos la referencia de sumar.

⑤ Invocamos a la función operación y como tercer parámetro pasamos la referencia de restar.

Ya es hora de explicar qué son las llamadas a orden superior y para ello ya tenemos los ingredientes necesarios que son: pasar como parámetro a una función, la referencia de otra función, pero anónima, es decir, una expresion lambda. Simplificaremos el código anterior sustituyendo cada una de las funciones que representan la lógica a efectuar por operación y en su lugar pondremos una lambda diferente (funcion anónima).

e5.5.kt

```
1 fun operacion (a: Int, b: Int, fn : ( Int, Int ) -> Int) = fn (a,b) //Aquí está lo
  bueno. Invocamos a la función pasada como argumento.
2
3 fun main(){
4     var res = operacion (2, 3) {                    ①
5         a, b -> a + b
6     }
7     println("La operación suma de 2 y 3, es $res")
8
9     res = operacion (2, 3 ) {                       ②
10        a, b -> a - b
11    }
12    println("La operación suma de 2 y 3, es $res")
```

```
13 }
```

① En vez de pasar la referencia de una función ya definida, pasamos una expresión lamda, que será invocada desde dentro de la misma operación, pero su tratamiento o ejecución de su lógica estará en el mismo lugar de donde se invoca a operación. Cuando el último parámetro de operación es una referencia a una función, podemos cerrar el paréntesis y a continuación poner la lambda. Otra forma de simplificarlo sería:

```
1 var res = operacion (2, 3, { a, b -> a + b })
```

Será algo particular, pero a mí me gusta más la primera forma ya que escribiremos bastante más código cuando tengamos que trabajar con Android Studio. <2> Volvemos a invocar a operación pero con otra lambda.

Por tanto, recordemos que si cogemos el cuerpo de la función anónima y la ponemos dentro del {}, sus variables corresponderán justamente, con los parámetros que utilizamos para invocar a la función pasada como referencia. Es el caso de los parámetros a,b en la invoación de **fn(a,b)**, que los hemos llamado (a, b) dentro de la lambda, en la llamada a la función **operación**.

Trabajando con Arrays y lambda.

Haremos un ejemplo de filtro para un array. Para ello, tenemos una función que recibe un array y una función lambda que definirá qué elementos son los que nos queremos quedar. Dicha función imprimirá todos los valores seleccionados.

e5.6.kt

```
1  import kotlin.random.Random
2
3  fun printValuesOfArr(arr : IntArray, fn : (Int) -> Boolean): Unit{
4      var newArr = IntArray(10) {-1}              ①
5      var j = 0
6      for (i in arr.indices){
7          if (fn (arr[i])){
8              newArr[j] = arr[i]                  ②
9              j++
10         }
11     }
12     for (i in newArr){                          ③
13         print("$i, ")
14     }
15 }
16
17 fun loadData():  IntArray{                       ④
18     var myArr = IntArray(10)
19     for (i in myArr.indices)
20         myArr[i] = Random.nextInt(0, 99)
21     return myArr
```

```
}


fun main(){
    var myArr = loadData()
    printValuesOfArr(myArr){true}                ⑤
    println()

    printValuesOfArr(myArr){ v-> v % 2 == 0 }    ⑥
    println()

    printValuesOfArr(myArr){
        if (it % 3 == 0 || it % 5 == 0) true     ⑦
        else false
    }
    println()

    printValuesOfArr(myArr){ it >= 50 }          ⑧
    println()

    printValuesOfArr(myArr){                     ⑨
        when (it){
            in 1..10 -> true
            in 20..30 -> true
            in 90..95 -> true
            else -> false
        }
    }
}
```

```
La función printValuesOfArr, recige tanto un array como una lambda. Dicha lambda tiene
como parámetro un valor y retornará true o false, dependiendo de la lógica que
utilizemos. Dicho valor será cada uno de los que tenga el array.
```

① Inicializaremos un array auxiliar con valor -1, del mismo tamaño que el recibido.

② Recorremos todos los valores del array y para cada uno de ellos, llamamos a nuestra funcion pasado por referencia. Si el resultado de la llamada es true, nos quedamos con el valor en caso contrario, lo desechamos.

③ Antes de que finalize la función, imprimimos los valores seleccionados.

④ Función que utilizamos para inicializar el array con valores aleatorios. Será el array que pasemos a nuestra función del punto 1.

⑤ Llamamos a la función, cuya lambda será true. Quiere decir que cada vez que se invoque desde printValuesOfArr a la función recibida, ésta devolverá siempre true.

⑥ Volvemos a llamar a la función, cuya lambda devolverá true, siempre que cada valor del array sea par. Recordemos que la lógica de la lambda se hace desde fuera de la función printValuesOfArr.

⑦ Volvemos a llamar a la función, cuya lambda devolverá true, siempre que el valor sea multiplo de 3 y de 5.

⑧ Aquellos que son mayores a 50.

⑨ Seleccionamos dependiendo de una selección múltiple.

Decimos que es una llamada de orden superior, porque desde printValuesOfArr, invocamos a la función recibida y la lógica es tratada fuera de la función printValuesOfArr.

Lo que haremos ahora, es simplificar el código ya que empezaremos a trabajar de una forma más concisa. El mismo ejemplo lo podemos ver de la siguiente forma.

e5.7.kt

```
1  import kotlin.random.Random
2
3  fun printValuesOfArr(arr: IntArray, fn: (Int) -> Boolean) {
4      val filteredArr = arr.filter(fn)              ①
5      println(filteredArr.joinToString(", "))       ②
6  }
7
8  fun loadData(): IntArray {
9      return IntArray(10) { Random.nextInt(0, 99) }
10 }
11
12 fun main() {
13     val myArr = loadData()
14
15     printValuesOfArr(myArr) { true }
16     printValuesOfArr(myArr) { it % 2 == 0 }
17     printValuesOfArr(myArr) { it % 3 == 0 || it % 5 == 0 }
18     printValuesOfArr(myArr) { it >= 50 }
19     printValuesOfArr(myArr) {
20         when (it) {
21             in 1..10, in 20..30, in 90..95 -> true
22             else -> false
23         }
24     }
25 }
```

① Lo que hacemos es filtrar por la lambda recibida. Para ello contamos con la función interna filter, ya vista.

② Para imprimir en pantalla, lo que hacemos es convertir todos los valores en un String separados por comas.

Veamos otro ejemplo muy parecido y posiblemente más sencillo. Ahora tendremos dos métodos que recibirán una función como parámetro, y en un principio haremos lo mismo, pero de forma diferente. En **myFun**, recorre todos los elementos del array y aplica la función (lambda) proporcionada a cada uno de ellos. La lógica específica de lo que queremos hacer con cada elemento (por ejemplo, contar cuántos son múltiplos de 3), se define dentro de la lambda y no

dentro de la función **myFun**. A diferencia de **myFun**, la función **myFun2** cuenta cuántos elementos del array cumplen la condición especificada en la lambda, devolviendo el total. La función que se pasa como lambda, contiene la lógica de filtrado, y myFun2 se encarga de contar cuántos elementos cumplen con esa lógica.

e5.8.kt

```
1  import kotlin.random.Random
2
3  fun myFun( arr : IntArray, fn: (Int) -> Unit){          ①
4      for (v in arr){
5          fn (v)
6      }
7  }
8
9  fun myFun2( arr: IntArray, fn: (Int) -> Boolean): Int{  ②
10     var cant = 0
11     for (v in arr){
12         if (fn(v))
13             cant ++
14     }
15     return cant
16 }
17
18 fun main(){
19     val myArray = IntArray(10) {Random.nextInt(0,100)}  ③
20     myArray.forEach {print("$it, ")}
21     println()
22
23     var cant = 0
24     myFun( myArray ){                                   ④
25         if (it %3 == 0)
26             cant ++
27     }
28     println ("La cantidad de valores del array multiplos de 3 es $cant")
29
30     println ("De otra forma------")
31     cant = 0
32     cant = myFun2( myArray ){ it %3 == 0}               ⑤
33     println ("La cantidad de valores del array multiplos de 3 es $cant")
34
35     println ("Ahora todos los componentes que suman mayor de 50")
36     var sum = 0
37     myFun (myArray ){                                   ⑥
38         if (it >= 50)
39             sum += it
40     }
41     println ("La suma de todos los mayores a 50 es $sum")
42 }
```

① Aplica la lambda a cada elemento, y la lógica (como contar o sumar) se define dentro de la lambda.

② Utiliza la lambda para filtrar elementos según una condición y devuelve el total de elementos que cumplen esa condición.

③ Cargamos el array con valores aleatorios y los mostramos en pantalla.

④ La lógica de contar múltiplos de 3, se implementa dentro de la lambda en myFun.

⑤ Llamada a la función myFun2. La lógica de filtrado (múltiplos de 3) se pasa como una lambda, y myFun2 devuelve la cantidad de elementos que cumplen esa condición.

⑥ Similar que en el punto 3, pero con otra condición.

Si tuviera que quedarme con una de las dos funciones puestas myFun y myFun2, sin duda lo haría con **myFun2**, ya que es una programación más declarativa porque sólo tenemos que pasarle una expresión booleana y desde fuera de la función el número de elementos es totalmente implícita.

Observar que fácil que sería hacer algo parecido sin utilizar ninguna función, sólo con la iteración del array proporcionado por **forEach**

e5.8.kt

```
1  import kotlin.random.Random
2
3  fun myFun3 ( arr: IntArray, fn: (Int) -> Boolean): Int {        ①
4      var cant = 0
5      arr.forEach{
6          if (fn( it ))
7              cant++
8      }
9      return cant
10 }
11
12 fun main(){
13     val myArray = IntArray(10) {Random.nextInt(0,100)}
14     var cant = 0
15     var sum = 0
16
17     myArray.forEach{
18         if (it % 3 == 0)
19             cant ++
20         if (it >= 50)
21             sum += it
22         }
23
24     println ("La cantidad de valores del array multiplos de 3, es $cant")
25     println ("La suma de todos los mayores o igual a 50, es $sum")
26
27     cant = myFun3 ( myArray) { it % 3 == 0}
28     println ( "El numero de multiplos de 3, es de $cant")
29
```

```
30 }
```

① Mirar lo sencillo que es volver a definir el mismo método de una manera más funcional.

```
1 fun myFun3(arr: IntArray, fn: (Int) -> Boolean): Int {
2     return arr.count(fn)
3 }
```

Es por esta razón, por la que kotlin es tan bueno. Un lenguaje funcional y muy expresivo. !!!!!

ACTIVIDADES

1. **Ejercicio de Referencia a una Función**
   - Define una función llamada `saluda` que reciba un `nombre` y muestre un saludo en pantalla.
   - Crea una variable `miSaludo` que almacene la referencia a la función `saluda`.
   - Usa `miSaludo` para invocar la función `saluda` con diferentes nombres.
2. **Ejercicio de Invocar Función mediante Variable**
   - Define funciones `multiplica`, `divide` y `resta` que realicen operaciones matemáticas básicas.
   - Crea una variable `operacion` que pueda almacenar la referencia a cualquiera de estas funciones.
   - Usa `operacion` para invocar las funciones `multiplica`, `divide` y `resta` con dos números enteros.
3. **Ejercicio de Expresión Lambda Simple**
   - Define una variable `sumaLambda` de tipo `(Int, Int) → Int` que use una expresión lambda para sumar dos enteros.
   - Imprime el resultado de invocar `sumaLambda` con diferentes pares de números.
4. **Ejercicio de Cambio de Lógica con Lambda**
   - Reemplaza la lógica de la variable `sumaLambda` por una lambda que realice una resta.
   - Imprime el resultado de la nueva lambda con diferentes pares de números.
5. **Ejercicio de Función de Orden Superior Básica**
   - Define una función `ejecutaOperacion` que reciba dos enteros y una función lambda para procesarlos.
   - Usa `ejecutaOperacion` para aplicar una lambda que realice la suma y otra que realice la multiplicación.
6. **Ejercicio de Uso de Lambda en Función de Orden Superior**
   - Crea una función `aplicaFiltro` que reciba un array de enteros y una lambda para filtrar los elementos.
   - Usa `aplicaFiltro` con diferentes lambdas para filtrar pares, impares y múltiplos de un número.
7. **Ejercicio de Función de Orden Superior con Parámetro Lambda**
   - Define una función `procesaArray` que reciba un array de enteros y una lambda para procesar cada elemento.
   - Imprime los resultados de procesar el array con lambdas que cuenten múltiplos de 3, sumen mayores a 50, etc.
8. **Ejercicio de Simplificación con Lambdas**
   - Simplifica la función `procesaArray` usando funciones estándar de Kotlin como `filter` y `map`.
   - Imprime los resultados de aplicar filtros y transformaciones al array de enteros.

