Programación Multimedia y Dispositivos Móviles

UD 1. Introducción a Kotlin

Javier Carrasco

Curso 2024 / 2025

Introducción a Kotlin

1. Introducción a Kotlin

Antes de adentrarse en el desarrollo de aplicaciones móviles, se presentará el lenguaje de programación que se utilizará durante todo el proceso. Durante este primer capítulo se realizará la toma de contacto al lenguaje de programación **Kotlin**. Como es posible que vayas a enfrentarte a dos retos nuevos de manera simultánea, aprender un nuevo lenguaje y programar aplicaciones móviles, se tratará de avanzar a ritmo lento y trabajando los conceptos básicos de Kotlin, todo con el máximo detalle posible.

1.1. Conceptos básicos de Kotlin

Cuando comencé con estos apuntes, **Kotlin** se encontraba en la versión **1.3**. y tímidamente trataba de abrirse camino luchando contra Java. Se comenzará con concepto básicos como la declaración de variables, creación de clases y funciones y conceptos básicos del lenguaje. Otros aspectos como la herencia de clases, entre otros, son similares a Java, de hecho, si ya conoces Java, te será muy fácil aprender **Kotlin**.

Para los ejemplos que se verán a continuación, puedes utilizar *Kotlin Playground*[1] para probar el código sin necesidad de crear proyectos, de momento. *IntelliJ IDEA*[2] también es un buen IDE para realizar pruebas. Los ejemplos que se verán a continuación están desarrollados con ***IntelliJ IDEA Community Edition***.

1.1.1. Primer programa en Kotlin

Para crear un nuevo proyecto en Kotlin con *IntelliJ* bastará con seleccionar la opción ***New Projet*** desde la ventana principal y seleccionar Kotlin como lenguaje.

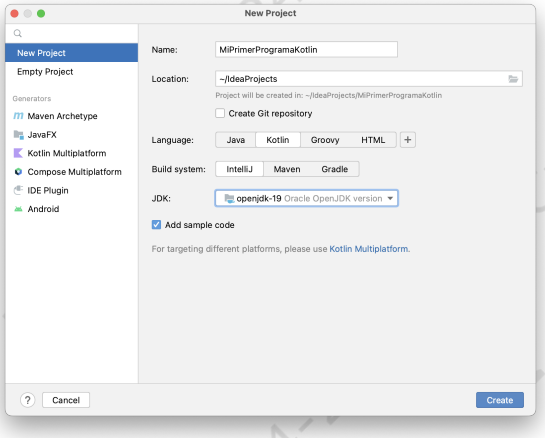

Figura 1

1 Kotlin Playground (https://play.kotlinlang.org)
2 IntelliJ IDEA (https://www.jetbrains.com/es-es/idea/)

Una vez creado, ya tendrás disponible tu primera clase Kotlin, llamada `Main.kt`, en la puedes encontrar el siguiente código.

```kotlin
fun main(args: Array<String>) {
    println("Hello World!")
}
```

Al igual que ocurre con Java, el punto de entrada inicial al programa será la función *main*.

1.1.2. Expresividad

Kotlin permite reducir código repetitivo, es capaz de cubrir la mayor parte de los patrones por defecto. Observa a continuación la siguiente clase Java.

```java
public class Artista {
    private long id;
    private String nombre;
    private String url;
    private String mbid;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getMbid() {
        return mbid;
    }

    public void setMbid(String mbid) {
        this.mbid = mbid;
```

```
    }

    @Override
    public String toString() {
        return "Artista{id=" + id + ", nombre='" + nombre + '\'' +
                ", url='" + url + '\'' + ", mbid='" + mbid + '\'}';
    }
}
```

En **Kotlin** se obtendría la clase equivalente con las siguientes líneas.

```
data class Artista(
    var id: Long,
    var nombre: String,
    var url: String,
    var mbid: String
)
```

1.1.3. Variables

El uso, y declaración de variables en **Kotlin** tiene ciertas diferencias con respecto al uso de variables en Java.

```
val i: Int = 42
```

- La declaración de variables comienza por **var** o **val**, según sea la declaración de una ***var**iable* o un *final("**val**ue")*, es decir, mutable o inmutable (se puede o no modificar).
- El tipo de dato se especificará después del nombre, separado por dos puntos (**:**).
- **Kotlin** tiene los tipos de datos inferidos, es decir, no es necesario especificar el tipo de la variable si se asigna un valor en la declaración, no siendo ambiguo para el compilador.
- **Kotlin** requiere la inicialización de las variables en el momento de la instanciación, para evitar esta obligatoriedad, con variables que se crean al inicio, pero que se inicializarán más adelante, se utiliza la palabra reservada `lateinit` delante de *var* y/o *val*.

