

Tema 1 parte 1 - Lenguaje Kotlin I

Descargar estos apuntes [pdf](#) o [html](#)

Índice

1. [Introducción](#)
2. [Creando nuestro "Hola Mundo" en Kotlin](#)
 1. [Usando Gradle](#)
 2. [Usando Android Studio](#)
3. [El lenguaje Kotlin](#)
 1. [Control de Nulos](#)
 2. [Variables](#)
 1. [Interpolación de cadenas](#)
 3. [Control de flujo](#)
 1. [Condicionales como expresión](#)
 2. [Bucles con for y while](#)
 3. [Condicional múltiple con when](#)
4. [Funciones](#)
 1. [Retornando una tupla 'Pair<A, B>'](#)
 2. [Número de argumentos indefinido con 'varargs'](#)
 3. [Funciones de extensión](#)
5. [Excepciones](#)
6. [Clases e Interfaces](#)
 1. [Constructores](#)
 2. [Definir métodos y propiedades estáticas](#)
 3. [Herencia](#)
 4. [Interfaces](#)
 5. [Definiendo ValueObjects o DTO's con data class](#)
 6. [Clases y métodos parametrizados](#)
 7. [Scope functions](#)
 8. [Clases enumeradas con enum class](#)
 9. [Clases Selladas \(Sum Type Pattern\)](#)
 10. [Delegación de propiedades con `by`](#)

Introducción

Kotlin es un lenguaje de programación fuertemente tipado desarrollado por **JetBrains** en 2010 y que está influenciado por lenguajes como **C#, Scala, Groovy y Java**.


A partir de la actualización Kotlin 1.3.30, se incluyeron las mejoras para Kotlin/Native que permite compilar el código fuente de Kotlin en datos binarios independientes para diferentes sistemas operativos y arquitecturas de CPU, incluido IOS, Linux, Windows y Mac.

La mayoría de desarrollos con Kotlin, tienen cómo destino Android o la máquina virtual de java (JVM) y puedes encontrar 'ports' al lenguaje de Frameworks y librerías cómo:

- Desarrollo Nativo a Android NDK
- [Multiplataforma](#).
- Desarrollo web con [interactividad y transpilación a JavaScript](#) con [JS IR Compiler](#).
- [Desarrollo Backend](#) de acceso a datos y Microservicios con Frameworks como [Spring boot](#), [Quarkus](#), [Micronaut](#), [Ktor](#)
- Videojuegos con Frameworks como [LibKTX](#) o [KorGE](#)
- Ciencia de datos con plataformas como [Jupyter Notebooks](#) o [Datalore](#)

Si quieres saber más del lenguaje puedes consultar los siguientes enlaces:

- [Página oficial del lenguaje](#).
- [PlayGround Kotlin](#). (Probar Kotlin On-line)
- [Capacitación oficial de Kotlin para programadores](#).
- [Curso de Kotlin](#) (Lista de reproducción de DevExperto en Castellano)
- [Curso de Kotlin Básico](#) (Lista de reproducción oficial de MoureDev)

 **Nota:** Aunque en el momento de escribir estos apuntes la versión actual es la 1.9.0. Para la versión 2.0.0, el equipo de Kotlin está trabajando en [un nuevo compilador denominado K2](#) previsto liberar a lo largo de 2024 y que acelerará el tiempo de compilación drásticamente.

Creando nuestro "*Hola Mundo*" en Kotlin

Usando Gradle

Aquí puedes encontrar los pasos en el web oficial :

https://docs.gradle.org/current/samples/sample_building_kotlin_applications.html

Aunque podemos resumirlos aquí:

1. Creamos la carpeta o wrkspace, por ejemplo, `_proyecto_consola_kotlin` .
2. Abrimos una consola de comandos `cmd` en la carpeta creada.
3. Ejecutamos el comando `gradle init` y seguimos los pasos introduciendo las opciones del ejemplo a continuación.

```

~\proyecto_consola_kotlin>gradle init
Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use -

Select type of project to generate:
...
2: application
...
Enter selection (default: basic) [1..4] 2

Select implementation language:
...
4: Kotlin
...
Enter selection (default: Java) [1..6] 4

Split functionality across multiple subprojects?:
1: no - only one application project
2: yes - application and library projects
Enter selection (default: no - only one application project) [1..2] 1

Select build script DSL:
1: Groovy
2: Kotlin
Enter selection (default: Kotlin) [1..2] 2

Generate build using new APIs and behavior (some features may change in the next minor re

Project name (default: proyecto_consola_kotlin): proyecto_consola_kotlin

Source package (default: proyecto_consola_kotlin):

...
BUILD SUCCESSFUL in 2m 36s
2 actionable tasks: 2 executed

```

4. Para compilar y ejecutar nuestro proyecto ejecutaremos **gradlew run**

```

~\proyecto_consola_kotlin>gradlew run

> Task :app:run
Hello World!

BUILD SUCCESSFUL in 8s
2 actionable tasks: 2 executed

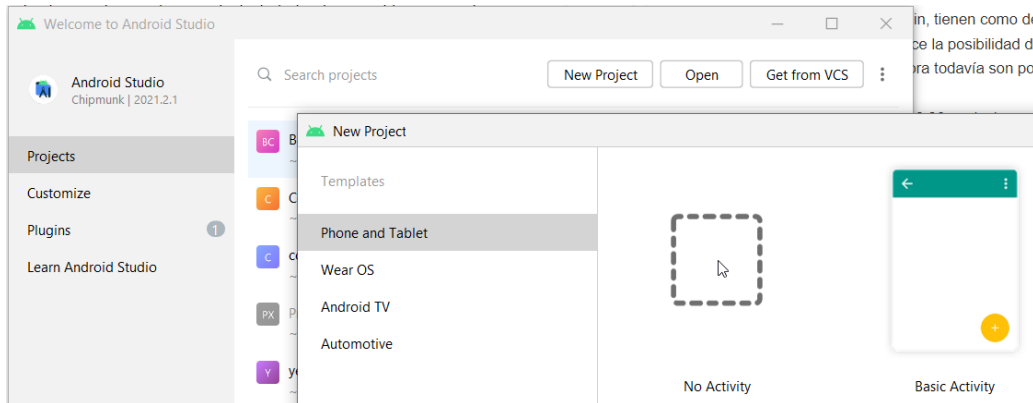
```

- Si quieres ver el resto de tareas predefinidas para nuestro proyecto, puedes ejecutar **gradlew tasks**. Si quieres indagar un poco más sobre la ejecución de tareas puedes ver la [documentación oficial del proyecto](#).
- Por último podremos abrir el proyecto con cualquier IDE que permita importar proyectos de Gradle como **Android Studio**.