9. **Ejercicio de Lambda Anónima en Función de Orden Superior**
    - Modifica una función `realizaOperacion` para aceptar lambdas anónimas que realicen diferentes cálculos.
    - Usa `realizaOperacion` con lambdas que sumen, resten y multipliquen.
10. **Ejercicio de Contar Elementos con Lambda**
    - Define una función `contarElementos` que cuente cuántos elementos de un array cumplen con una condición dada por una lambda.
    - Imprime el número de elementos que cumplen con la condición especificada en la lambda.
11. **Ejercicio de Sumar Elementos con Lambda**
    - Crea una función `sumarElementos` que sume los elementos de un array que cumplen con una condición dada por una lambda.
    - Imprime la suma de los elementos que cumplen con la condición especificada.
12. **Ejercicio de Filtrar y Contar con Lambdas**
    - Define una función `filtrarYContar` que reciba un array y una lambda para filtrar los elementos.
    - Usa `filtrarYContar` para contar los elementos que cumplen con diferentes condiciones de filtrado.
13. **Ejercicio de Uso de Lambda para Imprimir Valores**
    - Crea una función `imprimeValores` que reciba un array y una lambda para determinar qué valores imprimir.
    - Usa `imprimeValores` para imprimir todos los valores del array, los valores pares, y los valores mayores a 50.
14. **Ejercicio de Transformación de Datos con Lambda**
    - Define una función `transformarDatos` que reciba un array y una lambda para transformar cada elemento.
    - Imprime el resultado de transformar el array con diferentes lambdas.
15. **Ejercicio de Composición de Funciones**
    - Crea una función `componerFunciones` que reciba dos funciones y las aplique en secuencia a un valor.
    - Usa `componerFunciones` para aplicar diferentes combinaciones de funciones a un valor inicial.

CLASES

En este punto, no profundizaremos y simplemente nos decantaremos en ver aspectos comparativos de como se declara una clase, las diferentes formas que hay de definir atributos, constructores, sobreescribir métodos, el init de kotlin como alternativa a inicializar los atributos, visibilidad de los atributos, repaso de la extensión de las clases (herencia), las clases abstractas e interfaces. Lo veremos todo muy por encima por medio de algunos ejemplos.

Estructuraremos en :

1. Declaración de las clases.
2. Constructores e inicialización de atributos.
3. Métodos y atributos privados.
4. Mezclando algunas lambda.
5. Relaciones entre clases.
6. Herencia
7. Clases abstractas
8. Interfaces

Declaración de clases.

La declaración de una clase en kotlin, es algo similar que en java, salvo que podemos definirnos los atributos de una forma más sencilla. Recordar la diferencia entre declararnos los atributos como var o val.

El siguiente ejemplo, muestra la declaración de una clase Persona, con dos atributos nombre y edad. **Recordemos que cuando son atributos primitivos, no es necesario su inicialización**. Sobreescribimos el método toString con la palabra **override** y nos creamos un método que nos devuelve true/false dependiendo de si es mayor de edad.

Para darle más sentido al ejemplo, iteraremos un array de Personas con un forEach. Utilizaremos una lambda que evalúe si la persona es adulta e incrementando una variable. Después definimos una función que reciba como parámetro otra función y cuya lambda en su llamada, imprime en pantalla aquella Persona, cuyo nombre tenga más de 5 letras.

e6.1.kt

```
1 class Persona ( var name : String, var age : Int ){                ①
2     override fun toString() = ("Nombre: $name y su edad: $age")    ②
3     fun isAdult() = (age >= 18)
4
5 }
6
7 fun myFun (persons: Array<Persona>, fn : (String) -> Unit ) {
8     persons.forEach{
9         fn (it.name)
10    }
```

```
11 }
12
13 fun main(){
14     val persons : Array<Persona> = arrayOf(                          ③
15         Persona("Santi", 46),
16         Persona("Sonia", 45),
17         Persona("Guille", 14),
18         Persona("Diego", 11)
19     )
20
21     var cant = 0
22     persons.forEach{                                                 ④
23         println(it)
24         if (it.isAdult())
25             cant++
26     }
27     println("La cantidad de personas adultas es $cant")
28     myFun ( persons ){                                               ⑤
29        if (it.count() > 5)
30             println ("$it tiene mas de 5 letras")
31     }
32 }
```

① Dentro del mismo (), definimos los atributos que queramos. Cuidado con no poner val o var junto al nombre, porque sino no serán atributos. En el caso de **val**, será una constante y por tanto no podremos modificar su contenido.

② Sobreescribimos el toString con la palabra reservada **override**.

③ Cargamos un array de 4 Personas.

④ Lambda del forEach.

⑤ Lambda de nuestra función.

En el siguiente ejemplo, mostraremos la forma de declararnos los atributos de manera implícita a como lo hace java. También veremos como podemos crearnos los famosos setters/getters, que en kotlin es diferente. Cuando hagamos referencia a un atributo de la forma **objeto.atributo**, inmediatamente se referenciará al getter y de una forma análoga, cuando sobreescribamos un atributo de la forma **objeto.atributo** = **valor**, se referenciará al setter de manera explícita. En kotlin, cuando definimos un atributo, debemos incluir el método getter/setter de la forma que en el ejemplo os mostraré.

e6.2.kt

```
1 class Alumno (){                                                    ①
2
3     var dni : String                                                ②
4     var name : String = ""                                          ③
5         set (value){
6             if (value.isNotEmpty())
7                 field = value.uppercase()
8
```

```
9         }
10
11      var age: Int = 0
12      var phone : String = ""
13
14      constructor(dni: String, name : String, age: Int, phone: String) : this() { ④
15          this.dni = dni
16          this.name = name
17          this.age = age
18          this.phone = phone
19          println ("finaliza el constructor secundario")
20      }
21
22      init{                                                                    ⑤
23          println ("Extensión contructor por defecto. No responde a una extesión de
    constructor secundario")
24          dni = ""
25
26      }
27
28      override fun toString(): String = "Dni: ${name}, nombre: ${name}, edad: ${age},
    telefono: ${phone}"
29
30  }
31
32  fun main(){
33      val myAlum = Alumno()
34      val myAlum2 = Alumno("1234","Santi",46, "953 45 76 56")
35      myAlum.name = "Sonia"
36      println (myAlum2)
37      println (myAlum)
38  }
```

① Como objeto que es un String, hay que inicializarlo obligatoriamente. Mas adelante hablaremos de los nulables.

② Si no inicializamos el atributo de manera implícita, tendremos que hacerlo en el bloque init.

③ El atributo name, tiene un setter con una lógica interna. En el caso de que tenga un valor diferente a "", se convertirá automaticamente en mayúsculas. Siempre que hagamos algo como p.name = "santi", se llamará explícitamente al método setter. Si quisiéramos hacer algo parecido con el getter, tendríamos que definirlo como get. Dentro del método getter/setter, referenciamos a su valor con la palabra reservada **value** y para sobreescribirlo, podemos hacerlo mediante el nombre del atributo o de manera genérica con **field**. Hay que darse cuenta, que cuando existe un setter, se debe inicializar explícitamente, no cabe hacerlo en el init como sucede en el punto 2.

④ El constructor primario es llamando a Alumno(), pero podemos definirnos tantos constructores como queramos, por tanto con la palabra reservada **constructor**, inicializamos tantos atributos como queramos pero necesitamos extender del constructor primario, por eso necesitamos : this(). El orden es sencillo de intuir, primero se llamará al por defecto () y después al secundario

definido en constructor.

⑤ El bloque init, será llamado en primera instancia. Lo podemos utilizar para inicializar o realizar diferentes comprobaciones en la creación del objeto.

La ejecución:

```
Extensión contructor por defecto. No responde a una extesión de constructor secundario
Extensión contructor por defecto. No responde a una extesión de constructor secundario
finaliza el constructor secundario
Dni: SANTI, nombre: SANTI, edad: 46, telefono: 953 45 76 56
```

Constructores e inicialización de atributos.

En kotkin al igual que en Java, podemos tener tantos constructores como queramos. Definiremos nuestro constructor por defecto dentro de la declaración de la clase **()** y los constructores que deseemos serán extensiones del por defecto llamando al mismo this(). Por otra parte, hemos visto como el método init() es el primero que se llamará después de invocar al constructor por defecto. También hemos visto los setters y los getters como métodos que son llamados implícitamente después de sobreescribir o invocar a cualquier atributo de nuestro objeto. El siguiente ejemplo, lo estudiaremos para observar qué sucede cuando queremos realizar ciertas comprobaciones sobre atributos, por ejemplo que el nombre no sea nulo o forzar a que la edad sea siempre 0 cuando sea menor de edad. Examinemos el ejemplo y la impresión en pantalla:

e6.3.kt

```
1  public class PersonaGetSet(){
2      var name : String? = null
3
4          set(value){
5              if (value != null) {
6                  if (value.isEmpty()) {
7                      println("El valor debe contener texto")
8                  }
9                  else {
10                     field = value
11                 }
12             }
13             else {
14                 println("Has pasado un null al nombre y no esta permitido")
15
16             }
17         }
18
19         get(){
20             field?.let{
21                 return it
22             }?: run{
23                 return "<Sin nombre>"
```

```
            }
        }


    var age : Int = 0

        set(value){
            if (value >= 18)
                field = value
            else{
                field = 0
                println ("Fuerzo por mas narizes a poner a 0 la edad para aquellos menores de edad")
            }

        }
        get()=field



    constructor(_name: String?, _age: Int) : this() {
        println ("!!!!!!!!!!!Esto dentro del constructor secundario")
        name = _name
        age = _age
    }

   override fun  toString() = "Nombre: $name y edad: $age"

}

fun main(){
    println ("Creo pesona nula y después asigno nombre santi")
    val per = PersonaGetSet()
    per.name = "santi"
    println (per)
    per.name = ""
    println ("-------")
    println ("Creo persona con nombre null y edad 15")
    val per1 = PersonaGetSet(null, 15)
    println (per1)
    println ("-------")
    val per2 = PersonaGetSet("santi", 5)
    println (per2)
}
```

La ejecución:

```
Creo pesona nula y después asigno nombre santi
Nombre: santi y edad: 0
El valor debe contener texto
```

```
-------
Creo persona con nombre null y edad 15
!!!!!!!!!!Esto dentro del constructor secundario
Has pasado un null al nombre y no esta permitido
Fuerzo por mas narizes a poner a 0 la edad para aquellos menores de edad
Nombre: <Sin nombre> y edad: 0
-------
!!!!!!!!!!Esto dentro del constructor secundario
Fuerzo por mas narizes a poner a 0 la edad para aquellos menores de edad
Nombre: santi y edad: 0
```

Cuando queramos que el constructor por defecto sea explícito, debemos de no poner **()** en la definición de la clase y si la palabra reservada **constructor**. Pondré un ejemplo muy sencillo:

```
1 class A{
2     var a: Int
3     var b: String = ""
4
5     constructor (_a : Int, _b : String){
6         a = _a
7         b = _b
8     }
9 }
10
11 fun main(){
12     // val a = A()              ①
13     val b = A(2, "santi")       ②
14     println ("atributo a: " + b.a + ", atributo b: " + b.b)
15 }
```

① Ya no podemos utilizar el constructor por defecto (), porque en la declaración de la clase, no hemos puesto los **()**.

② El constructor por defecto que tenemos ahora, es el que tiene la palabra reservada **constructor**.

Métodos y atributos privados.

Por defecto, en kotlin todos los métodos son públicos, pero podemos poner perfectamente el modificador de visibilidad public explícitamente. Pondré un ejemplo muy sencillo de un atributo y métodos privados. Para ello utililizaré el ejemplo anterior. La idea es que añadiremos un atributo privado lower, que por defecto será siempre true. Dos métodos privados que intercambien de minúscula a mayúscula el nombre y viceversa. Para poder llamar al método que intercambie el estado del nombre, crearemos un nuevo método público, para que dependiendo del estado del nombre (minúscula o mayúscula), se intercambie. Veamos que fácil es:

e6.4.kt

```
1 //..... atributos y métodos añadidos con respecto al ejemplo anterior .e6.3.kt
2 private var lower : Boolean = true
```

```
3
4
5       private fun changeLowerToUpper() : Unit {
6           name = name?.uppercase()
7           lower = false
8       }
9
10      private fun changeUpperToLowwer() : Unit {          ①
11          name = name?.lowercase()
12          lower = true
13      }
14
15      public fun changeState() = lower.apply {            ②
16          if (this)
17              changeLowerToUpper()
18          else
19              changeUpperToLowwer()
20      }
21
22      public fun stateName() {                            ③
23          if (lower)
24              println("Está en minúscula")
25          else
26              println("Esta en mayúscula")
27
28      }
29  }
30
31  fun main(){
32     val per = PersonaGetSet("sAnTi", 40)
33     per.stateName()
34     println (per)
35     println ("------")
36     per.changeState()
37     per.stateName()
38     println (per)
39
40  }
```

① Unit puede ser perfectamente implícita.

② Lamda aplicada a un booleano. Apply debe utilizarse para aplicar alguna lógica más compleja. Lo he puesto sólo como ejemplo.

③ Verifica el estado del booleano.

La ejecución:

```
Está en minúscula
Nombre: santi y edad: 40
------
```

```
Esta en mayúscula
Nombre: santi y edad: 40
```

Mezclando algunas lambda.

En multitud de ocasiones, deberemos trabajar con clases y lambdas. El siguiente ejemplo, será una clase con un atributo de tipo Array, al que cargaremos datos de manera aleatoria y algunos métodos sencillos que requieran trabajar con las expresiones lambda. Recordamos que existen algunos métodos como **count**, **all**, **any** en el que reciben una expresión y en el caso de count, devolverá una cantidad de elementos del array, que cumpla con la expresión que pasemos. Con all y any, evaluarán si todos los elementos o alguno de ellos cumple con una condición.

e6.5.kt

```
1  import kotlin.random.Random
2
3  class MyArray {
4      var arr = IntArray(10)
5
6      fun loadValues(){
7          for (i in arr.indices){
8              arr[i] = Random.nextInt(0,11)
9          }
10     }
11
12     fun loadValues2(){
13         arr = IntArray(10){Random.nextInt(0,11)}                    ①
14     }
15
16     fun printArray(){
17         for (i in arr)
18             print("$i, ")
19     }
20
21     fun printElementTo5(){
22         val cant = arr.count{ it <= 5 }                              ②
23         println ("La cantidad de elementos menor que 5 es $cant")
24     }
25
26     fun printAllTo9(){
27         val less = arr.count{ it <= 9}
28         if (less == arr.count())
29             println("Todos son menor o igual que 9")
30         else
31             println("Hay números mayores que 9")
32     }
33
34     fun printAllTo9_2(){
35         if (arr.all { it <= 9})                                      ③
```

```
36              println("Todos son menor o igual que 9")
37          else
38              println("Hay números mayores que 9")
39      }
40
41      fun printElementoTo10(){
42          if (arr.any { it == 10})                                    ④
43              println ("Hay un elemento que tiene al menos un 10")
44          else
45              println ("No hay ningun elemento que sea 10")
46      }
47  }
48
49  fun main(){
50      var arr = MyArray()
51      arr.loadValues2()
52      arr.printArray()
53      println("--------")
54      arr.printElementTo5()
55      arr.printAllTo9()
56      arr.printAllTo9_2()
57      arr.printElementoTo10()
58  }
```

① Otra forma de cargar los valores del array.

