Tabla de contenidos

1. Introducción. Estado del arte

Hasta hace unos años los lenguajes de programación más usados se encontraban divididos en 3 ámbitos dependiendo de las necesidades:

- Servidor o BackEnd.
- Cliente o FrontEnd usando un navegador web.
- Clientes para aplicaciones nativas.

Actualmente ha empezado a utilizarse el mismo lenguaje para los tres ámbitos, realizándose una adaptación en el caso del cliente (Android, iOS, Web o Escritorio).

Las grandes empresas de software han desarrollado o se encuentran desarrollando sus propias plataformas para la creación de aplicaciones multiplataforma, con dos filosofías:

- Escritura del código en un lenguaje y generación en la compilación aplicaciones nativas para cada uno de los clientes. Los más destacados son:
    - Kotlin Multiplataform. Promovido por Google y JetBrains, en especial esta última.
    - .Net MAUI. De Microsoft, se desarrolla principalmente con C#.
    - Flutter. De Google, con un éxito relativo, hace unos meses se anuncio por parte de Google su desinteres y la apuesta por Kotlin Multiplataform.
    - Xamarin. Similar a Flutter en cuanto al abandono, pero en este caso por Microsoft.

Existen otras como Qt o Titanium SDK, pero el porcentaje de uso es testimonial.

- Escritura del código en Javascript/Typescript usando frameworks o librerías de terceros creando aplicaciones nativas que realmente son un navegador.
    - React Native. Usando Javascript/Typescript, en el cliente se tiene una máquina virtual que interpreta el código Javascript, pudiendo acceder al hardware del dispositivo.
    - Cordova. Uno de los primeros frameworks multiplataforma. Posee una serie de librerías que permite el acceso al hardware del dispositivo, se centra en las aplicaciones móviles. Para el usuario las aplicaciones escritas en Cordova no se diferencian de las nativas, denominándose aplicaciones híbridas, ya que no son 100% nativas, ni 100€ aplicación web. Es sofware libre, existiendo una versión de Adobe denominada PhoneGap que permite compilaciones en los servidores de Adobe
    - Ionic. Construido sobre Cordova, mejora el desarrollo del front-end y la interacción con en usuario.

Con respecto al servidor, existe también una gran cantidad de lenguajes/librerías/frameworks. En ocasiones la librería/framework ha devorado al lenguaje, siendo más popular que este, por ejemplo Spring en Java o Laravel en PHP. En el siguiente listado se definen tanto de lenguajes como de frameworks dentro de los lenguajes, o frameworks disponibles en varios lenguajes:

- .Net de Microsoft.
- PHP, con decenas de frameworks, siendo los más conocidos Laravel y Sympony.
- NodeJS. Basado en Javascript/TypeScrit.
- Jakarta. Java, en desuso, entorno institucional y bancario, anteriormente conocido como JEE.
- Spring. Disponible para varios lenguajes como Java y Kotlin.
- Python. Gran variedad, siendo los más conocidos Flask (sencillo), Django (más complejo) o CherryPy(sencillo).

Otra de las características de los lenguajes/frameworks actuales es la unión de clientes y servidores para el manejo de información. La comunicación se reliaza usando operaciones asíncronas, es decir, no se conoce el tiempo en que se realizará la petición (ficheros, json en web, validación...) y la aplicación que realiza la petición puede o no quedarse bloqueada a la espera de la finalización (normalmente no se bloquea la parte gráfica, de forma que se pueda seguir usándola). Proporcinando los lenguajes actuales, y los más antiguos operaciones y mecanimos primitivos (semáforos, monitoriores, regiones críticas...), centrados estos más en la comunicación entre procesos y la asignación de recursos, como operaciones de nivel superior para dar facilitar el manejo de operaciones asíncronas como corrutinas, promesas, callback..., dependiendo estas operaciones del lenguaje/framework.

Kotlin es un lenguaje relativamente moderno, que ofrece librerías, plataformas y frameworks que cumplen con las necesidades de las aplicaciones actuales (o hacia dónde se piensa que evolucionará la industria en los próximo años), siendo estas:

- Desarrollo en servidor.
- Creación de aplicaciones multiplataforma.
- Gestión de operaciones asíncronas.

Aunque no es el único, los fundamentos del desarrollo si son muy similares entre los diferentes lenguajes/plataformas existentes, independientemente de estas.

1.1. Historia.

La aparición de Java en los año noventa del siglo pasado fue una revolución en la industria del software, hasta ese momento, las aplicaciones se compilaban para una arquitectura hardware y sistema operativo concreto. Java introduce el concepto de máquina virtual sobre las que se ejecutan las aplicaciones, utilizando un lenguaje cercano al ensamblador, en el caso de Java BitCode. Con esto se consigue que un programa escrito en Java puede ser ejecutado en cualquier sistema operativo en la que se halla implementado la máquina virtual (JDK), por supuesto esta flexibilidad lleva asociada una bajada de rendimiento, que se ha idos solucionando a lo largo de los años.

No tardo mucho tiempo en comenzar a aparecer adaptaciones de lenguajes existentes (Pascal o Cobol) o lenguajes nuevos (Go, Scala, Clojure...) que realizan una compilación a código ByteCode, pudiendo incluso usar librerías escritas en otros lenguajes compatibles.