Se debe tener en cuenta que en **Kotlin** todo es un objeto. No dispone de tipos primitivos como Java. Pero se sigue disponiendo de los siguientes **tipos básicos**.

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

El nombre y funcionamiento de estos tipos básicos es similar a su uso en Java, pero con algunas diferencias a tener en cuenta.

- No hay conversiones automáticas entre tipos numéricos. No se puede asignar un *Int* a un *Double*, se debe utilizar para ello los métodos de conversión existentes.

```
val i: Int = 23
val d: Double = i.toDouble()
```

- Los caracteres (*Char*) no pueden se utilizados directamente como números, es necesario hacer uso, nuevamente, de los métodos de conversión existentes.

```
val c: Char = 'c'
val i: Int = c.toInt()
```

- Las operaciones a nivel de bit (*bitwise*[3]) son ligeramente diferentes. En Android se utilizan muy a menudo a modo de bandera (*flag*).

```
// En Java
int bitwiseOr = FLAG1 | FLAG2;
int bitwiseAnd = FLAG1 & FLAG2;

// En Kotlin
val bitwiseOr = FLAG1 or FLAG2
val bitwiseAnd = FLAG1 and FLAG2
```

- Los comentarios ayudan a dar pistas, en **Kotlin** es una práctica común el omitir el tipo de dato de las variables, de ahí el uso de comentarios para ayudar.

```
val i = 12 // An Int
val iHex = 0x0f // An Int from hexadecimal literal
val l = 3L // A Long
val d = 3.5 // A Double
val f = 3.5F // A Float
```

- Las cadenas (*Strings*) pueden se accedidas como si de un *array* se tratase y, pueden ser iteradas.

```
val s = "Ejemplo"
val c = s[3] // Carácter 'm'

// Iteración sobre un String
val s2 = "Example"
for (c2 in s2) {
    print(c2)
}
```

De vuelta a la declaración de variables en **Kotlin**, estas se pueden declarar fácilmente como **mutable** (*var*) o **inmutable** (*val*). Es como utilizar variables *final* en Java. Sin embargo, la inmutabilidad es un concepto fundamental en **Kotlin**, como ocurre con otros lenguajes modernos.

3 *Bitwise* (https://kotlinlang.org/docs/reference/basic-types.html#operations)

Un objeto **inmutable** es aquel que no puede cambiar su estado después de realizar su instanciación. Si necesitas modificar un objeto de este tipo, debes crear uno nuevo. El objetivo de la inmutabilidad es construir *software* más robusto. En Java, la mayoría de los objetos son mutables, lo que significa que se puede modificar la parte del código a la que se accede, esto puede afectar al resto de la aplicación.

Los objetos inmutables también son *thread-safe* (hilos seguros) por definición. Al ser constante su valor, no se necesita implementar ningún mecanismo de sincronización, ya que todos los hilos cogerán siempre el mismo objeto.

La codificación en **Kotlin** cambia si se quiere hacer uso de la inmutabilidad, la clave es: utilizar *val* tanto como sea posible. Existirán situaciones (especialmente en Android, donde un constructor puede ser llamado por el sistema desde varias clases) donde la inmutabilidad puede ser difícil de implementar, pero no imposible.

1.1.4. Null Safety

En Java, el código generalmente es defensivo, por lo que se debe comprobar en todo momento que no se produzca un *null* y prevenir *NullPointerException*. **Kotlin**, en cambio, es *null-safety*, lo que significa que se puede definir si un objeto puede o no ser *null* utilizando para ello el operador `?`. Fíjate en los siguientes ejemplos.

```kotlin
fun main(args: Array<String>) {
    // No compilaría, Artista no puede ser nulo.
    var notNullArtista: Artista = null

    // Artista puede ser nulo.
    val artista: Artista? = null

    // No compilará, artista podría ser nulo.
    artista.toString()

    // Mostrará por pantalla artista si es distinto de nulo.
    artista?.toString()

    // No necesitaríamos utilizar el operador ? si previamente
    // comprobamos si es nulo.
    if (artista != null) {
        artista.toString()
    }
    // Esta operación la utilizaremos si estamos completamente seguros
    // que no será nulo. En caso contrario se producirá una excepción.
    artista!!.toString()

    // También podríamos utilizar el operador Elvis (?:) para dar
    // una alternativa en caso que el objeto sea nulo.
    val nombre = artista?.nombre ?: "vacío"
}
```

1.2. Clases

Como se ha visto al inicio del capítulo, las **clases** en **Kotlin** se reducen a la mínima expresión, lo que permite un ahorro de código importante. A continuación se verá con más detalle algunos aspectos relacionados con ellas.