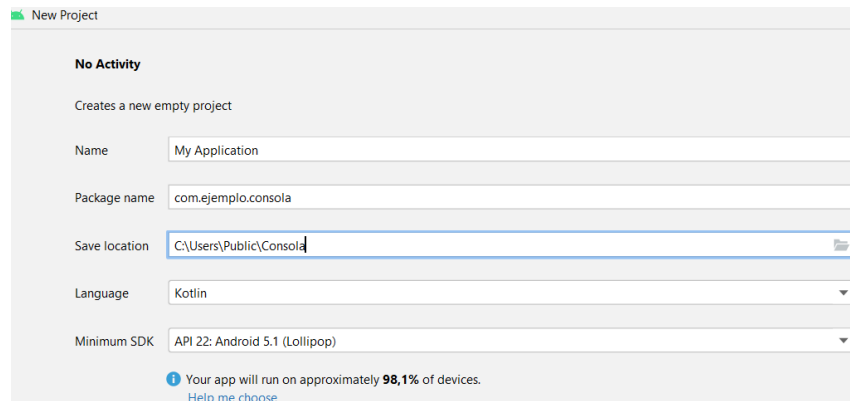
Usando Android Studio

Aunque no es lo recomendado porque añade dependencias de Android que no necesitamos. Podemos usar la plantilla 'No Activity' para hacerlo de una forma asistida.:

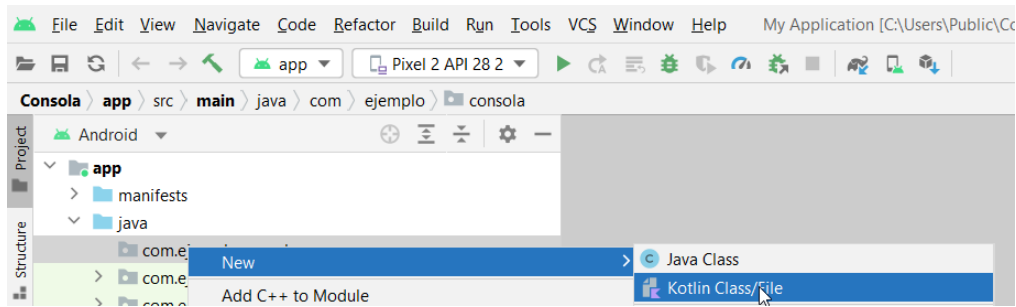
1. Arrancaremos Android Studio y seleccionaremos **New Project -> No Activity**



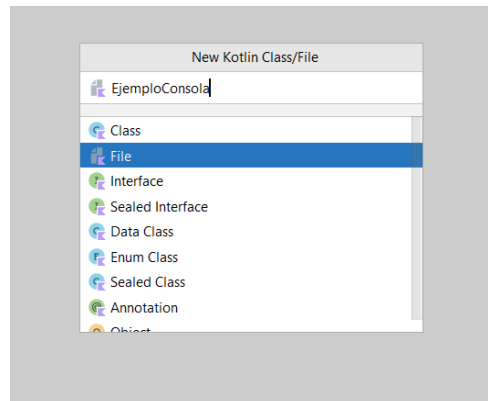
2. Indicaremos la ubicación y el nombre de paquete, esto se explica mejor en el siguiente tema.



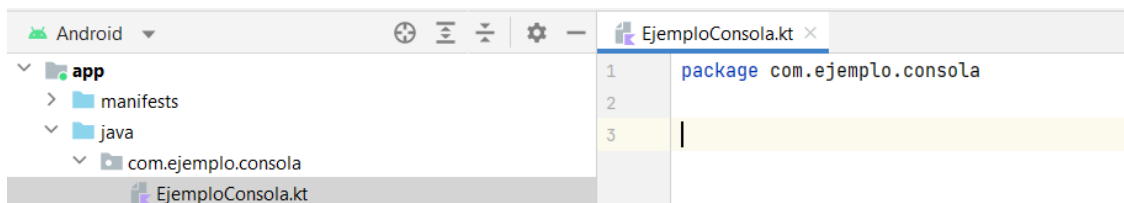
1. Una vez se ha creado el proyecto, dentro del paquete de la carpeta Java (vista Android) seleccionaremos **New -> Kotlin Class/File**



4. Seleccionaremos **File** y le indicaremos el nombre del fichero a crear.



5. De esta manera se habrá creado un fichero con extensión **.kt** (Kotlin), vacío.



6. Ahora solo quedará añadir las funciones que necesitemos, por ejemplo si queremos crear el **Hola Mundo** básico.

```
fun main() {  
    println("Hola Mundo")  
}
```

El lenguaje Kotlin

Control de Nulos

Kotlin ha aportado una serie de elementos que permiten realizar un mayor control de los tipos que pueden ser nulos **Null safety**, y que pretende evitar la tan conocida excepción **NullPointerException**.

Como ocurre con otros lenguajes, Kotlin permite definir un tipo no nulo como anulable. Esto lo hace mediante el elemento **?**, por ejemplo una variable de tipo entera no podría almacenar un valor nulo, pero se puede cambiar esta condición de la siguiente manera.

```
fun main() {  
    var numero: Int  
    numero = null // 🤖 ERROR de compilación, los tipos valor no son anulables  
  
    var numeroAnulable: Int?  
    numeroAnulable = null // OK  
}
```

Para comprobar si las variables son nulas, siempre se puede usar el condicional if/else, aunque Kotlin nos permite otras opciones. El operador **llamada segura** **?.** que solo realizará la llamada en caso que el valor sea distinto de nulo y devolverá null en otro caso.

```
fun main() {  
    var cadenaAnulable: String? = null  
    println(cadenaAnulable?.length)  
}
```

En este ejemplo, si no usáramos el operador de llamada segura, el código lanzaría NPE. Pero ahora mostrará **null**.

Otro operador, no tan recomendado es el operador de **aserción no nula** **!!**. convierte cualquier valor en un tipo no nulo y lanza una excepción si el valor es null. Podemos decir de este operador, que vuelve las cosas a la normalidad ya conocida.

Por lo tanto, si deseas un NPE, puedes tenerla solicitándola con este operador.

```

fun main()
{
    var cadenaAnulable: String? = null

    // 🤖 ERROR de compilación, no permite posibilidad de producir excepción
    println(cadenaAnulable.length)

    //Solicitamos esta opción con el operador !!
    println(cadenaAnulable!!.length)
}

```

Y por último tenemos el operador **elvis** `?:` que comprueba que el valor no es nulo, permitiendo la llamada en ese caso o devolviendo lo que *decidamos* en caso que sea nulo. Equivaldrá al operador `??` en C#.

```

fun main()
{
    var cadenaAnulable: String? = null
    println(cadenaAnulable?.length ?: 0)
}

```

Recuerda que si la expresión a la izquierda de `?:` no es null, se realiza la llamada a length, de lo contrario devuelve la expresión de la derecha, en este caso 0.

Variables

Igual que en todos los lenguajes de programación, en Kotlin también tendremos el recurso de las **variables** para almacenar valores. En Kotlin nos podemos encontrar con variables **inmutables** y variables **mutables**. En el caso de las primeras, una vez se le asigna un valor a la variable no podrá ser modificado, es decir, se comporta como una constante (lo mismo que utilizar final en Java o readonly en C#).