Microsoft a principios de siglo siguio esta filosofía con la plataforma .NET, el lenguaje intermiedo CIL (equivalente a ByteCode) , y con una máquina virtual denominada CLR (Common Language Runtime), siendo los lenguajes principales de esta plataforma los propios de Microsoft: C++, C#, VB, F#), aunque también de terceros como Python o Ruby.

En los últimos años se ha desarrollado diferentes técnicas para la mejora el tiempo de ejecución sobre máquinas virtuales, compilando si es posible en código nativo y por tanto también su ejecución nativa como es la compilación JIT (la primera vez que se ejecuta, se compila el ByteCode a código nativo, la próxima vez se ejecuta en nativo) o AOT (la compilación a nativo se realiza en la instalación, no en la primera ejecución).

Uno de los últimos lenguajes en incorporarse a la familia de lenguajes para la máquina virtual de Java a sido Kotlin (aunque puede funcionar sin ella) Aparece en julio del 2011 en la empresa JetBrains (famosa por sus IDE's). Su sintáxis y semática se han visto ha visto influenciado por Java y Scala principalmente, intentando tener que escribir menos código que su equivalte en Java y añadiendo o mejorando nuevas características usadas en lenguajes más modernos como son las lambdas, la gestión de valores nulos o una segunda vida para el uso de funciones. Google en el año 2017 selecciono Kotlin como lenguaje principal para el desarrollo de aplicaciones para Android.

Este año Google ha anunciado que se centra en la plataforma Kotlin Multiplaform para el desarrollo de aplicaciones multiplataforma (KMP) tanto en cliente como en servidor, dando sorte para:

- Servidor.
- Cliente: Escritorio, iOs, Android.
- Web. En fase de desarrollo.

Se espera un crecimiento significativo del uso del lenguaje en los próximo años, aunque como siempre en el desarrollo del software pueden aparecer nuevas filosofías o modas que hagan que la plataforma sea descontinuada, siendo el ejemplo más claro el competidor directo de KMP dentro de la misma empresa: Flutter.

1.2. Actualidad.

Si bien Kotlin como lenguaje es un producto maduro, no lo es tanto KMP a pesar de tener el apoyo tanto de JetBrains como de Google. Las tecnologías multiplataforma son la gran apuesta de las empresas de software más destacadas, existe cierta incertidumbre en cuanto a su adopción y su uso en un futuro a medio plazo. Muchas de las librerías son versiones alphas y betas, adaptadas de las existentes para Android y otras no se han portado, además, uno de los pilares del desarollo software actual, la web, no está implementado a un nivel aceptable.

Uno de los elementos más destacados de KMP es el compilador, pudiendo generar tanto aplicaciones para JVM incluyendo Android, como aplicaciones nativas para diferentes sistemas (iOS, JS, WebAssembly) gracias a LLVM, que transforma al igual que Java el código de alto nivel a un lenguaje intermedio (IR), y este código IR es optimizado y generando binario para cada plataforma concreta. Su filosofía es tener una gran parte común de la aplicación y una personalización lo más pequeña posible para cada plataforma, generando código nativo que optimice el rendimiento.

A pesar de ello, la arquitecturas, capas, principios y patrones que implementa Kotlin y la plataforma KMP son muy similiares a las que incorporan tecnologías más asentadas como React o Vue en el mundo web.

Evolución Kotlin(https://www.tiobe.com/tiobe-index/kotlin/)

En el verano del 2024, Kotlin es el lenguaje que ocupa la posición 18 según el índice Tiobe, y en GitHub el 15, aunque en ambos casos con una evolución ascendente.

1.3. Características del lenguaje.

Kotlin toma características de diferentes lenguajes, tal como se ha comentado se basa principalmente en Java y Scala, de los que toma la programación orientada a objetos, aunque combinandola con el uso clásico de funciones. Otra de la características destacadas es la de tipado fuerte, contando con un sistema complejo de inferencia de tipos para determinarlos, junto con la inclusión de la programación funcional ampliamente utilizada en los lenguajes más modernos. Existen otras características menos destacadas del lenguaje, las más significativas:

- Caracteristicas de la programación orientada a objetos.
- Variables mutables, inmutables y constantes.
- Programación clasica utilizando funciones.
- Programación funcional.
- Programación asíncrona.
- Anotaciones (precompilación, compilación y tiempo de ejeucución)
- Tipado estático.
- Inferencia de tipos en tiempo de compilación.
- Gestión avanzda de nulos.

- Definición de objetos planos (atributos, getters y setters unicamente).
- Nuevos ámbitos o alcances de código (let, run, with, apply y alo)

Se puede observar que no es un lenguaje que implemente únicamente un paradigma (funcional, OO, estructurado…), sino que es una mezcla de todos ellos, siguiendo con la tendencia de los lenguajes de programación más modernos.

1.4. Ecosistema.

El lenguaje de programación es la base sobre la que se añaden otras herramientas que facilitan la creación, evaluación y mantenimiento de software. Las herramientas principales son el compilador (en caso de lenguajes compilados) y el depurador. Además de las anteriores existen muchas otras que ayudan al desarrollo de software como son los IDE, constructores de código/gestores de proyecto (Gradle, Maven, NPM…), librerías de pruebas, consolas interactivas...