Para **declarar** una clase bastará con utilizar la palabra reservada `class` seguida por el nombre de clase, por ejemplo, `class Persona`. Por defecto en **Kotlin** las clases son públicas, sin necesidad de especificarse el ámbito. Tampoco se han utilizado llaves `{}`, aunque también se pueden utilizar en la declaración.

En **Kotlin**, las clases no tienen campos, sino **propiedades** (*properties*), que viene a ser la combinación del campo + *getter* + *setter*. El siguiente código muestra como añadir dos propiedades a la clase *Persona*.

```
class Persona {
    var nombre: String = ""
    var apellido: String = ""
}
```

Pero, se pueden sobrecargar los métodos *get* y *set* de las propiedades si fuese necesario. En el siguiente ejemplo puedes ver como se haría. Fíjate que se han sobrecargado de diferente manera.

```
class Persona {
   var nombre: String = "Vacío"
       set(value) {
           field = if (value.isEmpty()) "Sin nombre" else value
       }
       get() = "Nombre: $field"

   var apellido: String = "Vacío"
       get() {
           return "Apellido: " + field.uppercase()
       }
       set(value) {
           field = if (value.isEmpty()) "Sin apellido" else value
       }
}

fun main() {
    val persona = Persona()

    persona.nombre = ""
    println(persona.nombre)

    persona.nombre = "Javier"
    println(persona.nombre)
    println(persona.apellido)
```

```
26
27      persona.apellido = "Carrasco"
28      println(persona.apellido)
29  }
```

La salida que se obtendría por consola al ejecutar el código sería la que puede verse a continuación.

```
Nombre: Sin nombre
Nombre: Javier
Apellido: VACÍO
Apellido: CARRASCO
```

Las clases en **Kotlin** también disponen de **constructores**, concretamente, el constructor primario y constructores secundarios. Ahora se modificará la clase `Persona` para añadir un constructor primario, que está representado por la palabra reservada `init`.

```
1   class Persona(nombre: String, apellido: String) {
2       var nombre: String = "Vacío"
3           set(value) {
4               field = if (value.isEmpty()) "Sin nombre" else value
5           }
6           get() = "Nombre: $field"
7
8       var apellido: String = "Vacío"
9           get() {
10              return "Apellido: " + field.uppercase()
11          }
12          set(value) {
13              field = if (value.isEmpty()) "Sin apellido" else value
14          }
15
16      // Constructor primario
17      init {
18          this.nombre = nombre
19          this.apellido = apellido
20      }
21  }
22
23  fun main() {
24      val persona = Persona("Javier", "Carrasco")
25      println(persona.nombre)
26      println(persona.apellido)
27
28      persona.nombre = ""
29      println(persona.nombre)
30
31      persona.nombre = "Nacho"
32      println(persona.nombre)
33      println(persona.apellido)
```

```
34
35      persona.apellido = "Cabanes"
36      println(persona.apellido)
37  }
```

Fíjate que los parámetros están indicados en la propia declaración de la clase (constructor en línea), no en el bloque `init`, donde se asignan los valores de los parámetros a las propiedades de la clase. Se deberá obtener por consola la siguiente salida.

```
Nombre: Javier
Apellido: CARRASCO
Nombre: Sin nombre
Nombre: Nacho
Apellido: CARRASCO
Apellido: CABANES
```