② Expresión lambda pasada a la función count. Cuenta el numéro de elementos.

③ Expresión lambda pasada a la función all. Si todos los elementos cumplen con la condición.

④ Expresión lambda pasada a la funcíon any. Si algún elemento cumple con la condición.

Relaciones entre clases.

Lo mas normal en cualquier ejemplo de POO, son las relaciones entre diferentes clases. Pondremos un ejemplo muy sencillo de los datos personales de un empleado.

e6.6.kt

```
1  class PersonalData(val name: String?, val phone: String?) {              ①
2      override fun toString(): String {
3          return "PersonalData(name=$name, phone=$phone)"
4      }
5  }
6
7  class Employee(val nroEmp: Int, val personalData: PersonalData?) {       ②
8      override fun toString(): String {
9          return "Employee(nroEmp=$nroEmp, personalData=$personalData)"
10     }
11 }
12
13 fun main(){
```

```
14      val enterprise : Array<Employee> = arrayOf(                          ③
15          Employee (1, null),
16          Employee (2, PersonalData (null, null)),
17          Employee (3, PersonalData ("santi", null)),
18          Employee (4, PersonalData ("sonia", "953 12 34 56"))
19      )
20
21      for (emp in enterprise){
22          if (emp.personalData != null)
23              println (emp.personalData.name)
24      }
25      println("----")
26
27      enterprise.forEach{ emp ->                                           ④
28          if (emp.personalData != null){
29              if (emp.personalData.name != null)
30                  print ("${emp.personalData.name} y ${emp.personalData.name.count()}
    caracteres")
31              else
32                  print("No tiene nombre")
33
34              if (emp.personalData.phone != null)
35                  println (". Su teléfono es ${emp.personalData.phone}")
36              else
37                  println (". No tiene teléfono")
38          }
39      }
40
41  }
```

① Nuestra clase, al disponer de atributos de tipo String, tenemos que controlar que no sean nulos. Recordamos que un objeto nullable, hay que declararlo con ?. Es una de las grandes diferencias con respecto a Java. Este tipo de clases, las crearemos como dataclass. **Nos ahoraríamos sobreescribir el toString**.

② La clase empleado, dispone de un atributo identificativo y de los datos personales. En este ejemplo, se permite crear un empleado sin datos personales, por tanto el atributo de tipo PersonalData, debe ser nullable.

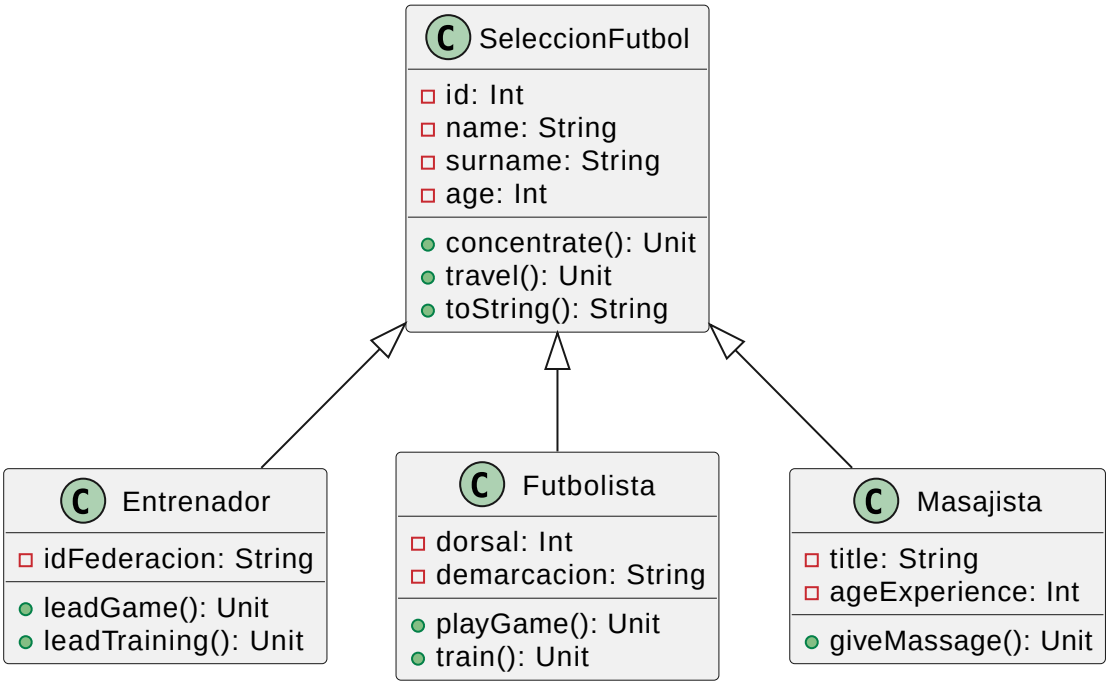
③ Cuando declaramos un array de objetos, podemos hacerlo incluso de manera implícita sin especificar el tipo. **var emp = arrayOf(Employee (4, PersonalData ("sonia", "953 12 34 56")))** De todas formas, recomiendo hacerlo: **var emp : Array <Employee> ...**

④ Recordemos otra de las formas de recorrer un array de objetos. Mas adelante, utilizaremos la sentencia **let** en vez de comprobar con un **if (objeto!= null)**.

Herencia

La herencia es exactamente igual que en java, salvo en la declaración que no hay que poner la palabra reservada extend, sino lo **:<Clase Padre>** y también para que una clase padre pueda tener descendencia, a la superclase debemos poner la palabra reservada **open**, de lo contrario no podrá

extenderse la clase padre. Este ejemplo, lo he sacado de una web Ejemplo en java



Consideraciones:

1. Aquellos atributos que deseemos heredar, tenemos que incluir la palabra reservada protected (igual que en java).
2. Como constructor por defecto, recibe el id, nombre y apellidos.
3. Sobreescribimos el toString del padre.
4. El padre tiene dos métodos que serán heredados a sus hijos. Son concentrarse y viajar.
5. No pondremos ningún atributo/método privado.

e6.7.kt (clase Padre)

```
open class SeleccionFutbol  {
    protected var id : Int = 0

    protected var name : String?= null
        set(value) {
            if (value != null){
                if (value.isNotEmpty())
                    field= value.uppercase()
            }else{
                println("El nombre no puede ser nulo")
                field = ""
            }
        }

    protected var surname : String?= null
        set(value) {
            if (value!= null){
                if (value.isNotEmpty())
```

```
                    field = value.uppercase()

            }else{
                println("El apellido no puede ser nulo")
                field = ""
            }
        }

    protected var age : Int = 0

    constructor(id : Int, name: String, surname: String, age : Int) {
        this.id = id
        this.name = name
        this.surname = surname
        this.age = age
    }

    fun concentrate(){
        println("El integrante se está concentrando")
    }

    fun travel(){
        println("El integrante está viajando")
    }

    override fun toString() = "integrante con id= $id, nombre es $name, $surname,
con edad $age"
}

fun main(){
①
    val s = SeleccionFutbol(1, "Santi", "Rodenas Herráiz", 40)
    println (s)
    s.concentrate()
    s.travel()
}
```

① Probamos nuestra clase.

Ahora pondremos las subclases. Cada una de ellas, tendrá acceso a los métodos y atributos protected de SeleccionFutbol. De momento, no es necesario que vuelvan a redefinir dichos métodos.

e6.7.kt (clase Entrenador)

```
class Entrenador(id: Int, name: String, surname: String, age: Int, idFede: String)
: SeleccionFutbol(id, name, surname, age) {  ①

    var idFederacion: String = idFede

    init {
```

```
6
7           idFederacion += "-ES" // Añade el sufijo "-ES" al idFederacion
  ②
8       }
9
10      fun leadGame() {
  ③
11          println("El entrenador con nombre $name está dirigiendo un partido de
  futbol")
12      }
13
14      fun leadTraining() {
15          println("El entrenador con nombre $name está dirigiendo un entrenamiento")
16      }
17
18       override fun toString(): String {
  ④
19          return "${super.toString()}, idFederacion=$idFederacion"
20      }
21  }
22
23
24  fun main(){
25      val santi = Entrenador(1, "santi", "rodenas herráiz", 40, "12345")
26      println (santi)
27  }
```

① Al igual que en Java, debemos llamar al constructor por defecto de la superclase. Se puede aceptar la palabra **super()** en vez del nombre de la superclase.

② Ejemplo típico del método init. Añadimos una cadena.

③ Son métodos específicos de un entrenador. Dirigir un partido y entrenar un partido.

④ Invocamos al toString del padre.

e6.7.kt (clase Futbolista)

```
1  class Futbolista : SeleccionFutbol {
2
3      constructor(id : Int, name: String, surname: String, age : Int, dorsal : Int,
  demarcacion : String?) :
4              super(id, name, surname, age) {
5          this.dorsal = dorsal
6          this.demarcacion = demarcacion
7      }
8
9      var dorsal : Int = 0
10
11     var demarcacion : String? = null
12
13         set(value) {
```

```
            if (value == null)
                println("Este jugador no tiene ninguna demarcación, hay que ponerle una.")             ①
            else
                field = value
        }

        get() = field ?: "Sin demarcacion..."
    ②


    fun playGame() : Unit{
        println("El jugador con nombre $name, $surname, está jugando un partido de futbol")
    }

    fun train() : Unit{
        println("El jugador con nombre $name, $surname, está entrenando")
    }
}

fun main(){
    val santi = Entrenador(1, "santi", "rodenas herráiz", 40, "12345")
    val sonia = Futbolista(2, "sonia", "mena delgado", 35, 4, "Delantera")
    println (santi)
    println (sonia)
}
```

① Lo razonable, es no dejar un atributo a null.

② Utilizamos el operador **elvis**, como alternativa.

e6.7.kt (clase Masajista)

```
class Masajista(id: Int, name: String, surname: String, age: Int, var title: String, ageExperience: Int?) : SeleccionFutbol(id, name, surname, age) {

    var ageExperience: Int? = ageExperience
    ①

        set(value) {
            if (value == null) {
                println("El masajista debe tener un mínimo de 5 años de experiencia.")
            } else if (value >= 5) {
                field = value
            } else {
                println("Este masajista no tiene años de experiencia suficiente.")
            }
        }

```

```
15          get() = field ?: throw IllegalStateException("El masajista debe tener al
    menos 5 años de experiencia.")                    ②
16
17
18      fun giveMassage() {
19          println("El masajista con nombre $name, está dando un masaje")
20      }
21
22      override fun toString(): String {
23          return "${super.toString()}, título=$title, años de
    experiencia=$ageExperience"
24      }
25  }
26
27  fun main() {
28      try {
    ③
29          val masajista = Masajista(1, "Guillermo", "Rodenas", 20, "Fisioterapeuta",
    null)
30          println(masajista.ageExperience)
31      } catch (e: IllegalStateException) {
32          println("Se produjo una excepción: ${e.message}")
33      }
34  }
```

① Los años de experiencia, pueden ser null. Lo hacemos así, para que se pueda generar una excepción. Otro ejemplo más.

② Propagamos una excepción, en el caso de los años de experiencia sea nulo.

③ El bloque try/catch es igual que en java. IllegalStateException es una superclase de RuntimeException. Representa la llamada a un método en un momento en el que no se debería.