En este punto se describen las herramientas básicas y más destacadas de Kotlin.

1.4.1. Compilador. Línea de comandos.

Elemento imprescindible para la generación de ejecutables, se encuentra disponible para macOS, Linux y Windows, pudiendo compilar tanto para la JVM como para diferentes sistemas de forma **nativa** como es iOS, Android, Linux o Windows.

El compilador se encuentra en [GitHub] (https://github.com/JetBrains/kotlin/releases/), siendo a fecha de hoy la última versión estable la Kotlin 2.0.10. Para instalar lo más sencillo es descargar el fichero zip para la JVM, en la carpeta bin, si se desea también se puede descargar la versión con comandos para la generación de código nativo (incluye kotlinc-native entre otros) . En la versión sin soporte nativo se puede observar que existen diferentes ficheros:

- kotlin: Interprete de código compilado, en ficheros .jar , sirve también para la ejecución de guiones.
- Kotlinc: El compilador a BiteCodes para la JVM, aunque tiene opciones para compilar a otras plataformas como JS o nativa.
- Kotlin-jvm: Personalización del comando anterior con opciones por defecto para JVM.
- Kotlin-js: Compilación del código a Javascript, personalización de Kotlinc
- Kotlin-dce-js: Otra adaptación de Kotlinc para Javascript, que optimiza la generación de código de Kotlin-js.

Por supuesto, si se trabaja desde la línea de comandos es necesario modificar las variables de entorno (como PATH) para poder ejectuar el comando de forma más cómoda.

Ejemplo. Hola mundo.

La extensión de los ficheros en Kotlin son kt, existiendo un punto de entrada, que ha de ser la función main, al igual que en C (en el caso de Java es un método estático de una clase). Si se crea un fichero llamado `Hola.kt` con el siguiente código:

```
fun main() {
    println("Hello, World!")
}
```

A continuación se puede compilar usando alguna de las versiones del compilador:

1. JVM con class

```
$ kotlinc Hola.kt
```

Genera un fichero llamado HolaKt.class, para ejecutarlo:

```
$ kotlin HolaKt.class
```

2. JVM con JAR

```
$ kotlinc Hola.kt  -d Hoja.jar
```

Ejecutando el fichero .jar creado con con:

```
$ kotlinc Hola.jar
```

En el caso de aplicaciones nativas y javascript es necesario realizar algunas configuraciones extras, resolviendo estas configuraciones y opciones de forma sencilla con Gradle.

1.4.2. Consola interactiva.

Kotlin puede ser compilado a código nativo o usar la JVM para su ejecución, al igual que Java con JShell (introducido a partir de la versión 9 de Java), existiendo un intérprete de código, de forma que no sea necesario compilar para probar el código. A estos intérpretes se les denomina bucle de lectura, evaluación e impresión (REPL), en Kotlin para usarlo se usa el mismo comando que para ejecutar un jar o un .class.

REPL

Esta herramienta resulta muy interesante para realizar comprobaciones rápidas o probar conceptos sin necesidad de compilar.

1.4.3. Gradle.

Si bien es posible crear pequeños proyectos usando únicamente la línea de comandos, la gestión de proyectos de tamaño medio y grande se hace demasiado complejo.

Para solucionarlo se dispone de diferentes herramientas, en el mundo Java: Ant, Maven y Gradle, en Python pip o en Javascript npm entre muchos otros. Estas herramientas permiten principalmente la ejecución de instrucciones automatizadas como es la compilación y enlazado, la generación de ejecutables, el empaquetamiento, evaluación de pruebas de forma automática o gestión de repositorios y dependencias de forma sencilla, pudiendo incluso añadir extensiones o crear guiones propios.

En el caso de Kotlin, es posible desarrollar usando Maven, pero la herramienta más utilizada, en especial en el mundo Android, es Gradle. La instalación, configuración y uso de Gradle se ha tratado en el módulo Entornos de Desarrollo de primer curso. Únicamente recordar los comandos más destacados:

1. Init. Inicial el proyecto, aparece un asistente para seleccionar diferentes opciones.

```
$ gradle init
```

2. Tasks. Lista las tareas disponibles para el proyecto.

```
$ gradle tasks
```

Ejemplo gradle tasks

1. Tarea. Ejecuta la tarea indicada

Ejemplo gradle run

1. –refresh-dependencies. Actualiza las dependencias del proyecto.

```
$ gradle --refresh-dependencies
```

Para personalizar y adaptar Gradle con extensiones, nuevas tareas o dependencias se utilizan ficheros de guiones, que pueden estar escritos en Groovy o Kotlin. En las últimas versiones se ha añadido un nuevo fichero para la gestión de las versiones de las dependencias de forma independiente. El fichero es `libs.versions` y se encuentra en la carpeta `gradle` del proyecto.