Se modifica nuevamente la clase `Persona` con dos constructores secundarios, al igual que en Java, se debe cambiar el orden de los tipos de datos de los parámetros o su número, para que el compilador no los considere iguales. También es muy importante la referencia al `init`, si este existe, o al constructor en línea, haciendo uso de la palabra reservada `this`, vendría a ser `super()` en Java.

```
1   class Persona(nombre: String, apellido: String) {
2       var nombre: String = "Vacío"
3           set(value) {
4               field = if (value.isEmpty()) "Sin nombre" else value
5           }
6           get() = "Nombre: $field"
7
8       var apellido: String = "Vacío"
9           get() {
10              return "Apellido: " + field.uppercase()
11          }
12          set(value) {
13              field = if (value.isEmpty()) "Sin apellido" else value
14          }
15
16      var edad: Int = 0
17          set(value) {
18              field = if (value < 0) 0 else value
19          }
20
21      var anyo: Int = 0
22
23      // Constructor primario
24      init {
25          this.nombre = nombre
26          this.apellido = apellido
27      }
28
```

```
    // Constructor secundario
    constructor(nombre: String, apellido: String, edad: Int) : this(nombre, apellido) {
        this.nombre = nombre
        this.apellido = apellido
        this.edad = edad
    }

    // Otro constructor secundario
    constructor(nombre: String, apellido: String, edad: Int, anyo: Int) :
            this(nombre, apellido) {
        this.nombre = nombre
        this.apellido = apellido
        this.edad = edad
        this.anyo = anyo
    }
}

fun main() {
    val persona1 = Persona("Nacho", "Cabanes")
    val persona2 = Persona("Javier", "Carrasco", 42)
    val persona3 = Persona("Pedro", "Pérez", 53, 1966)

    println("Persona 1")
    println(persona1.nombre)
    println(persona1.apellido)
    println("Edad: " + persona1.edad + "\n")

    println("Persona 2")
    println(persona2.nombre)
    println(persona2.apellido)
    println("Edad: " + persona2.edad + "\n")

    println("Persona 3")
    println(persona3.nombre)
    println(persona3.apellido)
    println("Edad: " + persona3.edad)
    println("Año: " + persona3.anyo)
}
```

El resultado tras esta modificación del código sería el que puedes ver a continuación.

```
Persona 1
Nombre: Nacho
Apellido: CABANES
Edad: 0

Persona 2
Nombre: Javier
Apellido: CARRASCO
Edad: 42
```

```
Persona 3
Nombre: Pedro
Apellido: PÉREZ
Edad: 53
Año: 1966
```

Ahora, ya se pueden añadir **funciones**, o **métodos**, a las clases en **Kotlin**. Por ejemplo, una función, o método, con una estructura similar a Java podría ser la siguiente.

```
class Persona (nombre: String, apellido: String) {
    ...
    fun getNombreCompleto() : String {
        return "$nombre $apellido"
    }
    ...
}
```

El tipo *String* indicado tras los dos puntos después del nombre indica el tipo de valor devuelto por el método. También es posible hacer una versión más compacta cuando se devuelve el valor directamente.

```
class Persona (nombre: String, apellido: String) {
    ...
    fun getNombreCompleto() = "$nombre $apellido"
    ...
}
```

La forma de utilizar los métodos, o funciones, en Kotlin sería exactamente igual a cómo se hace en Java. Observa como se llamaría al método desde el objeto *persona2*.

```
println(persona2.getNombreCompleto())
```

Para establecer la **herencia** en las clases **Kotlin**, debe tenerse en cuenta que es cerrado por defecto, esto quiere decir que no se puede extender de otras clases. Siguiendo el ejemplo de la clase `Persona`, no se podría crear una clase que heredase de ella. Para poder extender, heredar, se debe añadir la palabra reservada ***open*** a la definición de la clase `Persona`. Observa como quedaría la clase `Conductor` que heredará de `Persona`.

En primer lugar deberá modificarse la definición de la clase `Persona`.

```
open class Persona (nombre: String, apellido: String) {
    ...
}
```

Ahora define la clase `Conductor` que extiende de `Persona`, se le añadirá una propiedad `permiso` para indicar el tipo de carné de conducir que tiene.

```
class Conductor(nombre: String, apellido: String, permiso: String)
    : Persona(nombre, apellido) {

```

```
    var permiso: String = ""
        get() {
            return "Permiso: " + field.uppercase()
        }
        set(value) {
            field = if (value.isEmpty()) "No tiene permiso" else value
        }

    init {
        this.permiso = permiso
    }
}
```

Fíjate que se ha añadido `:Persona(nombre, apellido)` a la definición de la clase `Conductor`, esto indica que el constructor de la clase padre se debe utilizar, viene a ser el `super()` que se utiliza en Java.