Mientras que con la mutables podremos modificar en cualquier momento el valor de la variable.

Para declarar una variable como mutable, la tendremos que preceder de la palabra clave **var** mientras que para las inmutables usaremos **val**.

El concepto de inmutabilidad es muy interesante. Al no poder cambiar los objetos estos son más robustos y eficientes. Siempre que podamos hemos de usar objetos inmutables.


```
// Variables mutables
var mutable: Int = 5;
mutable += 7;

var numeroDecimales = 3.14F;
numeroDecimales += 5.5F;

// Variables inmutables
val inmutable: Char = 'C';
inmutable = 'A' // 🤖 ERROR compilación!! no se puede modificar una variable inmutable
```

Como se puede ver en el ejemplo, cuando declaramos una variable, podemos indicar el tipo de esta o esperar a que el compilador lo infiera.

Para definir el tipo, tendremos que indicarlo con **:tipo** después del nombre:

```
(var|val) nombreVariable [:tipo][=valor]
```


Los tipos básicos de variables en Kotlin son:

Tipo	Valor	Tamaño
Byte	5.toByte()	8 bits
Short	5.toShort()	16 bits
Int	5	32 bits
Long	5L	64 bits
Float	1.45F	32 bits
Double	1.45	64 bits
Boolean	true	
Char	'H'	16 bits
Unit	Unit	0 bits


 **Importante:** El tipo `Unit` corresponde a `void` en Java y C#

Cuando una **variable mutable** declarada en el cuerpo de una clase, no se quiere inicializar en el momento de la declaración, existe el concepto de **Inicialización tardía**, añadiendo el modificador **lateinit** delante de la declaración de la variable.

```
public class Ejemplo {  
  
    lateinit var cadena: String  
  
    fun miFuncion() {  
        cadena = "Hola Mundo";  
        println(cadena);  
    }  
}
```

 **Importante:** Como en otros lenguajes de programación, se debe controlar las operaciones con variables de distintos tipos, para evitar resultados inesperados o incluso excepciones. Kotlin tiene varios métodos `v.to<Type>()` para cambiar los valores al tipo que necesites:

```
fun main(args: Array<String>) {  
    var a: Int  
    var b = 3.5f  
    a = b + 2 //💀 ERROR error: type mismatch: inferred type is Float but Int was expected  
    a = b.toInt() + 2 //👍 CORRECTO  
    println(a);  
}
```

 **Nota:** Mientras que `a = b.toInt() + 2`, evitaría el error, aunque la función redondearía a 3.

Interpolación de cadenas

El tipo `String` representa el literal de cadena ya conocido, podemos ver un ejemplo del uso de interpolación con `$` para la salida por pantalla.

```
fun main (args: Array <String>) {  
    val cadena = "El resultado de:"  
    val a = 2  
    val b = 5  
    println("$cadena $a + $b es: ${a + b}")  
}
```

El resultado de: 2 + 5 es: 7

Control de flujo

El [control de flujo en kotlin](#) tiene algunas diferencias interesantes a otros lenguajes más conocidos. Las instrucciones **if/else**, **for** y **while** se pueden considerar similares, así que vamos a explicar solamente los elementos que las diferencian:

Condicionales como expresión

En Kotlin no existe el operador ternario para expresar condiciones simples como por ejemplo hacíamos en C# con `string t = exp ? "true" : "false"` en su lugar usaremos `if - else` como hace Python para realizar este tipo de expresiones. Por tanto, el Kotlin tendremos ...

```
val t = if (exp) "true" else "false"
```

Bucles con for y while

Se diferencia sobre todo en que se tienen que usar rangos en la sentencia for y se pueden usar en la while:

```
fun repeticionConForV1(v: Int) {  
    // Fíjate que Kotlin me permite definir rangos de forma similar a C#  
    for (i in (0..v).reversed()) {  
        println("$i")  
    }  
}  
  
fun repeticionConForV2(v: Int) {  
    // Otra forma de expresar lo anterior sería...  
    for (i in v downTo 0 step 1) {  
        println("$i")  
    }  
}  
  
// El operador igual me permite definir 'cuerpos de expresión'  
// al igual que hacíamos en C#  
fun main() = repeticionConForV1(10)
```

En ambos casos anteriores `i` tomara valores de 0 a `v` incluido este último. Si queremos hacer que llegue hasta `v - 1` usaremos `until` ...

```
fun repeticionConIndiceArray(v: Int) {  
    // i irá de 0 a v - 1  
    for (i in 0 until v) {  
        println("$i")  
    }  
}
```

Nota: En la [últimas versiones](#) de Kotlin podemos usar el operador `..<` en lugar de `until` .

También podemos usare un bucle `while` ...

```
fun repeticionConWhile(tope: Int) {  
    var contador: Int = 0  
    do {  
        print("Introduce numero: ")  
        var numero = readLine()!!.toInt();  
        contador++;  
        println(contador)  
    } while (contador < tope && numero !in 50..100)  
}  
  
fun main() = repeticionConWhile(10)
```

Condicional múltiple con when

Equivalente a los **switch** de **expresión** de C#. Sin embargo no dispondremos del **switch** como **instrucción**.

En el siguiente ejemplo podemos deducir la sintaxis fácilmente, teniendo en cuenta que el operador 'arrow' en Kotlin es `->` y en C# o JavaScript es `=>`:

```
fun getEstacion(entrada: Int): String {
    return when (entrada) {
        1 -> "primavera"
        2 -> "verano"
        3 -> "otoño"
        4 -> "invierno"
        else -> "Estación incorrecta"
    }
}

fun main() = println(getEstacion(2))
```

En este caso usamos **when** con patrones que se aplican a la variable **x** como hacíamos en C#.

```
fun main()
{
    val x=12
    val validNumbers= 1..15
    when (x) {
        in 1..10 -> print("x is in the range")
        in validNumbers -> print("x is valid")
        !in 10..20 -> print("x is outside the range")
        else -> print("none of the above")
    }
}
```

Otro ejemplo de la gran funcionalidad que nos permite la sentencia **when** podría ser el siguiente. Observa que en este caso el **when** es **sin argumentos** y tenemos diferentes expresiones que iremos comprobando en orden:

```
fun noHagoNada(x: Int, s: String, v: Float): String {  
    val res = when {  
        x in 1..10 -> "entero positivo menor de 10"  
        s.contains("cadena") -> "incluyo cadena"  
        v is Comparable<*> -> "Soy Comparable"  
        else -> ""  
    }  
    return res  
}  
fun main() = println(noHagoNada(2, "Hola", 3.5f))
```