Es usado por el fichero principal de Gradle, que se encuentra en la raíz de la carpeta app (para proyectos con solo un subproyecto), llamado `build.gradle.kts` (la letra “s” es de script/guión). Se define entre otros:

- Extensiones o plugins.
- Repositorios.
- Dependencias, que toman el fichero libs.versions con la librería y la versión.
- Versión de Java
- Configuración de la aplicación.
- Definición de tareas personalizadas.

```
plugins {
    // Apply the org.jetbrains.kotlin.jvm Plugin to add support for Kotlin.
    alias(libs.plugins.jvm)

    // Apply the application plugin to add support for building a CLI application in Java.
    application
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    // Use the Kotlin JUnit 5 integration.
    testImplementation("org.jetbrains.kotlin:kotlin-test-junit5")

    // Use the JUnit 5 integration.
    testImplementation(libs.junit.jupiter.engine)

    testRuntimeOnly("org.junit.platform:junit-platform-launcher")

    // This dependency is used by the application.
    implementation(libs.guava)
}

// Apply a specific Java toolchain to ease working on different environments.
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(21)
    }
}
```

```
application {
    // Define the main class for the application.
    mainClass = "org.example.AppKt"
}

tasks.named<Test>("test") {
    // Use JUnit Platform for unit tests.
    useJUnitPlatform()
}
```

Por último indicar que tanto el código (carpeta `src`) como la aplicación compilada (`build`) se encuentran en la carpeta app.

> JetBreans está desarrollando un sustituto para Gradle llamado Amper, se espera que en un futuro cercano sea la herramienta de gestión y construcción de proyectos más utilizada para Kotlin, aunque nunca se sabe :see_no_evil:

1.4.4. IDE's.

En principio sería suficiente para poder desarrollar grandes proyectos la utilización de Gradle (**cualquier IDE que gestione Gradle y sus tareas, es suficiente para poder trabajar**), pero para facilitar aún mas el desarrollo, como puede ser el uso de emuladores en el caso de aplicaciones multiplataforma se utilizan IDE's más o menos especializados.

- Viual Studio Code. Soporta Gradle y posee extensiones tanto para Kotlin como para Android.
- IntelliJ de Jetbrains. Especializado en Kotlin y Java. Jetbrains es la empresa principal de la fundación encargada de la gestión y desarrollo de Kotlin. **El alumnado de ciclos formativos en la Comunidad Valenciana tienen acceso mientras esté matriculado a la versión de pago.**
- Android Studio. Versión de IntelliJ adaptado especialmente para el desarrollo móvil y multiplataforma. Cuenta entre otros con la opción de creación y gestión de emuladores de dispositivos móviles.
- Fleet. IDE de Jetrains especializado en Kotlin Multiplataforma, aunque soporta muchos lenguajes. Pretende ser una alternativa a VSC, considerándose un IDE ligero. No es un producto maduro, y actualmente, la indústria no lo está adoptando, a la espera de nuevas versiones más maduras.

1.4.5. Depuración.

El depurador es una herramienta extremadamente útil en el desarrollo de software, imprescindible para conocer qué está sucediendo en el programa a lo largo de su ejecución. Al igual que su hermano Java, Kotlin utiliza `jdb` para depurar. Una vez iniciado el programa se pueden establecer puntos de interrupción/ruptura e ir ejecutando paso a paso o hasta el siguiente, de la misma forma que con cualquier programa Java (u otro lenguaje con depurador). Un ejemplo de uso (teniendo en cuenta que se ha instalado JDK y configurado correctamente):

1. Compilar el programa.

   ```
   $ kotlinc Depurando.kt -d Depurando.jar
   ```

2. Ejecutarlo usando `jdb` (**hay que añadir Kt al segundo parámetro**).

   ```
   $ jdb --classpath Depurando.jar DepurandoKt
   ```

3. Usar los comandos típicos del depurador: `stop`, `run`, `step`, `next`...

El uso de los depuradores en los IDE's es muy similar, ofreciendo los mismos comandos e información, pero de forma gráfica.

1.4.6. Pruebas.

Las pruebas son imprescindibles en la industria del software actualmente, existiendo metodologías que se centran en la creación y superación de las pruebas como elemento central del desarrollo (TDD).

En el caso de Kotlin se dispone de la conocida librería **JUnit Juniper** y otra própia llamada **kotlin.test** con opciones para desarrollo multiplataforma, pero cuyo funcionamiento es muy similar. **Se basa en la utilización de anotaciones y funciones de aserciones.**.

Al crear el proyecto con Gradle se puede seleccionar la librería. Se crea una carpeta en `app/src` llamada `test` en la que se han de crear los casos de prueba. El caso de prueba creado por defecto:

```
import kotlin.test.Test
import kotlin.test.assertNotNull

class AppTest {
    @Test fun appHasAGreeting() {
        val classUnderTest = App()
        assertNotNull(classUnderTest.greeting, "app should have a greeting")
    }
}
```

Se indica que la función `appHasAGreeting` es un test con la anotación `@test`, y en el cuerpo de la misma se utiliza una aserción que evalua el cumplimiento de la condición. Para ejecutar los test simplemente ejecutar la tarea Gradle:

```bash
$ gradle test
```

Otra librería muy utilizada en Kotlin es **Kotest**, diseñado específicamente para Kotlin y cuya filosofía es diferente a JUnit. Se basa en:

- Sintaxis Kotlin.
- Fluen. DSL específico para casos de prueba.
- Pruebas asincrónicas.

Un ejemplo de prueba:

```
class MyFirstTestClass : FunSpec({

   test("my first test") {
      1 + 2 shouldBe 3
   }

})
```

En la documentación del proyecto se puede profundizar en las características de este framework de pruebas.