Es necesario mencionar los **modificadores de visibilidad** disponibles en **Kotlin**. Estos modificadores permiten restringir el acceso, y se pueden aplicar a clases, interfaces, objetos, métodos y propiedades. **Kotlin** dispone de cuatro modificadores.

- ***Public***, modificador por defecto, se puede aplicar a cualquier clase, interfaz, objeto, método o propiedad, permitiendo que estos puedan ser accedidos desde cualquier parte.
- ***Private***, con este modificador una función de nivel superior, interfaz, o clase que se declare como `private` puede ser accedida únicamente desde el archivo que la contiene.

  Si una función o propiedad se declara como `private` dentro de una clase, objeto, o interfaz, solamente será visible para otros miembros de la misma clase, objeto, o interfaz.

  ```
  class Persona (){
      private var edad: Int = 0
  }
  ```

- ***Protected***, sólo se podrá aplicar a funciones y propiedades dentro de una clase, objeto o interfaz. Estas propiedades o funciones solo serán accesibles por la clase que las define y sus subclases.
- ***Internal***, en proyectos con módulo *Gradle* o *Maven*, las clases, objetos, interfaces o funciones declaradas con el modificador `internal`, solo serán accedidos desde ese módulo.

  ```
  internal class Persona() {
      val edad : Int = 0
  }
  ```

Existen otros tipos de clases que pueden resultar útiles, como las que se muestran a continuación.

1.2.1. Data classes

Las clases de datos, o ***data class***, utilizan del modificador ***data*** para identificarlas. El objetivo de este tipo de clases es almacenar únicamente información, no tienen métodos que manipulen los datos.

```
data class Artista(
    var id: Long,
    var nombre: String,
    var url: String,
    var mbid: String
)
```

Este tipo de clase de datos auto-genera todos los *getters* y *setters* de las propiedades y otros métodos útiles como *toString()*. También se generan los métodos *equals()* y *hashCode()* automáticamente, evitando así cometer alguna equivocación en su implementación, lo que podría resultar peligroso.

Una propiedad que puede resultar interesante de las *data classes* es la que se conoce como *"**desestructuración**"*. La desestructuración permite crear múltiples variables a partir de una *data class*, siempre en el mismo orden en el que está creadas las propiedades de la clase. Observa el siguiente ejemplo.

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

fun main() {
    val p1 = Person("Javier", "Carrasco", 45)
    println(p1)
}
```

El resultado que se obtendría sería la siguiente línea.

```
Person(name=Javier, surname=Carrasco, age=45)
```

La desestructuración se podría realizar de varias maneras.

```
val (_, _, edad) = p1 // Sólo se obtiene la edad.
val (nombre, apellido, _) = p1 // Se obtiene nombre y apellido.
val (nombre, apellido, edad) = p1 // Se obtienen todas las propiedades.
```

El uso de estas variables sería el típico, no tienen nada especial.

```
println(nombre) // Salida: Javier
```

1.2.2. Sealed classes

Este tipo de clases pueden restringir el número de hijos que se pueden crear a partir de ella. Al utilizar el modificador ***sealed*** en la clase padre, no podrán crearse más hijos fuera del proyecto.

```
open class Vehiculo(var nRuedas: Int)
```

```
data class Motocicleta(var ruedas: Int = 2) : Vehiculo(ruedas)
data class Turismo(var ruedas: Int = 4, var puertas: Int = 2) : Vehiculo(ruedas)
```

Si se crea esta estructura, a la hora de comprobar el tipo de un objeto, podrá contener más variaciones, es decir, otros hijos que no estén controlados. Por ejemplo, se crea el siguiente método para comprobar el tipo de vehículo.

```
fun tipoVehiculo(vehiculo: Vehiculo): String {
    return when (vehiculo) {
        is Motocicleta -> "Es del tipo Motocicleta"
        is Turismo -> "Es del tipo Turismo"
        else -> throw IllegalArgumentException("Tipo desconocido")
    }
}
```

Al ser la clase padre de tipo *open*, el IDE obliga a añadir la línea del *else*, para así poder controlar todas las posibilidades. Ahora bien, si cambiar el modificador de la clase padre…

```
sealed class Vehiculo(var nRuedas: Int)
```

… verás que el propio IDE te indicará que ya no es necesaria la línea *else* en el código. El modificador *sealed* indica al compilador que no debe hacer más comprobaciones, acotando así las verificaciones que deben hacerse.

1.2.3. Clase Pair

La clase `Pair` en **Kotlin** se utiliza para realizar representaciones genéricas de pares (dos valores), siendo estos de cualquier tipo de dato o incluso clases. A simple vista puede resultar poco práctica, pero puede resultar útil en determinadas situaciones. Observa a continuación varias formas de instanciar pares.