Funciones

Como ya hemos podido deducir de los ejemplos del tema, si queremos crear una **función** en kotlin tendremos que precederla de la palabra reservada **fun**. Por tanto, una función constará de la palabra fun seguida por el nombre de la función y entre paréntesis los parámetros, siempre y cuando los tenga. Si la función retorna un valor se definirá al final de la signatura (esto último se puede omitir siempre que la función no devuelva nada).

```
fun idFuncion([p1 : T1, p2 : T2, ... ]) : Tr
```

```
fun sumaDatos(datoUno: Int, datoDos: Char) {  
    println("${datoUno + datoDos.code}");  
}  
  
fun main() {  
    var mutable = 5;  
    mutable += 7;  
    val inmutable = 'C';  
    sumaDatos(mutable, inmutable);  
}
```

Con valor de retorno:

```
fun mayor(numeroUno: Int, numeroDos: Int): Int {  
    return if (numeroUno > numeroDos) numeroUno else numeroDos  
}
```

Aunque ya lo hemos usado en algunos ejemplos, si la función tiene una única instrucción o expresión. Se pueden omitir las llaves usando el operador **=**

```
fun mayor(numeroUno: Int, numeroDos: Int): Int =  
    if (numeroUno > numeroDos) numeroUno else numeroDos  
  
fun main() = println(mayor(5, 7))
```

Retornando una tupla 'Pair<A, B>'

Aunque no tenemos la sintaxis de tipo tupla tan avanzada como en Python o C# podemos usar el tipo `Pair<A, B>` para retornar dos valores a la vez.

```
fun angulo(grados: Int): Pair<Double, Double> {  
    val radianes = grados * 3.1416 / 180  
    return Pair(kotlin.math.cos(radianes), kotlin.math.sin(radianes))  
}  
  
fun main() {  
    val (seno, coseno) = angulo(grados = 45)  
}
```


Número de argumentos indefinido con 'varargs'

Para pasar un número variable de argumentos a una función, debemos declarar esa función con un parámetro `vararg`:

```
// Esto significa que la función suma() puede aceptar cero o más enteros.  
fun suma(vararg xs: Int): Int = xs.sum()  
  
fun main() {  
    println(suma())           // Muestra 0  
    println(suma(2))          // Muestra 2  
    println(suma(2, 4, 6))    // Muestra 12  
}
```


Funciones de extensión

Idénticos a los métodos de extensión en C#. Me permiten ampliar la funcionalidad de una clase cumpliendo el principio **OCP** (Abierto para extensión. Cerrado para modificación) de SOLID. No debemos abusar de ellas. Aunque en algunos casos nos van a ayudar a reducir dependencias.

 **Importante:** Su ámbito de aplicación se restringirá al paquete donde se definen y si quisiéramos usarlas en otros paquetes deberíamos haver un import de las mismas.

En el siguiente ejemplo extendemos la funcionalidad de la clase `String` para añadir dos operaciones más sobre los objetos de este tipo.

```
// Función de extensión que dado un objeto de tipo String, convierte
// la primera letra de cada palabra a mayúsculas.
fun String.capitaliza(): String {
    var sCapitalizada: String
    if (!this.isNullOrEmpty()) {

        // El objeto sobre el que realizamos la operación lo podemos
        // referenciar mediante la palabra reservada this.
        val sb = StringBuilder(this)

        sb[0] = sb[0].uppercaseChar()
        for (i in 1 until this.length)
            sb[i] = if (sb[i - 1].isWhitespace()) sb[i].uppercaseChar() else sb[i]
        sCapitalizada = sb.toString()
    } else {
        sCapitalizada = this
    }
    return sCapitalizada
}

// También podemos definir propiedades de extensión.
val String.numeroPalabras: Int
    get() = this.split(' ', '.', '?')
        .filter { it.isNotEmpty() }
        .size

fun main() {
    val texto = "esto es una cadena de texto"
    println(texto.capitaliza())
    println(texto.numeroPalabras)
}
```

Excepciones

👉 **Importante:** `try` es una **expresión**. Esto es interesante pues va a ser muy común retornar un `try` en un método teniendo en cuenta que la última instrucción de cada bloque debe evaluarse al mismo tipo.

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

👉 **Importante:** Podemos definir funciones que lancen una excepción, pero deberemos indicar que no retorna nada con la palabra reservada `Unit` que equivaldrá al `void` de C#.

```
fun fail(message: String): Unit = throw IllegalArgumentException(message)

fun main() {
    val name = readLine()
    val s = name ?: fail("Name required")
    println(s)
}
```

Clases e Interfaces

Todos los elementos en Kotlin son públicos por defecto, por lo que también lo serán las [clases](#). Las clases nos sirven para referenciar objetos del mundo real, y mediante las propiedades podemos definir las distintas características que nos interese manipular de estos. En Kotlin ya no existen los atributos, sino que todo pasa a ser propiedades, similares a las que utiliza C#, se les puede dar funcionalidad si lo necesitamos.

Por tanto, una clase **Persona** con **nombre** y **edad** a las que queremos dar funcionalidad, en C# se podría definir así:

```
// C# Example
class EdadException : Exception
{
    public EdadException(string message) : base(message) { }
}

class Persona
{
    private string _nombre = "";
    private int _edad = 0;

    public string Nombre
    {
        get => _nombre.ToUpper();
        private set => _nombre = value
    }

    public int Edad
    {
        get => _edad;
        set
        {
            if (value < 0 || value > 125)
                throw new EdadException("Edad Invalida");
            _edad = value;
        }
    }

    public Persona(string nombre, int edad) {
        Nombre = nombre;
        Edad = edad;
    }
}
```

Su equivalente en Kotlin sería ...

```
class EdadException(cadena: String) : Exception(cadena)

class Persona(nombre: String, edad: Int = 0) {
    var nombre: String = ""
    // get() y set() deben ir indentados justo después de
    // definir la propiedad.
    get() = field.uppercase()
    private set

    var edad: Int = 0 // Si no es lateinit debe ir inicializado.
    set(value) {
        if (value < 0 || value > 125)
            throw EdadException("Edad Invalida")
        field = value
    }

    init {
        this.nombre = nombre
        this.edad = edad
    }
}
```

Si te fijas, además de la palabra reservada **value** que ya teníamos en C#. Kotlin utiliza la palabra preservada **field** para referirse al '*campo anónimo*' asociado a la propiedad, por lo que no tiene que crear un campo asociado como ocurre con C#. Además, los identificadores de las propiedades en Kotlin irán en camel-casing.