2. Lenguaje (Sintaxis/Semántica)

2.1. Variables

En Kotlin se usa el formato `camelCase` para el nombre de las variables (como se hace en Java) y se dividen en: **mutables**, **inmutables** y **constantes**.

Si se quiere crear una variable cuyo valor pueda cambiar a lo largo del tiempo se tiene que usar la palabra reservada `var` (variable **mutable**).

Si lo que se quiere es crear una variable cuyo valor no pueda cambiar a lo largo de la ejecución del programa se tiene que usar la palabra reservada `val` (variable **inmutable**).

Por último, si lo que se quiere es crear una constante en tiempo de compilación entonces se tiene que hacer usando el modificador `const` delante de `val`. Para el nombre de las constantes se usa el formato `UNDER_SCORE` (en mayúsculas).

En todos los casos, después del nombre de la variable, y del carácter dos puntos, se tiene que indicar el tipo de la variable, como se ve en los siguientes ejemplos:

```
// myName es una variable de tipo "cadena de texto" (string)
val myName: String = "Alice"

// myAge es una variable inmutable de tipo entero.
var myAge: Int = 20

// PI es una constante en tiempo de compilación.
const val PI: Float = 3.14

// Esto generaría un error porque myName no puede cambiar de valor, es
// inmutable:
//myName = "Bob"

// Sí puedes cambiar una variable mutable como myAge
myAge = 21

// Tampoco puedes cambiar el valor de una constante... nada nuevo. Esto daría
// un error:
//PI = 3.15
```

2.2. Tipos de datos

Antes de continuar, **algo muy importante acerca de las variables en Kotlin**: todo es un objeto. ¿Qué quiere decir? Todas las variables pueden acceder a métodos concretos del tipo de dato al que pertenece.

Los tipos básicos en Kotlin son:

- Números: documentación oficial
- Booleanos: documentacón oficial
- Caracteres: documentación oficial
- Cadenas de caracteres o *strings*: documentación oficial

En esta tabla se hace un resumen de los tipos básicos en Kotlin:

Categoría	Tipos básicos	Ejemplo de código
Enteros	Byte, Short, Int, Long	val year: Int = 2024
Enteros sin signo	UByte, UShort, UInt, ULong	val score: UInt = 100u
Reales	Float, Double	val temp: Float = 24.5f
Booleanos	Boolean	val isEnabled: Boolean = true
Caracteres	Char	val separator: Char = ‘,’
Strings	String	val course: String = “DAM”

2.3. Inferencia de tipos

La **inferencia de tipos** quiere decir que no hay obligación de indicar el tipo de la variable ya que, el compilador de Kotlin, puede deducirlo.

Esto será así en los casos más sencillos y en general. Pero, en ocasiones se estará obligado a indicar el tipo de la variable como, por ejemplo:

- Cuando se declara una variable y no se inicializa.
- Hay casos en que Kotlin no será capaz de deducir el tipo de una variable (se verá más adelante, sobre todo cuando los programas a escribir crezcan en complejidad)
- A veces, simplemente, para mejorar la legibilidad y claridad de los programas.

En estos ejemplos se puede ver cómo funciona la inferencia de tipos:

```
// score es una variable de tipo Int
var score = 20

// name es una variable de tipo String
var name = "Alice"

// mark es una variable de tipo Double
var mark = 8.75

// otherMark es una variablede tipo Float
// Fíjate que para que sea Float tienes que añadir el modificar 'f'
var otherMark = 9.5f

// newScore es una variable de tipo Int porque es el resultado de multiplicar
// dos Int: una variable de tipo Int y un literal de tipo Int
var newScore = score * 5
```

2.4. Strings literales

En Kotline se tienen dos tipos de literales de `String`: *Escaped Strings* y *Multiline Strings*.

2.4.1. Escaped strings

Estos `String` se denominan así porque se pueden incluir caracteres de escape (nada nuevo en relación a Java). En este enlace se tienen los caracteres de escape. Por ejemplo:

```
val s = "Primera línea.\nSegunda línea.\nTercera línea."

print(s)
```

Al ejecutar este programa se obtiene esta salida:

```
Primera línea.
Segunda línea.
Tercear línea.
```

2.4.2. Multiline strings

Estos `String` usan como delimitadores de la cadena tres comillas (`"`) en vez de una:

```
val s = """
    Primera línea.
    Segunda línea.
    Tercear línea.
"""

print(s)
```

Al ejecutar este programa se obtiene una salida muy parecida a la anterior:

```
    Primera línea.
    Segunda línea.
    Tercear línea.
```

La diferencia está en que se han añadido los espacios delante de cada línea. Si se quiere eliminar esos espacios hay que usar un "truco":

```
val s = """
    |Primera línea.
    |Segunda línea.
    |Tercear línea.
    """.trimMargin()

print(s)
```

Ahora sí, se obtiene la salida buscada:

```
Primera línea.
Segunda línea.
Tercear línea.
```

Lo que hace el método `trimMargin()` del tipo `String` es usar como margen izquierdo el carácter `|` (que es el caracter por defecto). Se podría usar otro caracter, en cuyo caso habría que indicarlo como argumento del método `trimMargin()`. En el siguiente ejemplo se usa el carácter `>` como posición del margen izquierdo:

```
val s = """
    >Primera línea.
    >Segunda línea.
    >Tercear línea.
    """.trimMargin(">")

print(s)
```

2.5. Operadores

Los operadores se pueden resumir en: matemáticos, de asignación, lógicos, de comparación, de rango, indexación y de seguridad sobre nulos.