```
val parUno = Pair("Hola", "Mundo")
val parDos = Pair("Adiós amigos", 150)
val (usuario, contrasenya) = Pair("javier", "kotlin")
```

Fíjate que no es necesario que los pares sean del mismo tipo, pueden combinarse según convenga en cada momento. El acceso a los elementos creados mediante la clase *Pair* sería como se muestra a continuación.

```
println("Par UNO-1: ${parUno.first} || Par UNO-2: ${parUno.second}")
println("Par DOS-1: ${parDos.first} || Par DOS-2: ${parDos.second}")
println("Usu: $usuario - Pass: $contrasenya")
```

El resultado que se obtendría de este código se puede ver a continuación.

```
Par UNO-1: Hola || Par UNO-2: Mundo
Par DOS-1: Adiós amigos || Par DOS-2: 150
Usu: javier - Pass: kotlin
```

1.3. Funciones

Como ya habrás observado durante los capítulos anteriores, para declarar una función se utiliza la palabra reservada `fun`. Fíjate en la siguiente función básica.

```
fun sayHello() : String{
    return "Hi!!"
}
```

Ahora, la misma función, pero en formato abreviado.

```
fun sayHello() : String = "Hi!!"
```

Incluso se podría reducir todavía más, el tipo que devuelve la función puede ser omitido.

```
fun sayHello() = "Hi!!"
```

Observa ahora como añadir parámetros a las funciones.

```
fun sayHello(name: String, surname: String): String {
    return "Hello $name $surname"
}
```

1.3.1. Funciones de extensión

Gracias a las funciones de extensión, es posible añadir nuevas funcionalidades a cualquier clase sin necesidad de editar su código. Por ejemplo, se puede añadir un nuevo método a la clase *Fragment* existente para mostrar un *toast*.

```
fun Fragment.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT){
    Toast.makeText(getActivity(), message, duration).show()
}
```

Se podría utilizar de la siguiente forma con una variable de tipo *Fragment*.

```
fragment.toast("Hello world!!")
```

1.4. Colecciones

A diferencia de muchos otros lenguajes de programación, **Kotlin** distingue entre colecciones mutables e inmutables (*lists*, *maps*, *sets*, etc). Esta propiedad puede resultar útil para eliminar fallos y crear un mejor diseño. No se entrará en detalle sobre el funcionamiento interno de las colecciones, ya que funcionan exactamente igual que en Java, únicamente se verá como deben declararse en Kotlin y su uso básico.

1.4.1. Listas en Kotlin

```kotlin
// Lista inmutable, una vez creada no se puede modificar (read-only).
val listOfItems = listOf("Item 1", "Item 2", "Item 3")

println(listOfItems) // Salida: [Item 1, Item 2, Item 3]
```

Observa como se crean **listas** que se pueden modificar, mutables.

```kotlin
val listOfItems = mutableListOf<String>()

listOfItems.add("Item 1")
listOfItems.add("Item 2")
listOfItems.add("Item 3")
println(listOfItems) // Salida: [Item 1, Item 2, Item 3]

listOfItems.removeAt(1)
println(listOfItems) // Salida: [Item 1, Item 3]
```

Ahora, fíjate en los pasos que se deben seguir para recorrer una **lista**.

```kotlin
val listOfItems = listOf("Item 1", "Item 2", "Item 3")

println("ver. 1")
for (item in listOfItems){
    println(item)
}

println("ver. 2")
listOfItems.forEach {
    println(it)
}

println("ver. 3")
val listOfItemsIterator = listOfItems.iterator()

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

La salida que se obtendría para cada una de las versiones sería la misma.

```
Item 1
Item 2
Item 3
```

1.4.2. Mapas en Kotlin

```kotlin
// Mapa inmutable, una vez creada no se puede modificar (read-only).
val mapOfItems = mapOf(Pair(1,"Item 1"), Pair(2,"Item 2"), Pair(3,"Item 3"))
```

```
3  println(mapOfItems) // Salida: {1=Item 1, 2=Item 2, 3=Item 3}
```