Al final, desde cualquier subclase podemos invocar a los métodos de la clase padre sin problema alguno.

e6.7.kt (main)

```
1  fun main() {
2      val entrenador = Entrenador(1, "santi", "Rodenas", 53, "12345")
3      val futbolista = Futbolista(2, "Sonia", "Mena", 36, 10, "Delantero")
4      val masajista = Masajista(3, "Guillermo", "Rodenas", 25, "Fisioterapeuta", 6)
5
6      println("Entrenador:")
7      println(entrenador.toString())
8      entrenador.concentrate()
9      entrenador.travel()
10     entrenador.leadGame()
11     entrenador.leadTraining()
12     println()
13
14     println("Futbolista:")
```

```
15      println(futbolista.toString())
16      futbolista.concentrate()
17      futbolista.travel()
18      futbolista.playGame()
19      futbolista.train()
20      println()
21
22      println("Masajista:")
23      println(masajista.toString())
24      masajista.concentrate()
25      masajista.travel()
26      masajista.giveMassage()
27  }
```

La ejecución:

```
Entrenador:
integrante con id= 1, nombre es SANTI, RODENAS, con edad 53, idFederacion=12345-ES
El integrante se está concentrando
El integrante está viajando
El entrenador con nombre SANTI está dirigiendo un partido de futbol
El entrenador con nombre SANTI está dirigiendo un entrenamiento

Futbolista:
integrante con id= 2, nombre es SONIA, MENA, con edad 36
El integrante se está concentrando
El integrante está viajando
El jugador con nombre SONIA, MENA, está jugando un partido de futbol
El jugador con nombre SONIA, MENA, está entrenando

Masajista:
integrante con id= 3, nombre es GUILLERMO, RODENAS, con edad 25,
título=Fisioterapeuta, años de experiencia=6
El integrante se está concentrando
El integrante está viajando
El masajista con nombre GUILLERMO, está dando un masaje
```

Clases Abstractas

La idea de las clases abstractas es la misma que la que podemos encontrarnos con java. Aquella clase que declare un método como **abstract**

Antes de adaptar nuestro ejemplo con clases abstractas, analizemos lo siguiente: Una clase padre que tenga sus métodos, serán heredados por las clases hijas, por tanto la lógica es del padre. Al mismo tiempo, las subclases pueden tener sus propios métodos y por ahora no vemos ningún problema, pero imaginemos que todas las subclases tuvieran un mismo método llamada entrenar y cada una tubiera que implementar su propia forma de entrenar, es decir un futbolista no puede entrenar igual que un entrenador o masajista. Imaginemos también, que la clase padre no tubiera el método entrenar. Por supuesto, suponemos que cada subclase implementa el método training(). Contemos también con una lista de objetos de tipo SeleccionFutbol, para que podamos tener diferentes objetos de distintas subclases. ¿Qué sucederá si al recorrer cada instancia, queremos llamar al método entrenar? Lo que sucederá es que el método train no forma parte de la clase padre y por tanto habría que castear cada instancia a la subclase para que se pudiera llamar. Como ejemplo, examinar el siguiente código.

e6.8.kt

```
fun main() {
    val entrenador = Entrenador(1, "santi", "Rodenas", 53, "12345")
    val futbolista = Futbolista(2, "Sonia", "Mena", 36, 10, "Delantero")
    val masajista = Masajista(3, "Guillermo", "Rodenas", 25, "Fisioterapeuta", 6)

    var listIntegrantes : MutableList<SeleccionFutbol> = mutableListOf(
            entrenador, futbolista, masajista)

    for (integrante in listIntegrantes){
        println (integrante)
        integrante.concentrate()                ①
        integrante.train()                      ②
    }
}
```

① Podemos perfectamente llamar al método concentrate del padre.

② Tendríamos un error de compilación porque train no está definido en la clase SeleccionFutbol y deberíamos hacer un casteo.

```
when (integrante) {
    is Entrenador -> integrante.train()    ①
    is Futbolista -> integrante.train()
    is Masajista -> integrante.train()
}
```

① Realizamos los casteos oportunos.

Es ahora cuando queremos introducir el concepto de clase astracta y método abstracto. Lo que

haríamos es añadir a nuestra clase padre SeleccionFutol un método abstracto **train** sin código alguno y las subclases al heredar de una clase abstracta, están obligados a implementar su código. De esa forma, ya no es necesario realizar ningún casteo, porque aseguramos que todas las subclases implementen el método training(). Veamos la modificación.

```
1 //....
2     abstract fun training()                                          ①
3 //.....
4
5
6 //.....
7     override fun training() {
8         println("El entrenador con nombre $name está entrenando")    ②
9     }
10 //.....
11
12
13 //.....
14    override fun training() {
15        println("El jugador con nombre $name está entrenando")       ③
16    }
17 //.....
18
19
20 //.....
21    override fun training() {
22        println("El masajista con nombre $name está entrenando")     ④
23    }
24 //.....
25
26 fun main(){
27    val entrenador = Entrenador(1, "santi", "Rodenas", 53, "12345")
28    val futbolista = Futbolista(2, "Sonia", "Mena", 36, 10, "Delantero")
29    val masajista = Masajista(3, "Guillermo", "Rodenas", 25, "Fisioterapeuta", 6)
30
31    var listIntegrantes : MutableList<SeleccionFutbol> = mutableListOf(
32            entrenador, futbolista, masajista)
33
34     for (integrante in listIntegrantes){
35            println(integrante.training())                           ⑤
36            println(integrante)
37    }
38 }
```

① Definimos el método abstracto en SeleccionFutbol, por tanto se convierten en una clase abstracta.

② Sobreescribimos el método training en Entrenador.

③ Sobreescribimos el método training en Futbolista.

④ Sobreescribimos el método training en Masajista.

⑤ Ya no tenemos que comprobar la instancia, porque todos están obligados a sobreescribir training.

Perfecto, ya tenemos claro en qué consiste el polimorfismo a partir de las clases abstractas.

Interfaces

El polimorfismo se implementa de manera efectiva a través de las interfaces. Hasta hace poco, las interfaces solo definían los métodos que las clases que las implementaban debían sobrescribir. Sin embargo, a partir de Java 8 y de manera nativa en Kotlin, las interfaces también pueden incluir la implementación de algunos métodos.

En el desarrollo de aplicaciones Android, las interfaces son especialmente útiles, ya que nos permiten definir contratos entre diferentes partes del código, sin necesidad de conocer de antemano la implementación exacta que se utilizará. Por ejemplo, al desarrollar una aplicación Android, podemos necesitar un acceso a datos que puede variar según el caso: podría ser a una base de datos (como Room), a una API usando Retrofit, o incluso a una estructura de memoria interna con una lista de objetos.

Como no sabemos qué método de acceso a datos utilizaremos en la fase inicial del desarrollo, podemos separar responsabilidades usando una interfaz. Definimos una interfaz con los métodos necesarios (por ejemplo, para cargar, guardar o eliminar datos), y más adelante, creamos distintas clases que implementen esta interfaz según sea necesario: una para Room, otra para Retrofit, y así sucesivamente. De esta forma, las partes de nuestra aplicación que dependen del acceso a datos no se ven afectadas por el cambio de implementación, ya que todas siguen el mismo contrato definido por la interfaz.

Vayamos a nuestro ejemplo de SeleccionFutbol. Podemos definir una interfaz con los métodos training(), travel(), concentrarse(). Las subclases implementarán de la interfaz como si fuera el ejemplo anterior.

e6.9.kt

```
1  interface IntegranteSeleccionFutbol {
2      var anio : Int                                                    ①
3
4      fun training()                                                    ②
5
6      fun travel()
7
8      fun concentrarse(){ //puedo definir un método de interfaz         ③
9          println("Estamos concentrados desde la interfaz")
10     }
11
12 }
```

① Dentro de la interfaz, podemos tener variables o constantes.

② Tanto training, como travel son métodos que las subclases deberán implementar.

③ Podemos tener incluso un método concentrarse con código que

ACTIVIDADES

1. **Ejercicio de declaración de clase**: Declara una clase `Animal` con dos atributos: `nombre` (de tipo `String`) y `edad` (de tipo `Int`). Sobreescribe el método `toString` para que devuelva una cadena con el nombre y la edad del animal. Crea un objeto de la clase `Animal` y muestra su información en la consola.
2. **Ejercicio de constructor secundario**: Crea una clase `Vehiculo` con los atributos `marca` y `modelo`, ambos de tipo `String`. Define un constructor secundario que permita inicializar estos atributos al momento de crear un objeto. Incluye un método que imprima los detalles del vehículo.
3. **Ejercicio de método de validación**: Define una clase `Estudiante` que tenga los atributos `nombre`, `edad`, y `nota`. Implementa un método `isAprobado()` que devuelva `true` si la nota es mayor o igual a 60, y `false` en caso contrario. Crea un objeto de `Estudiante` y muestra si está aprobado o no.
4. **Ejercicio de setter modificado**: Implementa una clase `Libro` con atributos `titulo` y `autor`, ambos de tipo `String`. Usa un setter para el atributo `titulo` que convierta el valor a mayúsculas antes de asignarlo. Crea una instancia de `Libro`, cambia el título y muestra el libro en la consola.
5. **Ejercicio de método privado**: Declara una clase `CuentaBancaria` con un atributo `saldo` de tipo `Double`. Define un método privado `actualizarSaldo` que ajuste el saldo según el monto depositado o retirado. Implementa un método público `realizarTransaccion` que llame a `actualizarSaldo` y muestre el saldo actualizado.
6. **Ejercicio de descuento en producto**: Crea una clase `Producto` con atributos `nombre` y `precio`, ambos de tipo `String` y `Double` respectivamente. Implementa un método `aplicarDescuento` que reduzca el precio en un porcentaje dado. Muestra el precio del producto antes y después de aplicar el descuento.
7. **Ejercicio de cálculo de distancia**: Diseña una clase `Punto` con atributos `x` y `y` de tipo `Int`. Implementa un método que calcule la distancia entre dos puntos. Crea dos objetos `Punto` y muestra la distancia entre ellos en la consola.
8. **Ejercicio de impresión de datos de empleado**: Define una clase `Empleado` con atributos `id` (de tipo `Int`) y `datosPersonales` (de tipo `PersonalData`). Implementa un método que imprima el nombre y el teléfono del empleado si `datosPersonales` no es `null`.
9. **Ejercicio de suma de diagonales**: Implementa una clase `Matriz` que contenga un atributo `data` de tipo `Array<Array<Int>>`. Añade un método `sumaDiagonales` que calcule la suma de los elementos en las diagonales de la matriz. Crea una matriz y muestra la suma de sus diagonales.
10. **Ejercicio de cálculo de área de rectángulo**: Crea una clase `Rectangulo` con atributos `base` y `altura` de tipo `Double`. Implementa un método `calcularArea` que retorne el área del rectángulo. Define un objeto `Rectangulo`, calcula su área y muéstrala en la consola.
11. **Ejercicio de Herencia en Animales**. En este ejercicio, se muestra la herencia en Kotlin usando una jerarquía de clases relacionadas con animales. La clase base Animal tiene atributos protegidos como id, nombre y edad, y métodos comer y dormir. Las clases derivadas Perro y Gato extienden Animal y añaden sus propias características: Perro tiene un atributo raza y un método ladrar, mientras que Gato tiene un atributo color y un método maullar. La clase Animal debe ser abierta (open) para permitir la herencia, y los métodos y atributos que se quieren heredar deben ser protected.
12. **Ejercicio de Herencia de Hehículos**. Este ejercicio ilustra la herencia en una jerarquía de

vehículos. La clase base Vehiculo incluye atributos protegidos como marca, modelo y ano, junto con métodos arrancar y detener. Las clases derivadas Coche y Motocicleta extienden Vehiculo. La clase Coche tiene atributos adicionales como numeroPuertas y un método abrirMaletero, mientras que la clase Motocicleta incluye un atributo tipoManillar y un método hacerWheelie. Al igual que en el ejemplo anterior, la clase base Vehiculo debe ser abierta y los métodos y atributos que se deseen heredar deben ser protected.

13. **Ejercicio de Clases Abstractas en Animales**. En este ejercicio, se explora el concepto de clases abstractas utilizando una jerarquía de animales. La clase abstracta Animal define un método abstracto hacerSonido() que debe ser implementado por todas las subclases. Las clases derivadas Perro y Gato extienden Animal y proporcionan implementaciones concretas del método hacerSonido(). Al utilizar una lista de objetos de tipo Animal, puedes recorrerla y llamar al método hacerSonido() sin necesidad de realizar casteos, ya que cada subclase proporciona su propia implementación del método abstracto.
14. **Ejercicio de Clases Abstractas en Vehículos**. Este ejercicio muestra cómo usar clases abstractas en una jerarquía de vehículos. La clase abstracta Vehiculo define un método abstracto mover() que todas las subclases deben implementar. Las clases derivadas Coche y Motocicleta heredan de Vehiculo y proporcionan sus propias implementaciones del método mover(). Utilizando una lista de objetos de tipo Vehiculo, puedes recorrerla y llamar al método mover() sin tener que hacer casteos, ya que cada subclase concreta debe implementar el método abstracto definido en la clase base.
15. **Ejercicio de Gestión de Animales con Claes abstractas e Interfaces**.
    1. **Definir la Interfaz `ComportamientoAnimal`**:
        - Crea una interfaz llamada `ComportamientoAnimal` que defina los métodos:
            - `hacerSonido()`: Método abstracto que especifica el sonido que hace el animal.
            - `moverse()`: Método abstracto que especifica cómo se mueve el animal.
            - `dormir()`: Método que proporciona una implementación por defecto que imprime un mensaje indicando que el animal está durmiendo.
    2. **Definir la Clase Abstracta `Animal`**:
        - Crea una clase abstracta llamada `Animal` que implemente la interfaz `ComportamientoAnimal`.
        - Agrega un atributo `nombre` de tipo `String` para almacenar el nombre del animal.
        - Define un método abstracto `alimentarse()`: Método que debe ser implementado por las subclases para definir cómo se alimenta el animal.
        - Proporciona una implementación parcial del método `hacerSonido()` en la clase abstracta, que puede ser sobrescrito por las subclases.
    3. **Implementar Subclases**:
        - **Subclase `Perro`**:
            - Extiende `Animal`.
            - Implementa los métodos:
                - `hacerSonido()`

        - `moverse()`
        - `alimentarse()`
    - Opcionalmente, puedes sobrescribir el método `dormir()` si deseas un comportamiento diferente.
- **Subclase `Gato`**:
    - Extiende `Animal`.
    - Implementa los métodos:
        - `hacerSonido()`
        - `moverse()`
        - `alimentarse()`
    - Opcionalmente, puedes sobrescribir el método `dormir()` si deseas un comportamiento diferente.