En los siguientes apartados se describen y recuerdan aquellos similares o iguales a Java, y se entra en el detalle de aquellos operadores más complejos o que no existen en Java.

2.5.1. Matemáticos

Son exactamente los que se usan en Java, así que se nombran sin más:

- Suma: `+`
- Resta: `-`
- Multiplicación: `*`
- División: `/`
- Módulo: `%`

Además, también se tienen los operadores `++` y `--` que se usan para incrementar y decrementar, respectivamente, el valor de un número entero en uno.

2.5.2. De asignación

No hay novedades con respecto a Java: `=`.

También se tienen los operadores matemáticos que asignan el resultado al operando de la izquierda: `+=`, `-=`, `*=`, `/=` y `%=`.

En cualquier caso, conviene recordar que todos estos operadores asignan el valor de una expresión a la variable que se pone a la izquierda del mismo.

2.5.3. Lógicos y de comparación

De nuevo, no hay novedades con respecto a Java. Hay que recordar que estos operadores devuelven en todos los casos un resultado `Boolean` (`true` o `false`).

- Y lógico: `&&`
- O lógico: `||`
- Negación: `!`
- Igualdad y desigualdad: `==` y `!=`
- Menor y menor o igual: `<` y `<=`
- Mayor o mayor o igual: `>` y `>=`

Y a estos hay que sumar, en Kotlin, los **operadores de igualdad y desigualdad referencial**: === y !==. Y se ha comentado, y conviene recordar, que todas las variables en Kotlin son objetos. Así, dos objetos son iguales (==) si tienen el mismo valor y son referencialmente iguales (===) si son el mismo objeto.

Aquí puedes ver un ejemplo comentado donde se puede ver la diferencia entre == y === (lo mismo aplica para != y !==):

```
// Se usa la clase Integer para crear un objeto de tipo entero
var aliceAge = Integer(20)
var bobAge = Integer(20)

// Operador igualdad: los dos objetos tienen el mismo valor -> imprime true
print(aliceAge == bobAge)

// Operador de igualdad referencial: son objetos diferentes -> imprime false
print(aliceAge === bobAge)
```

2.5.4. De rango

Antes de seguir conviene aclarar **qué es un rango**. Se trata del **concepto matemático**, tal cual, en el que se define un **conjunto de elementos a través de sus límites inferior y superior**. Veamos ejemplos matemáticos con su explicación:

- `[1 - 5]` -> se trata del conjunto matemático de los números que van desde el 1 hata el 5, ambos incluidos.
- `(1 - 5]` -> mismo conjunto que el caso anterior pero sin incluir el número 1, es decir: 2, 3, 4 y 5.
- `[1 - 5)` -> mismo conjunto que en el primer caso pero sin incluir el número 5, es decir: 1, 2, 3 y 4.
- `(1 - 5)` -> mismo conjunto que en el primer caso pero sin incluir los número 1 y 5, es decir: 2, 3 y 4.

En Kotlin se usa una sintaxis diferente. El operador básico es `..` y, a continuación, tienes una serie de ejemplos:

```
var from1to5 = 1..5
var englishAlphabet = "a".."z"
```

Estos operadores se usan habitualmente en las estructuras de control y operadores lógicos y relacionales. Se deja, para cuando se estudien estas estructuras, los detalles de estos operadores.

2.6. Tipos nullables

Al indicar el tipo de una variable en Kotlin podemos especificar que dicha variable puede ser `null`, añadiendo el modificador ? al final del tipo. Por ejemplo:

```
var name: String? = "Alice"
var age: Int? = 20
```

Las variables `name` y `age` podrían tener valores nulos. La buena noticia es que en Kotlin se pueden usar estructuras del lenguaje para trabajar con variables que pueden tener valor `null`. Se ven en el siguiente apartado.

2.7. Nulos

No se puede olvidar que, en Kotlin, todas las variables son objetos y, como tal, pueden tener un valor nulo denominado `null`.

No es un concepto o idea nueva si se viene de programar en Java, por ejemplo. Lo que sí es novedoso en Kotlin es el tratamiento y los operadores de seguridad que tienen en cuenta la posibilidad de que una variable sea nula.

En Kotlin hay dos operadores básicos con los que hay que familiarizarse:

- Operador de llamada seguro (*safe call operator*): `?.`
- Operador *elvis*: `?:`

El operador de llamada seguro `?.` se usa para hacer llamadas a métodos de un objeto pero Kotlin solo invocará a dicho método si dicho objeto es no nulo. En otro caso devuelve `null`.

El operador elvis `?:` devuelve el valor del objeto que se indica a la izquierda si es no nulo o el valor que se pone a la derecha.

Con unos ejemplos comentados se debería entender:

```
var name: String? = "Alice"

// El valor de la variable "l" es 5 (la longitud del string)
var l = name?.length

var otherName: String? = null

// El valor de "otherL" es null porque no se hace la llamada a length al ser el objeto null
var otherL = otherName?.length
```

En Java se tienen que usar estructuras condicionales para averiguar si una variable es `null` antes de usarla. En Kotlin, gracias al

operador ?., el código es más compacto.