 **Resumen:** Podemos deducir pues que **en Kotlin no podemos definir campos** en las clases sino **solo propiedades**.

Constructores

Las clases pueden tener un constructor principal y uno o más constructores secundarios. El constructor principal es parte del **encabezado de la clase**, como vimos en el anterior ejemplo. Esto nos permite construir un objeto sin tener que definir el constructor. Por ejemplo, en otra versión de la clase **Persona** anterior, pero sin funcionalidad en las propiedades y estas de solo lectura, solo bastaría con haber hecho lo siguiente:

```
// Fíjate que al usar val nombre es una propiedad de solo lectura.
class Persona(val nombre: String, val edad: Int = 0)

val persona = Persona("Pepe", 23)
```

En la definición equivalente en C# sí que tendríamos que definir las propiedades y el constructor. Mientras que en Kotlin lo hemos hecho de forma muy simplificada:

```
// C# Example
class Persona
{
    public string Nombre { get; }
    public int Edad { get; }

    public Persona(string nombre, int edad) {
        Nombre = nombre;
        Edad = edad;
    }
}

Persona persona = new("Pepe", 23);
```

Si necesitamos que ocurra algo en el momento de crear el objeto, se puede usar la cláusula **init** que será ejecutada al llamar al constructor principal. Si te fijas en el siguiente código, la propiedad **edad** está declarada en el constructor, mientras que el **nombre** se ha declarado en el cuerpo de la clase.

```
// En esta definición nombre no lleva la palabra reservada val
// porque lo vamos a definir dentro del cuerpo de la clase.
class Persona(nombre: String, var edad: Int = 0) {
    // Podemos no asignarle un valor a la propiedad
    // ya que lo asignamos en el init.
    val nombre: String
        get() = field.uppercase()


    init {
        this.nombre = nombre
        if (edad < 0 || edad > 125)
            throw ExcepcionEdad("Edad Invalida")
        this.edad = edad
    }
}
```

Si lo que queremos es crear más de un constructor, tendremos que recurrir a la cláusula **constructor** indicando los parámetros necesarios y llamando al constructor por defecto con **this** para enviarle sus propiedades. Ahora hemos añadido una propiedad más a la clase **Persona**, el dni que será asignado con el segundo constructor

```
class Persona(nombre: String, var edad: Int = 0) {
    var dni: String = "NINGUNO"
    val nombre: String
        get() = field.uppercase()

    init {
        this.nombre = nombre
        if (edad < 0 || edad > 125) throw ExcepcionEdad("Edad Invalida")
        else this.edad = edad
    }

    // Fíjate que un constructor llama al otro.
    constructor(nombre: String, edad: Int = 0, dni: String)
        : this(nombre, edad) {
        this.dni = dni
    }
}
```

 **Resumen:** Fíjate en este otro ejemplo en el que a través de la definición de una clase **Cuenta** vemos las características de sintaxis descritas.

```
// Definición de la clase y del constructor principal.
// Además, hemos definido implícitamente dos propiedades titular.
class Cuenta(val titular: String, val número: Int) {

    // Propiedad mutable privada solo para modificación e inicialización.
    // Define un get y set auto-implementado como C#.
    var saldo: Double = 0.0
        private set

    // Propiedad pública mutable y anulable
    // en la que definimos nosotros el get y el set
    // field: palabra reservada para hacer referencia al campo asociado a la propiedad.
    // value: palabra reservada para hacer referencia al valor recibido en el setter.
    var banco: String? = null
        get() = field ?: "Desconocido"
        set(value) {
            field = value!!.toString()
        }

    // Propiedad pública de solo lectura calculada y por tanto (immutable)
    // También se puede declarar así -> val hayDescubierto: Boolean = saldo < 0
    val hayDescubierto: Boolean
        get() = saldo < 0 // Definimos el getter para la propiedad

    // Constructor secundario apoyándose en el principal.
    constructor(titular: String, número: Int, saldo: Double) : this(titular, número) {
        this.saldo = saldo
    }

    // Método público normal
    fun ingreso(cantidad: Double) {
        saldo += cantidad
    }

    // Invalidación ToString con cuerpo de expresión
    override fun toString(): String = "Cuenta $número titular $titular saldo ${saldo}€"

    // Método público que lanza una excepción
    @Throws(IllegalArgumentException::class)
    fun reintegro(cantidad: Double) {
        if (cantidad > saldo) throw IllegalArgumentException("Saldo insuficiente")
        saldo -= cantidad
    }
}
```


Definir métodos y propiedades estáticas

Creemos un **objeto global anónimo** denominado por ejemplo `Datos`. En otras palabras, no estamos definiendo una clase sino un objeto instanciado, referenciado por el id `Datos` y de tipo anónimo.

Después usaremos la anotación `@JvmStatic` para indicar que es un método o propiedad de clase. Si no, ponemos `@JvmStatic` estaremos definiendo un **Singleton**.

```
object Datos {  
    @JvmStatic  
    val empleados = listOf(  
        Empleado("Xusa", 45, Empleado.Ciudad.Alicante),  
        Empleado("Pepe", 54, Empleado.Ciudad.Alicante),  
        Empleado("Juanjo", 52, Empleado.Ciudad.Elche),  
        Empleado("Vicente", 45, Empleado.Ciudad.Elche))  
}  
  
// Para acceder a la propiedad estática empleados  
Datos.empleados
```

Si queremos hacerlo en otra clase, marcaremos el objeto creado con el modificador `companion`

 **Importante:** Fíjate que no hace falta que le pongamos un identificador de tipo al `companion object` ya que solo podemos definir uno por clase y dentro del mismo definiremos los métodos o propiedades 'estáticas' de la misma.

```
class MiClase  
{  
    companion object {  
        val empleados = listOf(  
            Empleado("Xusa", 45, Empleado.Ciudad.Alicante),  
            Empleado("Pepe", 54, Empleado.Ciudad.Alicante),  
            Empleado("Juanjo", 52, Empleado.Ciudad.Elche),  
            Empleado("Vicente", 45, Empleado.Ciudad.Elche))  
        }  
    }  
}  
  
// Para acceder a la propiedad estática empleados.  
MiClase.empleados
```


Herencia

En Kotlin, la clase `Any` es la raíz de la jerarquía de clases. Cada clase del lenguaje derivará de ella si no especificas una superclase. Sería similar a la clase `Object` de C# y Java.

Por otro lado en Kotlin, tanto las clases como los miembros de estas son cerrados, esto significa que no se puede heredar de una clase y tampoco se pueden sobrescribir sus miembros si no lo indicamos explícitamente.

Para que de una clase se pueda heredar habrá que añadirle el modificador `open`. Por ejemplo, si la clase `Persona` queremos hacerla abierta sería:

```
open class Persona(nombre: String, var edad: Int = 0) {  
    ...  
}
```

y ahora podríamos crear una clase hija de `Persona`, como por ejemplo:

```
class Estudiante(nombre: String, edad: Int = 0, var estudios: String)  
: Persona(nombre, edad)
```

Suponiendo que la propiedad `dni` queremos hacerla invalidable y que tiene un nuevo método también invalidable llamado `imprimir` y uno normal `esMayor`, ahora la clase quedaría:

```
open class Persona(nombre: String, var edad: Int = 0) {  
    open var dni: String = "NINGUNO"  
    ...  
    open fun imprimir() = println("Nombre: $nombre Edad: $edad")  
    fun esMayor() = edad >= 18  
}
```

y con los elementos que queremos invalidar en la clase `Estudiante` usaremos la palabra reservada `override` como en otros lenguajes. Así mismo usaremos la palabra reservada `super` para referenciar la la 'superclase' como en Java.

```
class Estudiante(nombre: String, edad: Int = 0, var estudios: String)  
: Persona(nombre, edad) {  
    override var dni: String = "ESTUDIANTE SIN DNI"  
    override fun imprimir() {  
        super.imprimir();  
        println("Soy estudiante de $estudios")  
    }  
}
```

Interfaces

En Kotlin podemos implementar clases abstractas, que son iguales a las que ya conocemos de otros lenguajes, salvo los cambios concretos para la herencia. Por esta razón, no vamos a comentar nada sobre ellas.