Fíjate como crear **mapas** que se puedan modificar según convenga.

```
1  val mapOfItems = mutableMapOf<Int, String>()
2
3  mapOfItems[0] = "Item 0" // mapOfItems.put(0, "Item 0")
4  mapOfItems[1] = "Item 1"
5  mapOfItems[2] = "Item 2"
6  println(mapOfItems) // Salida: {0=Item 0, 1=Item 1, 2=Item 2}
7
8  mapOfItems.remove(1)
9  println(mapOfItems) // Salida: {0=Item 0, 2=Item 2}
```

Ahora, observa los pasos a seguir para recorrer un **mapa**.

```
1  val mapOfItems = mapOf(Pair(1,"Item 1"), Pair(2,"Item 2"), Pair(3,"Item 3"))
2
3  println("ver. 1")
4  for (item in mapOfItems){
5      println(item)
6  }
7
8  println("ver. 2")
9  mapOfItems.forEach {
10     println(it)
11 }
12
13 println("ver. 3")
14 val mapOfItemsIterator = mapOfItems.iterator()
15
16 while (mapOfItemsIterator.hasNext()){
17     println(mapOfItemsIterator.next())
18 }
```

La salida obtenida para cada una de las versiones sería la misma.

```
1=Item 1
2=Item 2
3=Item 3
```

1.4.3. Conjuntos en Kotlin

```
1  // Conjunto inmutable, una vez creada no se puede modificar (read-only).
2  val setOfItems = setOf(1, 2, 3, 4)
3
4  println(setOfItems) // Salida: [1, 2, 3, 4]
```

Observa como se crearían **conjuntos** que se puedan modificar.

```
1  val setOfItems = mutableSetOf<Int>()
```

```
setOfItems.add(1)
setOfItems.add(2)
setOfItems.add(3)
println(setOfItems) // Salida: [1, 2, 3]

setOfItems.remove(2) // Elimina por elemento, no por índice.
println(setOfItems) // Salida: [1, 3]
```

Ahora, fíjate en los pasos que se deben seguir para recorrer un **conjunto**.

```
val setOfItems = setOf(1, 2, 3, 4)

println("ver. 1")
for (item in setOfItems){
    println(item)
}

println("ver. 2")
setOfItems.forEach {
    println(it)
}

println("ver. 3")
val setOfItemsIterator = setOfItems.iterator()

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

La salida obtenida para cada una de las versiones sería la misma, `1 2 3 4`.

1.5. Enumerados

Los enumerados, o *Enum*, son un tipo de datos especial que permite definir valores preestablecidos o constantes. Este es un tipo de datos utilizado por muchos otros lenguajes de programación, como Java, pero que también puede explotarse en **Kotlin** de forma similar.

```
enum class DiasSemana {
    LUNES,
    MARTES,
    MIERCOLES,
    JUEVES,
    VIERNES,
    SABADO,
    DOMINGO
}
```

Para hacer uso de una de las constantes enumeradas, bastaría con hacer referencia a ella como se muestra en la siguiente línea.

```
println(DiasSemana.LUNES)
```

También pueden utilizarse como valores que puedan ser evaluados, como puedes ver en el siguiente ejemplo.

```
val dia = DiasSemana.DOMINGO
val estado = when (dia) {
    DiasSemana.SABADO -> "Fiesta"
    DiasSemana.DOMINGO -> "Descanso dominical"
    else -> "Trabajo"
}

println("Variable estado: $estado \n")
```

El resultado que obtendrías de este código sería el que se muestra a continuación.

```
Variable estado: Descanso dominical
```

Al igual que en Java, en **Kotlin** también se pueden tener datos asociados en los `enum` como puedes ver en el siguiente ejemplo.

```
enum class DiasSemana(val numero: Int, val estado: String) {
    LUNES(1, "Trabajando"),
    MARTES(2, "Trabajando"),
    MIERCOLES(3, "Trabajando"),
    JUEVES(4, "Trabajando"),
    VIERNES(5, "Trabajando"),
    SABADO(6, "Descanso"),
    DOMINGO(7, "Descanso")
}
```

Se podría recorrer la variable *enum* como se muestra a continuación, accediendo además a los datos asociados de cada elemento.