3. Estructuras de control

Como todos los lenguajes estructurados Kotlin posee estructuras de control que modifican el flujo de ejecución del código. La base de estas estructuruas de control son similares a las de los lenguajes más populares como Java, C/C++ o C#, aunque con algunas características propias.

3.1 Estructuras condicionales

Permiten ejecutar en función de la evaluación de una condición/predicado ciertos fragmentos/bloques de código y no otros. Se tienen expresiones simples, alternativas, encadenadas, anidades y múltiples.

3.1.1 Selección simple/alternativa

La gran diferencia con otros lenguajes es que esta expresión devuelve un valor. Implica que se puede realizar una asignación de la evaluación, **sustituyendo a operador ternario de otros lenguajes**.

La sintaxis de la selección simple:

```
 if (predicado){
    bloque que se ejecuta si predicado es cierto
  }
```

En caso de la alternativa:

```
if (predicado){
  bloque que se ejecuta si predicado es cierto
}
else{
   bloque que se ejecuta si predicado es falso
}
```

Si el bloque contiene una única línea, se pueden odbiar las llaves.

Para realizar una asignación condicional:

```
var condition=45.0f
condition= if (predicado) 7.8f else 2.9f
```

Algunos ejemplos extraidos de la documentación oficial de Kotlin:

```
fun main() {
    val a = 2        //se realiza una inferencia de tipos para las 3 variables en tiempo de compilación, declarándos
    val b = 3

    var max = a
    if (a < b) max = b  //sentencia simple, el predicado es a<b, es cierto, por tanto se ejecuta max=b

    // sentencia alternativa, se ejecuta max=b
    if (a > b) {
        max = a
    } else {
        max = b
    }

    // uso como ternaria de otros lenguajes
    max = if (a > b) a else b
}
```

Se pueden combinar la asignación con la ejecución de código condicional, colocando como **última instrucción el bloque** el nombre de la variable/expresión/valor que se devolverá.

```
  var a=10
  var b=15

  val max = if (a > b) {
      print("Choose a")
      a
  } else {
      print("Choose b\n")
      b/3
      //si se tiene más instrucciones, la asignación no funciona
```

```
        //println("No funciona\n")
    }
    println(max)
```

3.1.2. Selección encadenada, anidada

En este tipo de estructuras de control sigue la misma sintaxis que en Java.

Para las estructuras de selección encadenadas a continuación del else se vuelve a introducir la instrucción if-else, ya que se considera un bloque único de código.

```
    if (a > b)
        print("Opcion 1")
    else if (a / 2 > b)
        print("Opcion 2")
    else if (a / 2 < b)
        print("Opcion 3")
    else
        print("Opcion 4")
```

Las estructuras anidadas consisten en incluir dentro del bloque if o else nuevos bloques que incluyen sentencias de selección. Un ejemplo:

```
var temperature = 10
var snowing=false
if (temperature > 15) {
    if (temperature > 25) {
// Si la temperatura es mayor que 25 ...
        println("A la playa!!!")
    } else {
        println("A la montaña!!!")
    }
} else if (temperature < 5) {
    if (snowing) {
        println("A esquiar!!!")
    }
} else {
    println("A descansar... zZz")
}
```

3.1.3. Selección múltiple

La lectura y gestión de código con if-else encadenados hace que el código sea poco legible, en caso de que se tenga que evaluar una variable/expresión y se tengan que ejecutar diferentes bloques en función del valor se dispone de la sentencia de selección múltiple conocida con WHEN. En caso de usarlo como asignación el valor devuelto es la última instrucción del bucle, al igual que con if. Si se utliza de forma clásica, es obligatorio definir un else final, equivalente a default en Java.

La sintásis de la instrucción WHEN:

```
 when ({variable|expresión|método}){
     {value1,value2...| método | función | [!]in valor_inicial..valor_final| [!]is type} -> bloque si se cumple
      {<value1,value2...| método | función | [!]in valor_inicial..valor_final| [!]is type} -> bloque si se cumple
          ...
      else-> bloque en caso de no cumplirse ninguna opción
 }
```

- { } argumento requerido, pero solo una opción. - | opción. - ... elementos que se pueden repetir - [ ] parámetro opcional. - ! negación.

Se pueden combinar en un mismo bloque when diferentes condiciones, un ejemplo:

```
var example = 10
when (example + 1) {
    0, 1 -> println("Examen1")
    in 0..4 -> println("Examen2")
    is Int -> println("Examen3")
    "11".toInt() -> println("Examen4")
    else -> println("Examen5")
}
```

Solo se ejecuta la primera condición que sea cierta, si existen otras que son ciertas no se ejecutan.

Un ejemplo similar con asignación:

```
var example = 10
```

```
    var result=when (example + 1) {
        0, 1 -> println("Examen1")
        in 0..15 -> println("Examen2")
        !is Int -> println("Examen3")
        "11".toInt() -> println("Examen4")
        else -> println("Examen5")
    }
    println(result)
```

La documentación oficial de WHEN.