También se pueden crear [interfaces](#), que permiten definir tipos cuyos comportamientos pueden ser compartidos por varias clases que no están relacionadas. Usa la palabra reservada `interface` y su implementación es similar a los lenguajes que conocemos con algunas pequeñas diferencias. Como ya sabemos permiten la herencia múltiple, y además:


- Pueden contener métodos abstractos (sin implementación) y métodos regulares (con implementación).
- Puede contener propiedades abstractas y regulares.
- No permite declaración de constructores.
- Las propiedades y métodos regulares de una interfaz pueden ser invalidados con el modificador `override` sin tener que marcarlos con `open`, a diferencia de en las clases abstractas.

```
interface Estudios {  
    var curso: Int           // Propiedad abstracta  
    val ultimoCurso: Boolean // Propiedad regular  
        get() = curso == 2  
    fun estudios(): String   // Método abstracto  
    fun soyEstudiante() =    // Método regular  
        println("Soy Estudiante de " + estudios())  
}
```

y ahora hacemos que la clase `Estudiante` además de heredar de `Persona`, implemente la interface `Estudios`, quedando:

```
class Estudiante(nombre: String, edad: Int = 0, var estudios: String)  
    : Persona(nombre, edad), Estudios {  
    override var curso: Int = 0  
        set(value) {  
            field = curso  
        }  
    override var dni: String = "ESTUDIANTE SIN DNI"  
    override fun estudios() = estudios  
}
```

Definiendo ValueObjects o DTO's con data class

 **Importante:** La gran mayoría de objetos que definiremos en nuestros programas serán de este tipo. Especialmente al recuperar información de las fuentes de datos.

Si queremos definir objetos con tipos que actúen como *'Value Objects'* o *DTO's* como el tipo `record` en C# 10 o Java 17, tenemos la posibilidad de definirlos como `data class` en Kotlin. Sus características son:

```
// Fíjate que al ser inmutables todos las propiedades del data class
// se definirán con val.

data class Empleado(
    val nombre: String,
    val edad: Int,
    val ciudad: Ciudad) {

    // Aunque definimos un cuerpo por anidar la definición del Enum
    // no haría falta hacerlo en un data class
    enum class Ciudad() { Elche, Alicante }
}

fun main() {
    // La sintaxis recomendada para construir un objeto es la
    // siguiente donde especificamos pares
    // en líneas diferentes. Nos facilitará lla edición, borrado
    // y cambio de orden de las propiedades.
    val e1 = Empleado(
        nombre = "Xusa",
        edad = 45,
        ciudad = Empleado.Ciudad.Alicante
    )

    // Como ves la sintaxis proiedad = valor nos permite no seguir
    // el orden de declaración en la inicilización de las propiedades.
    val e2 = Empleado(
        edad = 53,
        nombre = "Pepe",
        ciudad = Empleado.Ciudad.Alicante
    )

    // También podremos usar la forma tradicional.
    val e3 = Empleado("Juanjo", 52, Empleado.Ciudad.Elche)
    val e4 = Empleado("Juanjo", 52, Empleado.Ciudad.Elche)
}
```

Este tipo de clases:

1. Definen por defecto `equals()` , `hashCode()` así como `==` y `!=`

```
println(e3 == e4)    // muestra true en lugar de comparar referencias
```

2. Definen por defecto `toString()`

```
println(e3)          // muestra "Empleado(nombre=Juanjo, edad=52, ciudad=Elche)"
```

3. Ya que son inmutables, tenemos la posibilidad de crear fácilmente copias con `copy()`

```
val e5 = e4.copy(ciudad = Empleado.Ciudad.Alicante)
println(e4)          // Mostrara "Empleado(nombre=Juanjo, edad=52, ciudad=Elche)"
println(e5)          // Mostrara "Empleado(nombre=Juanjo, edad=52, ciudad=Alicante)"
println(e3 == e5)    // Mostrara false
```

Clases y métodos parametrizados

Kotlin también permite crear clases y métodos con alguno de sus miembros de tipo genérico. La lista de parámetros para tipos se incluyen en paréntesis angulares y se separan por coma si son varios `<T, U, V, ...>`

Para crear una clase con un tipo parametrizado de forma que una de sus propiedades sea de ese tipo, se hará de la siguiente manera:

```
class ClaseGenerica<T>(  
    private var t: T,  
    private val c: String) {  
  
    fun metodo(param: T) {  
        t = param  
    }  
    override fun toString() = "${t} ${c}"  
}
```

y si quisiéramos crear un objeto de esa clase con la propiedad parametrizada a tipo entero, se podría hacer ...

```
fun main() {  
    val objeto = ClaseGenerica(3, "Hola")  
}
```

Si quisiéramos realizar una restricción del tipo parametrizado a la interfaz `Comparable<T>` como hacíamos en C#, se tendría que hacer de la siguiente manera:

```
class ClaseGenerica<T: Comparable<T>>(  
    private var t: T,  
    private val c: String) {  
    ...  
}
```

Scope functions

Kotlin nos permite usar una [serie de métodos](#) denominadas **Scope Functions** (Funciones de Ámbito) que nos permiten trabajar con objetos de forma más cómoda y en una **única expresión**. Muchas de ellas son aproximaciones para hacer la misma cosa. Pero las principales y más utilizadas son ...

- **let** : Nos permite ...

Hacer algo con un objeto anulable si es distinto de null

```
val nombre: String? = "Pepe"
nombre?.let { println(it) }
```

Introducir una expresión como variable en el ámbito local.

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }
    .let { it.joinToString(" - ") }
    .let { println(it) } // Muestra 5 - 4 - 4
```

- **apply** : Configuración de un objeto. Todas las operaciones se hacen sobre **this** .

```
val persona = Persona().apply {
    nombre = "Pepe"
    edad = 45
}
```

Podemos considerarlo una forma hacer un **interfaz fluida** sobre un objeto que no la tiene definida.

- **run** : Para ejecutar instrucciones donde se requiera una **expresión**.

```
val hexNumberRegex = run {
    val digits = "0-9"
    val hexDigits = "A-Fa-f"
    val sign = "+-"

    Regex("[$sign]?[$digits$hexDigits]+")
}
```

Clases enumeradas con enum class

Igual que en otros lenguajes, Kotlin nos permite crear tipos **enumerados**, aunque en este caso se puede ver como un modificador de clase. Una enumeración es un conjunto de valores que usan como identificador un nombre. Dicho nombre se comporta como una constante en nuestro lenguaje. Al marcar una clase con el modificador **enum**, la declara como una de enumeración.