4. **Ejemplo de Uso**:
    - Crea una lista de objetos de tipo `ComportamientoAnimal` que incluya instancias de `Perro` y `Gato`.
    - Recorre la lista y llama a los métodos:
        - `hacerSonido()`
        - `moverse()`
        - `dormir()`
    - Muestra en consola el comportamiento de cada animal.
5. **Consideraciones**:
    - Asegúrate de que todas las subclases implementen los métodos abstractos definidos en la interfaz y en la clase abstracta.
    - Aprovecha el polimorfismo para manejar diferentes tipos de animales a través de la interfaz `ComportamientoAnimal`.

FUNCION DE EXTENSIÓN EN KOTLIN

Las funciones de extensión, son como dice la propia palabra una posibilidad de añadir a clases que ya están implementadas nuevas funciones para extender su funcionalidad.

Función de extensión con la clase Date.

Pongamos una extensión para la clase Date, en el cual inventaremos una nueva función llamada myFormat, que devuelva una fecha formateada según la clase SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZZZ"). La idea es que un objeto de tipo Date, al invocar a myFormat devuelva un String con la fecha formateada según hemos comentado anteriormente.

Ampliaremos también a que nos devuelva la longitud de la fecha y otra que nos devuelva la fecha en minúsculas.

e7.1.kt

```
1  fun Date?.myFormat() : String ? {
2      val myDate = SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZZZ", Locale.getDefault())
   ①
3
4      if (this != null)
5          return myDate.format(this)
6      else
7          return null
8  }
9
10 fun Date?.myLength() : Int = this.myFormat()?.length ?: 0
   ②
11
12 fun Date?.toLower () : String ? = this.myFormat()?.lowercase() ?: null
   ③
13
14 fun main() {
15     val dat = Date()
16     println ("Santi, la fecha actual es ${dat.myFormat()} y su longitud es
   ${dat.myLength()}")
17     println ("Ahora devuelvo la misma fecha en minusculas ${dat.toLower()}")
18     println ("La longitud de un null es " + null.myLength())
19 }
```

① Permitimos que un objeto de tipo null, pueda llamar a myFormat y por eso debemos poner Date? indicando que su receptor puede ser nulo. También puede devulver un null, por esto lo indicamos con un **String ?** . Comprobamos si el objeto que llama a myFormat no es nulo, en cuyo caso a partir de una llamada a un objeto de tipo SimpleDateFormat, formateamos la hora y fecha del sistema devolviendo un objeto de tipo Date. Con ese Date y el objeto this, devolvemos la fecha en forma de String. En el caso de que this sea nulo (receptor de la llamada nulo), devolvemos null.

② El receptor de myLenth también puede ser nulo, por tanto devolveremos 0 precedido por el operador elvis **?:**, pero primero se evalúa myFormat() con el operador **?.**, lo que significa que si el método que invoca a length no es nulo, devolverá su longitud.

③ De la misma forma, devolveremos la fecha a minúsculas siempre y cuando no sea nulo el objeto this.myFormat() con el operador **?.**, en cuyo caso devolverá null.

La ejecución:

```
Santi, la fecha actual es 2024-09-11T15:37:29+0000 y su longitud es 24
Ahora devuelvo la misma fecha en minusculas 2024-09-11t15:37:29+0000
La longitud de un null es 0
```

Como podemos ver, cuando necesitemos una o varias funciones de extensión de una clase ya implementada, podemos extenderla de la forma anterior indicada.

Recordar que en un método, ponemos el operador **?** tanto en el receptor que llamará a dicha función como en su retorno, siempre que permitimos nulos. También recordar que podemos simplificar la sentencia del tipo *if (variable != null) return <lo que sea> else return <lo que sea 2>*, utilizando el operador **?.** y el operador elvis **?:**, como lo explicado en el punto 2 y 3.

ACTIVIDADES

1. **Ejercicio de Funciones de Extensión en Cadenas de Texto** Este ejercicio explora cómo agregar funciones adicionales a la clase `String` mediante funciones de extensión.
    - **Definir la Función de Extensión `isPalindrome`**
    - Crea una función de extensión para la clase `String` llamada `isPalindrome` que determine si la cadena es un palíndromo (es decir, si se lee igual de adelante hacia atrás que de atrás hacia adelante).
    - **Definir la Función de Extensión `toPigLatin`**
    - Crea una función de extensión para la clase `String` llamada `toPigLatin` que convierta una cadena en Pig Latin. Para simplificar, considera que Pig Latin se forma moviendo la primera letra al final de la palabra y añadiendo "ay" (por ejemplo, "hello" se convierte en "ellohay").
    - **Definir la Función de Extensión `reverseWords`**
    - Crea una función de extensión para la clase `String` llamada `reverseWords` que invierta el orden de las palabras en una cadena de texto. Por ejemplo, la cadena "hola mundo" se convertiría en "mundo hola".
    - **Definir la Función de Extensión `wordCount`**
    - Crea una función de extensión para la clase `String` llamada `wordCount` que devuelva el número de palabras en una cadena de texto. Considera que las palabras están separadas por espacios.
2. **Ejercicio de Funciones de Extensión en Listas** Este ejercicio explora cómo agregar funciones adicionales a la clase `List` mediante funciones de extensión.
    - **Definir la Función de Extensión `sumSquares`**
    - Crea una función de extensión para la clase `List<Int>` llamada `sumSquares` que devuelva la suma de los cuadrados de los números en la lista.
    - **Definir la Función de Extensión `maxMinDiff`**
    - Crea una función de extensión para la clase `List<Int>` llamada `maxMinDiff` que devuelva la diferencia entre el valor máximo y el valor mínimo de la lista. Si la lista está vacía, devuelve 0.
    - **Definir la Función de Extensión `average`**
    - Crea una función de extensión para la clase `List<Int>` llamada `average` que devuelva el promedio de los números en la lista. Si la lista está vacía, devuelve 0.
    - **Definir la Función de Extensión `filterEven`**
    - Crea una función de extensión para la clase `List<Int>` llamada `filterEven` que devuelva una nueva lista con solo los números pares de la lista original.

DATA ClASS

En ciertas ocasiones, utilizaremos Data Class, porque kotlin ofrece una manera muy simple de crearnos clases cuando necesitamos objetos que desempeñen la función de repositorio. La diferencia entre Class y Data Class, está en la funcionalidad o lógica que queramos implementar. Si nuestras clases sólo seran objetos que representen información, pero que no requieran de lógica alguna, utilizaremos **Data Class**. El porqué es muy sencillo, un Data Class ya viene los métodos toString, equals y hasCode implementados. También permite la opción de copiar un objeto con **copy()**

La forma de declarar un data class es entre paréntesis los atributos que queramos para dicha clase.

Ejemplo data class Persona

Pondremos un ejemplo muy sencillo de Persona con el nombre y la edad. Veremos como se puede imprimir el objeto sin tener que hacer un override del toString, como los podemos copiar en otros objetos, como cada objeto copiado tiene un hashcode diferente cuando los valores son distintos y como podemos compararlos.

e8.1.kt

```
1 data class Persona (var nombre: String, var edad: Int)
2
3 fun main(){
4     val per1 = Persona ("Santi", 40)
5     val per2 = Persona ("Sonia", 35)
6
7     println (per1)
8     println (per1 == per2)
9     val per3 = per1.copy(edad = 50)
10    println (per1 == per3)
11
12    println ("El hascode de Santi es ${per1.hashCode()} y el de Sonia es
  $}{per2.hashCode()}")
13    println ("El hascode de la copia de Santi es ${per3.hashCode()}")
14
15 }
```

Cuando definimos atributos fuera del constructor principal (), tener cuidado porque no contarán para las implementaciones automáticas de equals, hashCode y toString. Hagamos un último ejemplo muy sencillo, en el que no pondremos el teléfono de la persona y veremos que sucede.

e8.2.kt

```
1 data class Persona (var nombre: String, var edad: Int) {
2     var telefono : String ? = null
3
4
5 }
```

```
 6 fun main(){
 7     val per1 = Persona ("Santi", 40)
 8     val per2 = Persona ("Sonia", 35)
 9     per1.telefono = "953 111 222"
10     per2.telefono = "953 222 222"
11
12     val per3 = per1.copy()              ①
13
14     println (per1)
15     println (per2)
16     println (per1 == per2)
17     println (per1 == per3)              ②
18
19 }
```

① No podemos utililizar copy con atributos que no estén en el constructor principal.

② Como es lógico, per1 tiene un teléfono diferente al null de per3 y sin embargo, dicen que son iguales.

La ejecución:

```
Persona(nombre=Santi, edad=40)
Persona(nombre=Sonia, edad=35)
false
true
```

ACTIVIDADES

1. **Ejercicio 1: Comparación y Copia de Data Class** Este ejercicio explora la funcionalidad de comparación y copia en una `data class`.
   - **Definir la Data Class `Producto`**
     - Crea una `data class` llamada `Producto` con los atributos:
       - `nombre: String`
       - `precio: Double`
   - **Ejemplo de Uso**
     - Crea dos instancias de `Producto` con diferentes valores.
     - Imprime ambas instancias.
     - Compara las dos instancias usando el operador `==`.
     - Crea una copia de una de las instancias y modifica el precio en la copia.
     - Imprime la instancia original, la copia, y compara ambas para verificar si son iguales.
2. **Ejercicio 2: Extensión de Data Class con Métodos Adicionales** Este ejercicio muestra cómo extender una `data class` con métodos adicionales.
   - **Definir la Data Class `Empleado`**
     - Crea una `data class` llamada `Empleado` con los atributos:
       - `nombre: String`
       - `salario: Double`
   - **Agregar Método Adicional**
     - Dentro de la `data class`, añade un método llamado `anualSalario` que devuelva el salario anual del empleado (suponiendo 12 meses).
   - **Ejemplo de Uso**
     - Crea una instancia de `Empleado`.
     - Imprime la instancia.
     - Llama al método `anualSalario` e imprime el resultado.
3. **Ejercicio 3: Uso de Data Class con Atributos Opcionales** Este ejercicio explora el uso de atributos opcionales en una `data class`.
   - **Definir la Data Class `Libro`**
     - Crea una `data class` llamada `Libro` con los atributos:
       - `titulo: String`
       - `autor: String`
       - `anioPublicacion: Int`
       - `genero: String?` (opcional)
   - **Ejemplo de Uso**

- Crea dos instancias de `Libro`: una con todos los atributos y otra sin el atributo `genero`.
- Imprime ambas instancias.
- Usa el método `copy()` para crear una copia de una instancia cambiando el `anioPublicacion`.
- Imprime la instancia original, la copia, y compara ambas para verificar si son iguales.

LISTAS INMUTABLES Y MUTABLES

En kotlin, Las listas inmutables son aquellas cuyo contenido no puede ser modificado después de su creación. En otras palabras, una vez que se crea una lista inmutable, no puedes añadir, eliminar o modificar sus elementos.

Uso: Se utilizan cuando quieres garantizar que la lista no será alterada a lo largo del ciclo de vida de tu aplicación. Esto ayuda a prevenir errores inesperados y asegura la consistencia de los datos.

Listas inmutables

Su declaración es del tipo:

```
1 val numbers: List<Int> = listOf(1, 2, 3, 4, 5)
2 // numbers.add(6)  ①
3 println(numbers)  ②
```

① Error: Unresolved reference: add

② Imprime: [1, 2, 3, 4, 5]

Tenemos dos opciones a utilizar.

1. listOf() Lo utilizamos para crear una lista inmutable de elementos específicos que pasamos como parámetros.
2. List() Lo utilizamos para crear una lista inmutable de elementos, pero utilizaremos una lambda para inicializar sus valores iniciales, anteponiendo el número de elementos.

Ejemplo de lista inmutable utilizando listOf y List()

En el siguiente ejemplo, veremos como podemos utilizar ambas opciones pero de forma diferente. Tendremos un data class PersonalData, ya utilizado en otros temas y en el main crearemos varias listas inmutables jugando con el acceso a ciertos elementos.

e9.1.kt

```
1 data class PersonalData ( val name: String, val phone: String ? ){
2
3     override fun toString () = if (phone == null)
  ①
4                                     ("Nombre: $name y no tiene telefono")
5                                 else
6                                     ("Nombre: $name, Telefono: $phone")
7 }
8
9 fun load(): PersonalData = PersonalData ("Anonimo_repetido", null)
  ②
10
11 fun main(){
```

```
12
13      val listSamePersonal : List<PersonalData> = List<PersonalData>(3){
    ③
14          PersonalData ("Santi", "953 34 54 34")
15      }
16
17      println ()
18
19      val listAnonimous : List<PersonalData> = List<PersonalData>(3, { load() })
    ④
20
21      val listPersonal : List<PersonalData> = listOf (
    ⑤
22           PersonalData ("Santi", "953 34 54 34"),
23           PersonalData ("Sonia", "953 34 54 35"),
24           PersonalData ("Guille", null),
25           PersonalData ("Diego", null)
26
27      )
28
29      listAnonimous.forEach { println(it) }
    ⑥
30      println ("-----")
31      listSamePersonal.forEach { println(it) }
    ⑦
32      println ("-----")
33      listPersonal.forEach { personal -> println (personal) }
34      println ("----- Total de la lista ${listPersonal.size}")
    ⑧
35      println ("Primero de la lista -> ${listPersonal.first()}")
    ⑨
36      println ("En una posicion de la lista, pos = 1 -> ${listPersonal[1]}")
    ⑩
37      println ("Ultimo de la lista -> ${listPersonal.last()}")
    ⑪
38      println ("-------")
39      for (personal in listPersonal)
    ⑫
40          println ("Personal -> $personal")
41
42  }
```

① Sobreescribimos el toString del data class, para que informe de que no tiene teléfono

② Función que devuelve siempre el mismo objeto cada vez que se invoque.