```
DiasSemana.values().forEach {
    println("${it.numero} - ${it.name} - ${it.estado}")
}
```

1.6. Objetos

Un objeto en **Kotlin** es una variable con una única implementación, y viene a representar los elementos *static* de Java, además, utilizan el patrón *singleton*, por tanto, solo existirá una instancia del objeto creado.

La referencia al objeto, y a sus propiedades o métodos, se realizará directamente mediante el nombre del propio objeto. Estos también son un buen lugar para declarar constantes, ya que únicamente se inicializan una vez.

Los objetos pueden resultar muy útiles cuando se necesite trabajar con instancias únicas de una estructura, como pueda ser una conexión a una base de datos, datos de usuario, etc.

Fíjate en el siguiente código con un ejemplo básico para ver la creación de un objeto en **Kotlin**, en este caso, con dos propiedades y un método totalmente accesibles.

```
object MiObjeto {
    val usuario: String = "Javier"
    val base_URL: String = "http://www.javiercarrasco.es/"

    fun funcionDeMiObjeto() {
        println("Has llamado a la función de un objeto.")
    }
}
```

Observa las siguientes líneas en un *main* para hacer uso del objeto creado.

```
val nombreUsuario = MiObjeto.usuario

println(nombreUsuario)
println(MiObjeto.funcionDeMiObjeto())
```

La salida que se obtendría por pantalla, por consola, sería la siguiente.

```
Javier
Has llamado a la función de un objeto.
```

Para comprobar que realmente se instancia una única vez, añade el método `init` al objeto, el cual hará las veces de constructor. Modifica también la instancia de la variable `usuario`.

```
object MiObjeto {
    lateinit var usuario: String
    val base_URL: String = "http://www.javiercarrasco.es/"

    init {
        usuario = "Javier"
        println("Método INIT, solo se llamará una vez.")
    }

    fun funcionDeMiObjeto() {
        println("Has llamado a la función de un objeto.")
    }
}
```

Si se hace uso nuevamente del objeto, verás que el constructor únicamente es llamado la primera vez. Debes tener en cuenta que cada vez que se escribe `MiObjeto`, se crea nuevamente, pero se mantienen los valores.

```
var nombreUsuario = MiObjeto.usuario

println(nombreUsuario)

MiObjeto.usuario = "Juan"
nombreUsuario = MiObjeto.usuario
```

```
println(nombreUsuario)
println(MiObjeto.funcionDeMiObjeto())
```

Observa como el nombre cambia tras la segunda llamada y no vuelve a su valor original la tercera vez que se llama como se establece en el *init*.

1.6.1. Companion objects

Los *companion objects*[4] pueden implementarse en cualquier clase, los cuales serán comunes a todas las instancias de dicha clase.

Los *companion objects* funcionan exactamente igual que los objetos, y sus miembros pueden ser accedidos a través del nombre de la clase que los contiene. Al ser de tipo *singleton*, son muy utilizados en el terreno de **Android**, como podrás comprobar de aquí en adelante.

Este tipo de objetos puede resultar muy útil cuando se necesite trabajar con instancias únicas de una estructura, como pueda ser una conexión a una base de datos, datos de usuario, etc.

Para definir un *companion object*, bastará con realizar la instancia dentro de la clase que debe contenerlo. Fíjate en la clase `Empleados` que viene a continuación.

```
class Empleados(nom: String, ape: String) {
    var idEmpleados: Int
    lateinit var nombre: String
    lateinit var apellido: String

    init {
        // Se ejecuta con cada instancia de Empleado.
        println("Init clase Empleado.")
        this.nombre = nom
        this.apellido = ape
        numEmpleados++
        idEmpleados = numEmpleados
    }

    companion object {
        // Se ejecutará una sola vez.
        init {
            println("Init Companion Object Empleado.")
        }
        var numEmpleados = 0
    }
}
```

Fíjate ahora en el uso que se haría de la clase `Empleados` desde un *main*.

```
val empleado1 = Empleados("Javier", "Carrasco")
val empleado2 = Empleados("Nacho", "Cabanes")
```

4 *Companion Objects* (https://kotlinlang.org/docs/reference/object-declarations.html)

```
println("Empleado 1: ${empleado1.nombre} ${empleado1.apellido}")
println("Empleado 2: ${empleado2.nombre} ${empleado2.apellido}")
println("Total empleados: ${Empleados.numEmpleados}")
```

Observa la salida obtenida, fíjate que la llamada la *init* del *companion object* solo se realiza la primera vez que se hace uso del *object*.

```
Init Companion Object Empleado.
Init clase Empleado.
Init clase Empleado.
Empleado 1: Javier Carrasco
Empleado 2: Nacho Cabanes
Total empleados: 2
```