```
enum class CiclosInformatica { SMR, ASIR, DAM, DAW }

fun nivelCiclo(ciclo: CiclosInformatica): String {
    return when (ciclo) {
        CiclosInformatica.SMR -> "Medio"
        else -> "Superior"
    }
}

fun main() {
    val ciclo = CiclosInformatica.ASIR
    println(nivelCiclo(ciclo)) //Superior
}
```

Valor en las enumeraciones

Además a las enumeraciones en Kotlin también podemos asignarles uno o más valores. Esto se hará a través del constructor de la clase, lo podemos ver en el siguiente ejemplo, en el que al constructor se le ha añadido tanto un valor entero como un grado de tipo cadena.

```
enum class CiclosInformatica(
    private val valor: Int,
    private val grado: String
) {
    SMR (1, "Grado Medio"),
    ASIR(2, "Grado Superior"),
    DAM (3, "Grado Superior"),
    DAW (4, "Grado Superior")
}

fun main() {
    for (v in CiclosInformatica.values()) {
        println(v.name + " " + v.ordinal)
    }
    // Recorrido de secuencia pasando una HOF como Consumer
    CiclosInformatica.values().forEach { ciclo ->
        // Formateo de cadenas similar a Java
        println("%-2d %-4s %s".format(ciclo.valor, ciclo.name, ciclo.grado))
    }
}
```

```
SMR 0
ASIR 1
DAM 2
DAW 3
1 SMR Grado Medio
2 ASIR Grado Superior
3 DAM Grado Superior
4 DAW Grado Superior
```

Enumeraciones con comportamiento

También se les puede añadir un comportamiento a través de funciones abstractas o no, o incluso de interfaces. En el siguiente ejemplo podemos ver que al constructor se le ha añadido un elemento más con el nombre completo del ciclo, y además tenemos el comportamiento añadido mediante el método `informacionCompleta()`. De forma que la ejecución del programa nos sacará las siglas del ciclo, el nombre completo y el grado que le corresponde a cada uno de los elementos de la enumeración.

```
enum class CiclosInformatica(
    val valor: Int,
    val grado: String,
    val nombre: String
) {
    SMR(1, "Grado Medio", "Sistemas Microinformáticos y Redes"),
    ASIR(2, "Grado Superior", "Administración de Sistemas Informáticos en Red"),
    DAM(3, "Grado Superior", "Desarrollo de Aplicaciones Multiplataforma"),
    DAW(4, "Grado Superior", "Desarrollo de Aplicaciones Web");

    fun informacionCompleta()= "${name} - ${nombre} - ${grado}"
}
fun main() {
    CiclosInformatica.values().forEach{println("${it.informacionCompleta()}")}
}
```


Funciones Genéricas

Para las funciones genéricas el parámetro de tipo se añadirá justo después de la palabra reservada `fun` y las restricciones se harán de la misma manera que en las clases.

```
fun <T> funcionGenerica(param: T): T = param
```

Supongamos que definimos una clase estática con métodos de utilidad sobre `tablas dentadas` cuyo contenido sean tipos diferentes y definimos un método `flat` que las convierte en un array unidimensional.

```
object TablaDentada {  
    // Una función inline es una función que se expande en el código que la llama.  
    // Esto permite que el compilador pueda optimizar el código de la función.  
    // reified: Permite acceder al tipo de dato de la función.  
    inline fun <reified T> flat(tablaDentada: Array<Array<T>>): Array<T> {  
        val d = mutableListOf<T>()  
        for (i in tablaDentada.indices) {  
            for (j in tablaDentada[i].indices) {  
                d.add(tablaDentada[i][j])  
            }  
        }  
        return d.toTypedArray()  
    }  
}
```

El programa principal para probarla con tablas dentadas de diferente tipos podría ser ...


```
fun main() {  
    val tablaDentadaInt = arrayOf(  
        arrayOf(1, 2, 3, 4),  
        arrayOf(4, 5, 6),  
        arrayOf(7, 8, 9, 10, 11)  
    )  
    println(TablaDentada.flat(tablaDentadaInt).contentToString())  
  
    val tablaDentadaString = arrayOf(  
        arrayOf("a", "b", "c", "d"),  
        arrayOf("e", "f", "g"),  
        arrayOf("h", "i", "j", "k", "l")  
    )  
    println(TablaDentada.flat(tablaDentadaString).contentToString())  
}
```

Clases Selladas (Sum Type Pattern)

Las clases selladas son un tipo de clase que nos permite definir un tipo de dato que puede ser de un tipo u otro, pero no de ambos. Es decir, que no se puede crear un objeto de una clase sellada, sino que se creará un objeto de una de sus clases hijas.

En el fondo **podríamos decir que es un tipo enumerado mejorado** de hecho es un superconjunto y cualquier tipo enumerado podríamos representarlo a través de una clase sellada. La diferencia radica en que en los enumerados sólo tenemos un único objeto por tipo, mientras que en las sealed classes podemos tener varios objetos de la misma clase y **permiten almacenar estado**.

Esto es la forma en que Kotlin y otros lenguajes como [C# implementan](#) el patrón funcional [Sum Type Pattern](#).

 **Resumen:** En resumen podemos decir que una clase sellada permite definir una enumeración de tipos con propiedades diferentes de los cuales una instancia solo podrá ser de uno de ellos de forma excluyente.

Veámoslo a través de varios ejemplos comentados:

Supongamos primero el siguiente tipo enumerado...

```
enum class TipoUsuario(val tipo: String) {  
    Admin("Admin"),  
    Personal("Personal"),  
    Cliente("Cliente")  
}
```

La forma equivalente de representar su funcionalidad a través de una clase sellada sería ...

```
sealed class TipoUsuario(val tipo: String) {

    // Definimos tres instancias estáticas de objetos
    // de tipo estableciendo la propiedad común tipo
    // a los respectivos valores del enumerado.
    object Admin : TipoUsuario("Admin")
    object Personal : TipoUsuario("Personal")
    object Cliente : TipoUsuario("Cliente")

    companion object {
        // Método estático que me devuelve los valores.
        fun values(): Array<TipoUsuario> {
            return arrayOf(Admin, Personal, Cliente)
        }

        // Método estático para tener la funcionalidad
        // equivalente del valueOf de los enums
        fun valueOf(value: String): TipoUsuario {
            return when (value) {
                "Admin" -> Admin
                "Personal" -> Personal
                "Cliente" -> Cliente
                else -> throw IllegalArgumentException(
                    "No object consola.TipoUsuario.$value")
            }
        }
    }
}
```

Pero en este caso no nos aporta nada frente al `enum class` que de forma más concisa me representa dicha funcionalidad. Con lo cual vamos a quedarnos con dicha sintaxis y vamos a definir dos tipos enumerados más.