③ Nos creamos una lista inmutable de 3 objetos y utilizamos una lambda con el mismo objeto explícito.

④ Es igual que en el punto anterior, pero el objeto explícito lo sustituímos por la invocación del método load.

⑤ Nos creamos una lista inmutable, con los elementos a insertar. Aquí no podemos utilizar una lambda.

⑥ Recorremos la lista como cualquier array utilizando el forEach {}.

⑦ Lo mismo para la otra lista, ya que son List. El forEach se utiliza tanto en List como el listOf.

⑧ Imprimimos el tamaño de la lista.

⑨ Imprimimos el primero de la lista

⑩ Imprimimos el elemento den posición 1 (según índice).

⑪ Imprimimos el último elemento de la lista.

⑫ Recorremos la lista con un simple for.

La ejecución:

```
Nombre: Anonimo_repetido y no tiene telefono
Nombre: Anonimo_repetido y no tiene telefono
Nombre: Anonimo_repetido y no tiene telefono
-----
Nombre: Santi, Telefono: 953 34 54 34
Nombre: Santi, Telefono: 953 34 54 34
Nombre: Santi, Telefono: 953 34 54 34
-----
Nombre: Santi, Telefono: 953 34 54 34
Nombre: Sonia, Telefono: 953 34 54 35
Nombre: Guille y no tiene telefono
Nombre: Diego y no tiene telefono
----- Total de la lista 4
Primero de la lista -> Nombre: Santi, Telefono: 953 34 54 34
En una posicion de la lista, pos = 1 -> Nombre: Sonia, Telefono: 953 34 54 35
Ultimo de la lista -> Nombre: Diego y no tiene telefono
-------
Personal -> Nombre: Santi, Telefono: 953 34 54 34
Personal -> Nombre: Sonia, Telefono: 953 34 54 35
Personal -> Nombre: Guille y no tiene telefono
Personal -> Nombre: Diego y no tiene telefono
```

Que sean listas inmutables, no quiere decir que en los atributos de sus elementos (en caso de objetos), podamos modificar sus referencias.

Listas mutables

Las listas mutables, son las que utilizamos cuando queremos alterar el número de elementos. Lo declaramos como **MutableList** y para hacernos una idea, es lo más parecido a los ArrayList de Java. En este caso, cuando creemos una lista mutable, **NO** estamos obligados a inicializar sus elementos como en el caso de las listas inmutables. Podemos elegir la opción de crearnos una lista mutable con el número de elementos y una lambda, crearnos una lista mutable vacía e ir insertando después. Veamos mejor un ejemplo para entenderlo.

e9.2.kt

```
1  data class PersonalData ( val name: String, val phone: String ? ){
2      override fun toString () = if (phone == null)
3                                      ("Nombre: $name y no tiene telefono")
4                                  else
5                                      ("Nombre: $name, Telefono: $phone")
6  }
7
8  fun load(): PersonalData = PersonalData ("Anonimo_repetido", null)
9
10
11 fun main(){
12
13     val listSamePersonal : MutableList<PersonalData> = MutableList<PersonalData>(
   3){            ①
14         PersonalData ("Santi", "953 34 54 34")
15     }
16
17     val listAnonimous : MutableList<PersonalData> = MutableList<PersonalData>(3, {
   load() })    ②
18
19     val listAutomaticPersonal = MutableList<PersonalData>(4){
   ③
20         index-> PersonalData( "Nombre_$index", null )
21     }
22
23     val listPersonal : MutableList<PersonalData> = mutableListOf()
   ④
24
25     listPersonal.add(PersonalData ("Santi", "953 34 54 34"))
   ⑤
26     listPersonal.add(PersonalData ("Sonia", "953 34 54 35"))
27     listPersonal.add(PersonalData ("Diego", null))
28     listPersonal.add(PersonalData ("Guille", null))
29
30     listAnonimous.forEach { println(it) }
31     println ("-----")
32
33     listSamePersonal.forEach { println(it) }
34     println ("-----")
35
36     listPersonal.forEach { personal -> println (personal) }
37     println()
38
39     println ("----- Total de la lista ${listPersonal.size}")
40     println ("Primero de la lista -> ${listPersonal.first()}")
41     println ("En una posicion de la lista, pos = 1 -> ${listPersonal.get(1)}")
42     println ("Ultimo de la lista -> ${listPersonal.last()}")
43     println()
44
```

```
45      println ("-------")
46
47      for (personal in listPersonal)
48          println ("Personal -> $personal")
49
50      println ( "Me cargo el de posicion index 0 ")
    ⑤
51      listPersonal.removeAt(0)
52
53      val numbersPhone = listPersonal.count{ it.phone != null }
    ⑥
54      println ("El numero de telefonos disponibles son $numbersPhone")
55
56      println()
57      println (" Ahora me cargaré a todos los que no tienen telefono asignado ")
    ⑦
58      listPersonal.removeAll { it.phone == null }
59
60      println()
61      listPersonal.forEach{ println ("Personal: $it") }
62
63      println ("Ahora me cargaré a los que tienen 5 o mas letras en su nombre")
64      listPersonal.removeAll { it.name.count() >= 5 }
    ⑧
65
66      println()
67      println ("El numero de Personal es de ${listPersonal.count()}")
68
69      println()
70      println ("Imprimiremos nuestra lista automatica")
71      listAutomaticPersonal.forEach{
72          println(it)
73      }
74
75  }
```

① Puedo crearme un MutableList con el número de elementos y una lamda con los objetos dentro del cuerpo.

② Puedo crearme un MutableList con el número de elementos y una lambda que invoque a otra función de creación de objeto.

③ Mismo caso que el punto 1, pero con otros datos.

④ El caso de **mutableListOf** es importante cuando sólo quiero crear una lista mutable vacía. Mas adelante iré incorporando los objetos. Un caso parecido en java, sería algo como **ArrayList<PersonalData> lista = ArrayList<PersonalData>();**.

⑤ Elimino el elemento en posición 0.

⑥ Devuelvo el número de elementos cuya expresión en una lamda **(numero de teléfonos nulos)**.

⑦ Elimino todos los objetos, cuya expresión en una lamda **(teléfonos nulos)**. Muy interesante para eliminar por ejemplo por dni.

⑧ Eliminamos todos los objetos, cuyo nombre sea mayor a 5 caracteres. Utilizamos una lamda.

La ejecución:

```
Nombre: Anonimo_repetido y no tiene telefono
Nombre: Anonimo_repetido y no tiene telefono
Nombre: Anonimo_repetido y no tiene telefono
-----
Nombre: Santi, Telefono: 953 34 54 34
Nombre: Santi, Telefono: 953 34 54 34
Nombre: Santi, Telefono: 953 34 54 34
-----
Nombre: Santi, Telefono: 953 34 54 34
Nombre: Sonia, Telefono: 953 34 54 35
Nombre: Diego y no tiene telefono
Nombre: Guille y no tiene telefono

----- Total de la lista 4
Primero de la lista -> Nombre: Santi, Telefono: 953 34 54 34
En una posicion de la lista, pos = 1 -> Nombre: Sonia, Telefono: 953 34 54 35
Ultimo de la lista -> Nombre: Guille y no tiene telefono

-------
Personal -> Nombre: Santi, Telefono: 953 34 54 34
Personal -> Nombre: Sonia, Telefono: 953 34 54 35
Personal -> Nombre: Diego y no tiene telefono
Personal -> Nombre: Guille y no tiene telefono
Me cargo el de posicion index 0
El numero de telefonos disponibles son 1

 Ahora me cargaré a todos los que no tienen telefono asignado

Personal: Nombre: Sonia, Telefono: 953 34 54 35
Ahora me cargaré a los que tienen 5 o mas letras en su nombre

El numero de Personal es de 0

Imprimiremos nuestra lista automatica
Nombre: Nombre_0 y no tiene telefono
Nombre: Nombre_1 y no tiene telefono
Nombre: Nombre_2 y no tiene telefono
Nombre: Nombre_3 y no tiene telefono
```

En ocasiones necesitaremos convertir o devolver una lista mutable a partir de una inmutable. Para ello, utilizaremos el método **toMutableList()**.

e9.3.kt

```
1 fun main() {
2     val immutableList = listOf("Elemento 1", "Elemento 2", "Elemento 3")
```

```
    val mutableList = immutableList.toMutableList()                    ①

    println("Lista mutable: $mutableList")

    mutableList.add("Elemento 4")
    mutableList[1] = "Elemento Modificado"

    println("Lista mutable después de modificar: $mutableList")
    println("Lista inmutable original: $immutableList")
}
```

① A partir de una lista inmutable, devolvemos la misma lista pero mutable. Recordemos que mutableList es de tipo **MutableList<String>**.

Pondremos otro ejemplo sencillo. Crearemos una lista mutable de 20 números aleatorios entre 1 y 6, mostraremos el numero de elementos con valor 1, nos cargaremos todos los elementos con valor 6 y los imprimiremos en pantalla. No pondré comentarios.

e9.4.kt

```
import kotlin.random.Random

fun printList (list : MutableList <Int>) {
    list.forEach { print ( "$it, ") }
    println()
}

fun main (){
    val listInt : MutableList <Int> = MutableList <Int> (20) {
        Random.nextInt(1, 7)
    }

    val number = listInt.count {
        v-> v == 1
    }

    printList (listInt)
    println( "---------")
    println ("Numero de elementos con valor 1 es de $number")
    println( "---------")
    println ("Ahora me cargaré a los valores 6")
    listInt.removeAll {
        it == 6
    }
    printList (listInt)
    println( "---------")
    println ("También puedo imprimir la lista directamente sin función")
    println (listInt)

}
```

La ejecución:

```
5, 5, 3, 3, 5, 2, 6, 1, 4, 5, 1, 2, 4, 2, 3, 4, 2, 3, 1, 2,
---------
Numero de elementos con valor 1 es de 3
---------
Ahora me cargaré a los valores 6
5, 5, 3, 3, 5, 2, 1, 4, 5, 1, 2, 4, 2, 3, 4, 2, 3, 1, 2,
---------
También puedo imprimir la lista directamente sin función
[5, 5, 3, 3, 5, 2, 1, 4, 5, 1, 2, 4, 2, 3, 4, 2, 3, 1, 2]
```

ACTIVIDADES

Ejercicio de Creación y Acceso a Listas Inmutables: Define una lista inmutable de enteros usando `listOf()`, e inicialízala con los números del 1 al 5. Imprime la lista completa y accede a los elementos individuales mediante índices. Muestra el primer y el último elemento de la lista.

Ejercicio de Transformación de Listas Inmutables: Crea una lista inmutable de strings que contenga nombres de ciudades. Usa el método `map` para convertir todos los nombres a mayúsculas. Imprime la lista original y la lista transformada para comparar los resultados.

Ejercicio de Filtrado de Listas Inmutables: Declara una lista inmutable de números enteros. Utiliza el método `filter` para obtener una nueva lista que solo contenga los números pares. Imprime la lista original y la lista filtrada para verificar el resultado.

Ejercicio de Reversión de Listas Inmutables: Inicializa una lista inmutable de objetos `PersonalData`, cada uno con un nombre y un teléfono. Usa el método `reversed` para crear una nueva lista con el orden invertido de los elementos. Imprime ambas listas para observar el cambio en el orden.

Ejercicio de Comparación de Listas Inmutables: Crea dos listas inmutables de enteros, una con los números del 1 al 5 y otra con los números del 3 al 7. Usa el método `intersect` para encontrar los elementos comunes entre ambas listas y el método `subtract` para encontrar los elementos únicos en la primera lista que no están en la segunda. Imprime los resultados de estas operaciones.

Ejercicio de Inicialización y Modificación de Listas Mutables: Crea una lista mutable de enteros con cinco elementos inicializados a cero usando `MutableList()`. Luego, agrega elementos adicionales a la lista y modifica algunos valores existentes. Imprime la lista después de cada operación para observar los cambios.

Ejercicio de Eliminación y Contador en Listas Mutables: Inicializa una lista mutable de strings con varios nombres. Usa el método `removeAt` para eliminar un elemento en una posición específica y el método `count` para contar cuántos nombres tienen más de 4 caracteres. Imprime la lista antes y después de la eliminación y muestra el conteo de nombres largos.

Ejercicio de Conversión de Listas Mutables: Declara una lista mutable de enteros y llena la lista con números del 1 al 10. Luego, convierte esta lista en una lista inmutable utilizando `toMutableList()`. Modifica algunos elementos de la lista mutable resultante y muestra tanto la lista mutable como la lista inmutable original para comparar.

Ejercicio de Filtrado y Eliminación de Elementos en Listas Mutables: Crea una lista mutable de objetos `PersonalData` con diferentes nombres y teléfonos. Utiliza `removeAll` para eliminar todos los objetos que tienen un teléfono nulo y `removeIf` para eliminar todos los objetos cuyo nombre tenga menos de 5 caracteres. Imprime la lista después de cada operación para verificar los resultados.

Ejercicio de Generación y Manipulación de Listas Mutables Aleatorias: Genera una lista mutable de 15 números aleatorios entre 1 y 10. Usa `count` para contar cuántos números son menores que 5 y `removeAll` para eliminar todos los números mayores de 8. Imprime la lista original, el número de elementos menores que 5, y la lista final después de eliminar los números mayores.

MAPAS

Los mapas son estructuras muy utilizadas ya que permiten identificar a un objeto a partir de una clave. En otros lenguajes se conocen como bibliotecas y debemos de tener en cuenta que las claves sean únicas por cada uno de los objetos. Al igual que en las listas, tenemos mapas inmutables **mapOf** y mutables **MutableMap**. Veremos las dos formas de crearnos los mapas y algunos métodos interesantes.