```
enum class TipoCategoria(val tipo: String) {
    Mascotas("Mascotas"),
    Gatos("Gatos"),
    Perros("Perros")
}

enum class TipoArchivo(val tipo: String) {
    Foto("Foto"),
    Video("Video"),
    Audio("Audio")
}
```

Supongamos ahora una clase `AccionesUsuario` con diferentes acciones que puede realizar un usuario. En este caso, el usuario puede hacer login, logout, visualizar un archivo o buscar un archivo. En el caso de visualizar y buscar, se le pasa el nombre del archivo, el tipo de usuario, el

tipo de archivo y en el caso de buscar, el tipo de categoría. Cada tipo de acción va a estar representada por un tipo anidado **con sus propiedades específicas** y que heredará del propio

AccionesUsuario .

```
sealed class AccionesUsuario {
    // Tipos Login y Logout
    data class Login(val tipoUsuario: TipoUsuario) : AccionesUsuario()
    data class Logout(val tipoUsuario: TipoUsuario) : AccionesUsuario()

    // Tipos Visualizar y Buscar ya definen otras propiedades
    // específicas para definir su propio estado.
    data class Visualizar(
        val fileName: String,
        val tipoUsuario: TipoUsuario,
        val mediaType: TipoArchivo
    ) : AccionesUsuario()

    data class Buscar(
        val terminoBusqueda: String,
        val tipoUsuario: TipoUsuario,
        val tipoArchivo: TipoArchivo,
        val tipocategoria: TipoCategoria
    ) : AccionesUsuario()

    // Defino un método para pasar a cadena que según el tipo
    // generamos una cadena con las características del estado.
    fun aTexto(): String {
        return when (this) {
            is Login ->
                "El usuario $tipoUsuario ha iniciado sesión"

            is Logout ->
                "El usuario $tipoUsuario ha cerrado sesión"

            is Visualizar ->
                "El usuario de tipo $tipoUsuario ha visto " +
                    "el fichero $fileName que es un $mediaType"

            is Buscar ->
                "El usurio de tipo $tipoUsuario ha buscado \"$terminoBusqueda\" " +
                    "de tipo $tipoArchivo en la categoría $tipocategoria"
        }
    }
}
```

En el siguiente código simulamos que vamos añadiendo acciones del usuario a una lista que hará de 'Log' cada una con sus propiedades específicas pero de entre un conjunto restringido de acciones posibles. Todas ellas tienen en común la operación **aTexto** que según el tipo de acción nos devolverá una cadena con las características de la acción realizada.

```
fun registroAcciones() {
    val acciones = mutableListOf<AccionesUsuario>().apply {
        add(AccionesUsuario.Login(tipoUsuario = TipoUsuario.Admin))
        add(
            AccionesUsuario.Visualizar(
                fileName = "gato mirando cosas",
                tipoUsuario = TipoUsuario.Personal,
                mediaType = TipoArchivo.Video
            )
        )
        add(
            AccionesUsuario.Buscar(
                terminoBusqueda = "perro gracioso",
                tipoUsuario = TipoUsuario.Cliente,
                tipoArchivo = TipoArchivo.Foto,
                tipocategoria = TipoCategoria.Perros
            )
        )
        add(AccionesUsuario.Logout(tipoUsuario = TipoUsuario.Cliente))
    }

    acciones.forEach { println(it.aTexto()) }
}
```

Delegación de propiedades con `by`

La [delegación de propiedades](#) es una característica de Kotlin que nos permite delegar la implementación de una propiedad a un objeto externo. Por tanto, utilizará otro objeto que es capaz de devolver un resultado cuando se llame al `get` y al `set` (en caso de que se utilice `var`).

Veamos el siguiente ejemplo comentado:

```
// Definimos un tipo Dato que guarda un valor
class Dato(var valor : Int){
    override fun toString(): String {
        return valor.toString()
    }
}

// Definimos un tipo delegado encargado de inicializar una propiedad
class MiDelegadoParaDato {


    // EL delegado devuelve una única instancia de Dato
    // o la crea si no existe (patrón creacional Singleton)
    companion object {
        var dato: Dato = Dato(-1)
    }

    // Como solo hemos definido getValue,
    // solo podremos utilizarlo en propiedades de solo lectura 'val'
    operator fun getValue(thisRef: Any?, property: KProperty<*>): Dato {
        return dato
    }
}

fun main() {
    // En ambos casos se inicializa la propiedad
    // a la misma instancia del objeto dato.
    val dato1: Dato by MiDelegadoParaDato()
    val dato2: Dato by MiDelegadoParaDato()

    println("dato1 = ${dato1} y dato2 = ${dato2}")
    // mostrará dato1 = -1 y dato2 = -1

    dato1.valor = 6
    println("dato1 = ${dato1} y dato2 = ${dato2}")
    // mostrará dato1 = 6 y dato2 = 6
}
```

 **Importante:** Fíjate que hemos usado la palabra reservada `by` para indicar que inicialización de los datos se va a delegar en una instancia de del tipo `MiDelegado`.

lazy

Existen **delegados estándar** ya implementados en el lenguaje que nos permiten implementar la delegación de propiedades de forma sencilla. Por ejemplo, tenemos la interfaz **lazy** que nos permite inicializar el valor de una propiedad de **forma perezosa**. Es decir, que no se inicializa hasta que no se accede a ella por **primera vez**.

Por ejemplo:

```
// Definimos una clase llamada Dato
data class Dato(val valor : Int = 0)

// Definimos una clase llamada A
class A {
    // Definimos una propiedad pública de tipo Dato
    // que se inicializa de forma perezosa.
    val p: Dato by lazy { Dato() }
}

fun main() {
    val a = A()

    // Hasta el momento que accedemos a la propiedad p
    // no se llama al constructor de Dato.
    println(a.p)
}
```

observable

Existen muchos estelados estándar que nos permiten implementar la delegación de propiedades usando `by`. Por ejemplo, la interfaz `observable` nos permite observar los cambios que se producen en una propiedad. Es decir, que cada vez que se cambie el valor de la propiedad, se ejecutará un código que nosotros definamos. Muchos de ellos están definidos en el paquete `kotlin.properties` dentro de la clase `Delegates`.

```
data class Dato(val valor : Int = 0)

class A {
    // Definimos una propiedad pública de tipo Dato que se inicializa al
    // valor por defecto y cada vez que se modifica se la función lambda que se le indica
    // que se modifica se la función lambda que se le indica
    var p: Dato by Delegates.observable(Dato()) {
        _, old, new -> println("p cambia de $old a $new")
    }
}

fun main() {
    val a = A()
    a.p = Dato(1)
    a.p = Dato(4)
}
```

Al ejecutarse este programa principal se mostrará por el terminal...

```
p cambia de Dato(valor=0) a Dato(valor=1)
p cambia de Dato(valor=1) a Dato(valor=4)
```