Mapas inmutables.

mapOf es una función que nos crea un mapa no mutable, es decir que una vez que se crea, no podemos modificarlo. El acceso a cada uno de los elementos del mapa, se hace por medio de su clave.

e10.1.kt

```
fun main() {
    val immutableMap = mapOf(
        "clave1" to "valor1",
        "clave2" to "valor2",
        "clave3" to "valor3"
    )

    println("Map inmutable: $immutableMap")
    println("------")

    println("Valor asociado con 'clave2': ${immutableMap["clave2"]}")
    println("------")

    for ((clave, valor) in immutableMap) {
        println("Clave: $clave, Valor: $valor")
    }
}
```

Pondremos un ejemplo sencillo de un map de paises. Hay que tener en cuenta, que un map es de la forma "clave" to "valor" lo que en otros lenguajes se ve de la forma "clave" ⇒ "valor". Esa <clave, valor>, representa lo que llamamos **Pair<Tipo><Tipo>**, por eso en este ejemplo, un Pair se representa como un objeto que está compuesto por dos partes que como intuirás, el primero será la clave y el segundo el valor. Ya sabemos pues, que un mapa lo podemos crear de dos formas, como "clave" to "valor" ó como Pair("clave", "valor").

También, podemos tener la necesidad de devolver un map a partir de una lista de Pair. Para ello utilizaremos el método de lista **toMap()**. En el siguiente ejemplo, veremos como crearnos un mapOf utilizando tanto los **Pair**, como la opción **to**.

e10.2.kt

```
fun main () {
```

```
    val countries : Map <String, Int> = mapOf (                          ①
        Pair ("España", 47000000),
        Pair ("Francia", 60000000),
        "Alemania" to 80000000
    )

    val listCities : List<Pair<String, Int>>  = listOf (                 ②
        "Albacete" to 200000,
        "Jaen" to 120000,
        "Toledo" to 180000
    )

    val cities : Map<String, Int> = listCities.toMap()                   ③

    println( "Paises----" )
    countries.forEach { println ("Pais-> ${it.key}, Hab-> ${it.value}")}  ④
    println( "Ciudades----" )
    cities.forEach {
        (country, hab) -> println ( "Pais-> $country, Hab-> $hab" )      ⑤
        }

    val numHabTo = cities.count { (country, hab ) -> hab >150000 }       ⑥
    println ( "-----" )
    println ( "Número de ciudades con mas de 150000 habitantes es de $numHabTo" )
    println ( "-----" )

    var totalHab = 0
    countries.forEach { totalHab += it.value  }                          ⑦
    println ( "Número de habitantes de todos los paises es de $totalHab" )
    println ( "-----" )

    var totalHabless = 0
    cities.forEach { if (it.value <200000) totalHabless += it.value }    ⑧
    println ( "Suma de habitantes de ciudades inferiores a 200000 es de
$totalHabless" )
    println ( "-----" )

}
```

① Nos creamos un mapOf de dos formas, con Pair y con to. Recordar que hay que tener en cuenta los tipos de la clave y del valor.

② La lista debe ser de tipo Pair<String, Int>, por tanto **List<Pair<String, Int>>**.

③ Devolvemos un mapa inmutable a partir de la lista de Pair.

④ Recorremos todos los elementos del mapa con un forEach, imprimiendo su clave, valor con **it.key**, **it.value**.

⑤ Hacemos los mismo que en el punto 4, pero redefinimos los parámetros de la lambda it.key con el nombre country y it.value con el nombre hab.

⑥ Utilizamos una lambda para contar el numero de ciudades con habitantes de mas de 150000.

⑦ Dentro del forEach, sumamos los habitantes de todas las ciudades.

⑧ Dentro de la lambda del forEach, sumamos todos los habitantes cuyas ciudades tengan menos de 200000.

Mapas mutables.

Hasta ahora, los mapas inmutables los podemos utilizar perfectamente como biblioteca de objetos. En la asignatura de Android, podemos utilizar los map para simular un acceso a datos que de momento estén en memoria. En este punto, recordaremos como trabajar con los map mutables, es decir, los que si se pueden modificar, añadir, eliminar, etc.

Pondremos un ejemplo más extenso y completo. Imaginar que tenemos alumnos mediante un Mapa en el que su dni es la key y el mismo objeto de tipo Alumno es el value. Cada alumno tiene una lista mutable de sus asignaturas. Cada asignatura, a parte del nombre debe de contar con una nota. Para cada alumno, podremos añadir asignaturas, eliminarlas por su nombre y devolver información.

En este ejemplo, veremos el uso de apply como alternativa a realizar una configuración previa justamente en la creación de un objeto. El método apply, hace referencia al mismo this y acepta un bloque con todo aquello que queramos inicializar o configurar. Veamos el ejemplo.

e10.3.kt

```
1  data class Signature (var name : String, var note : Double = 0.0){
2      override fun toString() = "Asignatura: $name y su nota es de $note \n"
3  }
4
5
6  class Alumn(var dni: String, var name: String) {
7      var signatures: MutableList<Signature>
8
9      init {
10         signatures = mutableListOf()                                    ①
11     }
12
13     fun addSignature(name: String, note: Double) {
14         signatures.add(Signature(name, note))                           ②
15     }
16
17     fun removeSignature(name: String) {
18         signatures.removeAll { it.name == name }                        ③
19     }
20
21     override fun toString(): String {
22         var dev: String = "(dni): $dni, (Alumno): $name \n"
23         signatures.forEach {
24             dev += it
25         }
26         return dev
27     }
```

```
28
29      fun devSignatureNotab(): String {                                    ④
30          var dev: String = ""
31          signatures.forEach {
32              if (it.note >= 7)
33                  dev += it
34          }
35          return dev
36      }
37
38
39  } //fin de la clase
40
41
42  fun printAll ( al: MutableMap <String, Alumn>)  {                        ⑤
43          al.forEach { println (it.value)}
44  }
45
46  fun printAllWithSignature ( al: MutableMap <String, Alumn>, fn: (String) -> Unit){
    ⑥
47      al.forEach {
48          fn (it.key)
49      }
50  }
51
52
53  fun main() {
54      val alumns: MutableMap<String, Alumn> = mutableMapOf<String, Alumn>().apply {
    ⑦
55          listOf(
56              Pair("11111", Alumn("11111", "Santiago")),
57              Pair("22222", Alumn("22222", "Sonia")),
58              Pair("33333", Alumn("33333", "Mariano"))
59          ).forEach { (dni, alum) ->
60              put(dni, alum)
61          }
62      }
63
64      // Añadir asignaturas y mostrar información
65      var alum: Alumn? = alumns["11111"]
66      alum?.apply {
    ⑧
67          addSignature("Matematicas", 6.87)
68          addSignature("Fisica", 7.87)
69          addSignature("Tecnologia", 9.87)
70          addSignature("Filosofia", 3.68)
71
72          println("------Datos de un alumno-----")
73          println(this)
74          println("------Asignaturas con notable o más")
75          println(devSignatureNotab())
```

```
76
77          // Eliminar una asignatura y mostrar la información actualizada
78          removeSignature("Fisica")
79          println("------Datos del alumno después de eliminar asignatura-----")
80          println(this)
81      }
82
83      alum = alumns["22222"]
84      alum?.apply {
85          addSignature("Matematicas", 7.87)
86          addSignature("Fisica", 6.87)
87          addSignature("Tecnologia", 8.87)
88          addSignature("Filosofia", 9.68)
89
90          println("------Datos de un alumno-----")
91          println(this)
92          println("------Asignaturas con notable o más")
93          println(devSignatureNotab())
94      }
95
96      println("Ahora mostraré todos los datos de todos los alumnos y sus
    asignaturas")
97      printAll(alumns)
98      println("---------")
99
100     println("Ahora utilizaré algunas lambdas")
101     printAllWithSignature(alumns) { dni ->
    ⑨
102         println(alumns[dni])
103         val alum: Alumn? = alumns[dni]
104         if (alum?.signatures?.isEmpty() == true) {
105             println("No tiene asignada ninguna asignatura")
106         }
107     }
108
109     println("")
110     println("Ahora mostraré con una lambda, el número de asignaturas suspensas")
111     printAllWithSignature(alumns) {
    ⑩
112         val alum: Alumn? = alumns[it]
113         val num = alum?.signatures?.count { signature -> signature.note < 5 } ?: 0
114         println("El alumno ${alum?.name} tiene $num asignaturas suspensas")
115     }
116 }
```

① Cada alumno, inicializa sus asignaturas a partir de un mutableListOf(). Esto genera una lista mutable vacía.

② Añadimos una asignatura a la lista mutable con el add.

③ Eliminamos todas las asignaturas por el nombre.

④ Devolvemos todas las asignaturas con nota mayor o igual a 7.

⑤ Recorremos todo el map y mostramos en pantalla los datos de los alumnos.

⑥ Llamada de orden superior. Por cada alumno, Contabilizamos las asignaturas que tiene suspensas y las imprimimos. Pasamos todo el mapa y una función a la que invocaremos dentro y cuya lamda será tratada en el punto 9 y 10.

⑦ Creamos el map de alumnos. Para ello, vemos que dentro de apply aceptamos un bloque de inicialización. Dicho bloque consiste en la creación de una lista inmutable de alumnos con **listOf**, recorriendo cada elemento con un forEach y haciendo un put dentro del map.

⑧ Creo un alumno y como tiene un atributo de tipo **mutableListOf**, con apply inicializo sus asignaturas invocando al método addSignature, tantas veces como asignaturas quiera crear.

⑨ Lambda del punto 6.

⑩ Lo mismo que en el punto 9.

La ejecución:

```
------Datos de un alumno-----
(dni): 11111, (Alumno): Santiago
Asignatura: Matematicas y su nota es de 6.87
Asignatura: Fisica y su nota es de 7.87
Asignatura: Tecnologia y su nota es de 9.87
Asignatura: Filosofia y su nota es de 3.68

------Asignaturas con notable o más
Asignatura: Fisica y su nota es de 7.87
Asignatura: Tecnologia y su nota es de 9.87

------Datos del alumno después de eliminar asignatura-----
(dni): 11111, (Alumno): Santiago
Asignatura: Matematicas y su nota es de 6.87
Asignatura: Tecnologia y su nota es de 9.87
Asignatura: Filosofia y su nota es de 3.68

------Datos de un alumno-----
(dni): 22222, (Alumno): Sonia
Asignatura: Matematicas y su nota es de 7.87
Asignatura: Fisica y su nota es de 6.87
Asignatura: Tecnologia y su nota es de 8.87
Asignatura: Filosofia y su nota es de 9.68

------Asignaturas con notable o más
Asignatura: Matematicas y su nota es de 7.87
Asignatura: Tecnologia y su nota es de 8.87
Asignatura: Filosofia y su nota es de 9.68

Ahora mostraré todos los datos de todos los alumnos y sus asignaturas
(dni): 11111, (Alumno): Santiago
Asignatura: Matematicas y su nota es de 6.87
```

```
Asignatura: Tecnologia y su nota es de 9.87
Asignatura: Filosofia y su nota es de 3.68

(dni): 22222, (Alumno): Sonia
Asignatura: Matematicas y su nota es de 7.87
Asignatura: Fisica y su nota es de 6.87
Asignatura: Tecnologia y su nota es de 8.87
Asignatura: Filosofia y su nota es de 9.68

(dni): 33333, (Alumno): Mariano

---------
Ahora utilizaré algunas lambdas
(dni): 11111, (Alumno): Santiago
Asignatura: Matematicas y su nota es de 6.87
Asignatura: Tecnologia y su nota es de 9.87
Asignatura: Filosofia y su nota es de 3.68

(dni): 22222, (Alumno): Sonia
Asignatura: Matematicas y su nota es de 7.87
Asignatura: Fisica y su nota es de 6.87
Asignatura: Tecnologia y su nota es de 8.87
Asignatura: Filosofia y su nota es de 9.68

(dni): 33333, (Alumno): Mariano

No tiene asignada ninguna asignatura

Ahora mostraré con una lambda, el número de asignaturas suspensas
El alumno Santiago tiene 1 asignaturas suspensas
El alumno Sonia tiene 0 asignaturas suspensas
El alumno Mariano tiene 0 asignaturas suspensas
```

ACTIVIDADES

1. **Ejercicio de Creación y Acceso a Mapas**: Crea un mapa inmutable que asocie tres claves a valores de tipo string. Luego, accede a un valor utilizando su clave y recorre el mapa para imprimir cada clave y su correspondiente valor.
2. **Ejercicio de Mapa de Paises y Ciudades**: Inicializa un mapa inmutable de países con sus respectivas poblaciones. Crea una lista de pares (pares de ciudad y población) y conviértela en un mapa inmutable. Imprime el contenido de ambos mapas y cuenta cuántas ciudades tienen una población superior a 200,000 habitantes.
3. **Ejercicio de Suma de Habitantes en Mapas**: Crea un mapa inmutable que asocie nombres de ciudades con sus poblaciones. Calcula el total de habitantes de todas las ciudades usando `forEach`. Luego, calcula la suma de las poblaciones de las ciudades que tienen menos de 150,000 habitantes.
4. **Ejercicio de Transformación de Lista a Mapa**: Declara una lista de pares (clave-valor) donde cada par representa una ciudad y su población. Convierte esta lista en un mapa inmutable usando `toMap()`. Imprime el mapa resultante y utiliza `forEach` para imprimir cada ciudad con su población.
5. **Ejercicio de Conteo y Filtrado en Mapas**: Crea un mapa inmutable con varias entradas de nombres de productos y sus precios. Usa `count` para contar cuántos productos tienen un precio superior a un valor específico. Además, utiliza `filter` para obtener un nuevo mapa que contenga solo los productos cuyo precio esté en un rango determinado y muestra el resultado.
6. **Ejercicio de Mapas mutables**:
    1. **Definir la Clase `Autor`**:
        - Crea una clase llamada `Autor` con los siguientes atributos:
            - `nombre`: de tipo `String` que almacena el nombre del autor.
            - `edad`: de tipo `Int` que almacena la edad del autor.
        - Define los métodos:
            - `toString()`: Método que devuelve una representación en cadena del autor en el formato `"Nombre: [nombre], Edad: [edad]"`.
    2. **Definir la Clase `Libro`**:
        - Crea una clase llamada `Libro` con los siguientes atributos:
            - `titulo`: de tipo `String` que almacena el título del libro.
            - `añoPublicacion`: de tipo `Int` que almacena el año de publicación del libro.
            - `autor`: de tipo `Autor` que representa al autor del libro.
        - Define los métodos:
            - `toString()`: Método que devuelve una representación en cadena del libro en el formato `"Título: [titulo], Año: [añoPublicacion], Autor: [autor]"`.
            - `esAutor(adm): Boolean`: Método que recibe un objeto `Autor` y devuelve `true` si el libro ha sido escrito por ese autor, `false` en caso contrario.
    3. **Implementar Ejemplo de Uso**:

- Crea instancias de `Autor` y `Libro` con diferentes datos.
- Crea una lista de libros.
- Recorre la lista y utiliza los métodos para imprimir la información de cada libro y verificar si un libro ha sido escrito por un autor específico.
- Muestra en consola:
  - La lista de libros con sus detalles.
  - La verificación de autor para cada libro.

4. **Consideraciones**:
   - Asegúrate de que los métodos `toString()` devuelvan la información en el formato correcto.
   - Utiliza correctamente el método `esAutor` para determinar la relación entre libros y autores.
   - Aprovecha la polimorfismo para manejar diferentes tipos de libros y autores a través de sus instancias.

CALLBACK

Un callback es un concepto en programación en el cual una función (función1) recibe como parámetro otra función (función2). Cuando función1 haya completado su tarea, invoca a función2, ejecutando la lógica definida en ella. Esto permite una programación más flexible y modular, ya que se delega parte del control del flujo a otras funciones.

Los callbacks son especialmente útiles en operaciones asíncronas, donde no sabemos cuánto tiempo tardará una tarea en completarse (por ejemplo, acceder a una base de datos, leer un archivo o hacer una solicitud a una API). A diferencia de las operaciones sincrónicas (donde el hilo de ejecución principal espera a que se complete la operación antes de continuar), las operaciones asíncronas permiten que el hilo principal siga ejecutándose normalmente, sin bloquearse.

Una vez que la operación asíncrona finaliza, el callback (función2) es invocado para ejecutar la lógica que hemos definido, como por ejemplo procesar los datos obtenidos o actualizar la interfaz de usuario. Esta es la principal diferencia entre sincronía (donde el código se ejecuta de manera secuencial) y asincronía (donde las tareas pueden completarse en un futuro sin bloquear el flujo principal).

Llamada síncrona.

El código sigue un flujo secuencial, lo que significa que el hilo principal quedará bloqueado hasta que la operación de acceso a datos se complete. En este caso, imaginemos que la función realiza una operación que toma un tiempo considerable, como el acceso a una base de datos (BBDD), y tarda un cierto tiempo en devolver una respuesta. Durante este período de espera, el hilo principal no puede realizar otras tareas y debe esperar a que la operación termine.

Una vez que la operación de lectura de datos ha finalizado, el callback (una función pasada como parámetro) es ejecutado. Este callback será el encargado de recibir los datos obtenidos y permitir que el programa, en el main, procese los resultados de manera adecuada.

Es decir, el callback se invoca solo después de que la operación costosa ha terminado, permitiendo que el flujo del programa continúe y que los datos sean manipulados de acuerdo a la lógica que se desee implementar.

Ejemplo:

e11.1.kt

```
1 fun getDataBBDD (sql : String, callback: (List<String>)-> Unit){
  ①
2     println ("Se procede a petición de datos.....")
3     Thread.sleep(3000)
  ②
4     val data : List<String> = List<String>(3){"Datos obtenidos a partir de $sql"}
  ③
5     println ("Se han devuelto los datos")
6     callback (data)
  ④
```

```
}

fun main(){
    println ("Comenzamos nuestra aplicación..")
    Thread.sleep(1000)
⑤
    println ("Ahora procedemos a petición de datos...")
    getDataBBDD ("name = santi") {
⑥
        data -> println ("Los datos obtenidos son: ")
        data.forEach { println (it) }
    }
}
```

① getDataBBDD es una función que simula la petición de datos. Recibe dos parámetros: sql: Una cadena de texto que representa una consulta. callback: Una función de tipo (List<String>) → Unit que se llamará con los datos obtenidos.

② Dentro de getDataBBDD, se utiliza Thread.sleep(3000) para simular un retraso en la operación de acceso a datos. Esto representa el tiempo que podría tardar una operación de red o base de datos.

③ Se crea una lista de cadenas data con 3 elementos. Cada elemento es una cadena que contiene "Datos obtenidos a partir de $sql", simulando los datos que podrían ser recuperados de una base de datos.

④ Después de simular la operación de acceso a datos, se llama a la función callback pasando la lista data. Esta es la función que maneja los datos una vez que la operación asíncrona se ha completado.

⑤ En la función main, se inicia la aplicación y se simula un retraso de 1000 milisegundos (1 segundo) para esperar antes de realizar la petición de datos.

⑥ getDataBBDD se llama con una cadena de texto como parámetro. El segundo parámetro es una lambda que actúa como el callback. Esta lambda recibe los datos y los imprime en la consola.

La ejecución:

```
Comenzamos nuestra aplicación..
Ahora procedemos a petición de datos...
Se procede a petición de datos.....
Se han devuelto los datos
Los datos obtenidos son:
Datos obtenidos a partir de name = santi
Datos obtenidos a partir de name = santi
Datos obtenidos a partir de name = santi
```

Llamada asíncrona.

Este tipo de llamadas, son más óptimas porque no requieren un bloqueo del hilo principal a la espera de que se contemple las operaciones oportunas. Imaginar una app que necesite hacer un

acceso a datos y mientras la aplicación puede realizar otro tipo de tareas, debe bloquearse a la espera de los datos. Esto es muy común en aplicaciones REST. Para adaptar nuestro ejemplo a una llamada asíncrona, lo que haremos es dentro del método getdataBBDD, la simulación de los datos lo haremos dentro de otro hilo de ejecución y por tanto, tendremos dos hilos el principal y el de acceso a los datos.

e11.2.kt

```
fun getDataBBDD (sql : String, callback: (List<String>)-> Unit){
    println ("Se procede a petición de datos.....")

    Thread{
        Thread.sleep(5000)
        val data : List<String> = List<String>(3){"Datos obtenidos a partir de $sql - Elemento ${it + 1}"}
        println ("Se han devuelto los datos asíncronos.")
        callback (data)
    }.start()
}

fun main(){
    println("Comenzamos nuestra aplicación...")
    Thread.sleep(1000)
    println("Ahora procedemos a la petición de datos de manera asíncrona...")
    getDataBBDD ("name = santi") { data ->
        println ("Los datos obtenidos son: ")
        data.forEach { println (it) }
        println (" A que esto es lo último que veis?")
    }
    println("-------")
    println("Lo normal sería que esto se ejecutara al final, pero fijaros lo último que se ejecuta.")
}
```

La ejecución:

```
Comenzamos nuestra aplicación...
Ahora procedemos a la petición de datos de manera asíncrona...
Se procede a petición de datos.....
--------
Lo normal sería que esto se ejecutara al final, pero fijaros lo último que se ejecuta.
Se han devuelto los datos asíncronos. A que esto es lo último que veis?
Los datos obtenidos son:
Datos obtenidos a partir de name = santi - Elemento 1
Datos obtenidos a partir de name = santi - Elemento 2
Datos obtenidos a partir de name = santi - Elemento 3
A que esto es lo último que veis?
```

Ejemplo adaptado a corrutinas.

Intentar seguir el flujo de ejecución. De todas formas, es sencillo ya que la operación asíncrona se ha simulado en otro hilo principal, por lo que al haber dos hilos y el de la ejecución no es bloqueante del principal, éste acaba sus impresiones en pantalla, antes de que lo haga el método que ejecuta la operación asíncrona. Nosotros en Android, no trabajaremos con hilos de manera implícita pero si con las Corrutinas que en la asignatura de PSP la daremos. Lo que si podemos adelantar, es que una corrutina es una operación asíncrona, que se ejecuta dentro de un hilo de ejecución y no tiene por qué ser el mismo de la UI. De hecho, existen diferentes Scope, (escenarios) en los que una corrutina la podemos asociar a uno u otro hilo de ejecución. También decimos que si lanzamos varias corrutinas dentro del mismo hilo, éstas se ejecutarán en orden secuencia. **Más adelante lo veremos....**

Más adelante entenderos cosas como esta:

e11.3.kt

```
1  import kotlinx.coroutines.*
2
3  fun getDataBBDD(sql: String, callback: (List<String>) -> Unit) {
4      GlobalScope.launch(Dispatchers.IO) {                          ①
5          println("Se procede a petición de datos.....")
6          delay(3000)
7
8          val data: List<String> = List(3) { "Datos obtenidos a partir de $sql" }
9          println("Se han devuelto los datos asíncronos.")
10
11         // Cambia al hilo principal para ejecutar el callback
12         withContext(Dispatchers.Main) {                           ②
13             callback(data)
14         }
15     }
16 }
17
18 fun main() {
19     println("Comenzamos nuestra aplicación...")
20
21     // Llama a getDataBBDD y proporciona un callback para manejar los datos
22     getDataBBDD("name = santi") { data ->
23         println("Los datos obtenidos son: ")
24         data.forEach { println(it) }
25         println("A que esto es lo último que veis?")
26     }
27
28     println("-------")
29     println("Lo normal sería que esto se ejecutara al final, pero fijaros lo último
   que se ejecuta.")
30
31     Thread.sleep(5000)                                            ③
```

```
32 }
```

① Inicia una nueva corutina en el scope global y asociado al hilo de E/S. Que sea global, significa que el ciclo de vida de esta corrutina es global, es decir, mientras dure la ejecución de la aplicación. Si cerráramos la aplicación antes de que finalizara la corrutina, ésta también lo haría (lógico). Existen otros scope vinculados a Activitys o Fragments **(CoroutineScope)**, que tendría más sentido, pero recordemos que esto es kotlin y no Android studio.

② Cambia al hilo principal para ejecutar el callback. La UI siempre debe actualizarse dentro del hilo principal.

③ En una app de Android, **NO** sería necesario poner ese sleep, pero en nuestro ejemplo de consola si. Hay que dejar que la corrutina se ejecute antes de que pueda finalizar nuestra aplicación de ejemplo por consola.

ACTIVIDADES

1. **Ejercicio Callback Síncrono**:
    1. **Definir la Función Principal `procesarNombres`**:
        - Crea una función llamada `procesarNombres` que tome dos parámetros:
            - `nombres`: una lista de cadenas de texto que representan nombres de personas.
            - `callback`: una función de tipo `(String) -> Unit` que se invoca con cada nombre procesado.
        - Dentro de `procesarNombres`, recorre la lista de nombres y aplica la función `callback` a cada nombre de manera secuencial.
    2. **Definir la Función de Callback**:
        - Crea una función de tipo `callback` que reciba un nombre y lo imprima en consola con un mensaje que indique que el nombre ha sido procesado.
    3. **Implementar Ejemplo de Uso**:
        - Crea una lista de nombres.
        - Llama a `procesarNombres`, pasando la lista y la función de callback.
        - Muestra en consola cómo cada nombre es procesado por la función de callback.
    4. **Consideraciones**:
        - Asegúrate de que la función de callback se ejecute de manera síncrona y se aplique a cada elemento de la lista.
        - Verifica que los nombres sean procesados en el orden en que aparecen en la lista.
2. **Ejercicio Callback Asíncrono**:
    1. **Definir la Función Principal `descargarContenido`**:
        - Crea una función llamada `descargarContenido` que tome dos parámetros:
            - `url`: una cadena de texto que representa una URL de la cual se desea descargar contenido.
            - `callback`: una función de tipo `(String) -> Unit` que se invoca con el contenido descargado.
        - Dentro de `descargarContenido`, utiliza una técnica de ejecución asíncrona (como corutinas, hilos o `ExecutorService`) para simular la descarga del contenido de la URL.
        - Después de simular la descarga (por ejemplo, usando un retraso), llama a la función `callback` pasando el contenido descargado.
    2. **Definir la Función de Callback**:
        - Crea una función de tipo `callback` que reciba una cadena de texto (el contenido descargado) y lo imprima en consola con un mensaje que indique que el contenido ha sido descargado.
    3. **Implementar Ejemplo de Uso**:
        - Crea una llamada a `descargarContenido` en la función `main`, proporcionando una URL

simulada y una lambda como callback.

- Muestra en consola cómo se maneja el contenido una vez que el callback es invocado.

4. **Consideraciones**:
   - Asegúrate de que el hilo principal no esté bloqueado mientras se simula la descarga del contenido.
   - Verifica que el callback se ejecute después de que la operación asíncrona haya finalizado y que el flujo del programa sea fluido.