3.2 Bucles for

Los bucles for son algo distintos a los clásicos de Java o C++, entre otros, más parecidos a Scala o Python. En estos bucles se recorre un vector, colección o secuencia de elementos, es decir posea un iterador. La sintaxís del bucle:

```
for (element in elements){
     print(element)
}
```

Al igual que if, si el bloque posee una única instrucción, las llaves no son necesarias. Para utilizar le bucle for de forma similar a como se hace en C u otro lenguaje derivado, se ha de crear una secuencia de elementos, la forma más sencilla es indicando el límite inferior la secuencia, a continuación punto punto y finalizar con el límite superior. La secuencia se puede definir en una variable o en la definición del bucle. Un ejemplo sencillo en el que se puede ver recorrer un vector, una variable de tipo secuencia y una secuencia dentro de un bucle:

```
    val vector= intArrayOf(1,2,3,4,5,6,7,8,9,10)
    val secuence= 1..10

    for(item in vector){
        println(item)
    }
    for (x in secuence) {
        println(x)
    }
    for(i in 1..10) {
        println(i)
    }
```

Para simular la última expresión de los bucles clásicos en Java se puede indicar en la creación de la secuencia si es creciente o decreciente y en incremento/decremento.

```
     val secuence= superiorLimit downTo inferiorLimit step decrement
     for ( i in secuence){
     ...
     }
     for ( i in superiorLimit downTo inferiorLimit step decrement){
     ...
     }
```

Otra de las posibilidades que ofrece el bucle for es la de poder obtener el índice en el que se está iterando, pudiendo añadir además el contenido del índice.

```
    val vector= intArrayOf(1,2,3,4,5,6,7,8,9,10)
    val secuence= 30 downTo 10 step 3

    for(index in vector.indices){
        println("["+index+"]="+vector.get(index))
    }
    for (( index,value) in secuence.withIndex()) {
        println("["+index+"]="+value)
    }
```

En la documentación oficial se pueden profundizar en el bucle for.

3.3 Bucles while

En este caso, los bucles while y do-while son similares a los del resto de lenguajes, se tiene un predicado más o menos complejo que se evalua a cierto/falso, un falso provoca salir del bucle, y en el caso del do-while se evalua al final, con lo que se ejecuta **siempre al menos una vez**. Un par de ejemplos extraidos de la documentación oficial:

```
     while (x > 0) {
         x--
```

```
    }

    do {
        val y = retrieveData()
    } while (y != null) // y is visible here!
```

Las instrucciones break y continue no se tratan, ya que están desaconsejadas, ya que dificultan la comprensión, modicación y depuración del código.

4. Funciones

Existen tareas que se repiten una y otra vez en diferentes puntos del programa, además son tareas que se pueden utilizar en diferentes programas. Reescribir una y otra vez el mismo código parece una tarea un poco inútil, que puede facilitar que se comentan errores y que dificulta el mantenimiento del código. Para solucionarlo se define el concepto de función que permite crear código reutilizable y ser usado en diferentes lugares. El concepto se hereda de la funciones matemáticas, por ejemplo coseno, ∑ o ∏, y siguiendo con la analogía, una función en programación tiene:

- Un nombre, por ejemplo sumatorio.
- Un conjunto de parámetros, en la definición llamados formales, que tiene un tipo y un nombre que los identifica para su posterior uso, en el sumatorio puede ser un vector de enteros a sumar. Al utilizarse esos parámetros se denominan actuales o reales.
- Un cuerpo, instrucciones que realizan la acción o el cálculo y que utilizan los parámetros pasados a la función.
- Un valor de retorno. Al igual que una función matemática, una función en programación ha de devolver algún valor, es necesario en los lenguajes tipados indicar el tipo de dato que devuelve la función. En el sumatorio el valor devuelto será la suma y el tipo de retorno un entero o entero largo.

Las funciones han vuelto a utilizarse después de unas décadas donde la POO parecía haberlas arriconado a lenguajes relativamente antiguos como C. Frameworks tan ampliamente conocidos como React utilizan las funciones como base.

En el caso de Kotlin se han añadido características que concuerdan con los lenguajes más modernos (también en los métodos de las clases, que se definen como funciones) como los valores por defecto o realizar llamadas indicando el nombre del argumento y el parámetro, sin necesidad de mantener el orden de la definición.

4.1. Declaración y uso de funciones

4.1.1. Declaración

Para indicar que se declara una función se usa la palabra reservada **fun** el nombre de la función, se abre un paréntesis, a continuación se indican los argumentos o parámetros formales con un nombre y su tipo asociado, se cierra el parentesís, dos puntos indicando el tipo de dato devuelto (si es nulo, no es necesario indicarlo) y a continuación entre llaves el bloque de código de la función.

```
fun functionSample(){
    println("Hello World!")
}
```

Para llamar a la función simplemente usar el nombre, para la función anterior:

```
    println(functionSample())
```

4.1.2. Valor devuelto

El concepto matemático de función devuelve un valor, de igual forma en programación una función devuelve un valor. Se ha de indicar el tipo del valor devuelto, en este caso después del paréntesis de fin de los parámetros, seguido de dos puntos y el tipo devuelto. En caso de no devolver ningún valor, el tipo de dato es **Unit** u odviarse el valor devuelto. Para indicar el valor devuelto, se usa la palabra reservada **return**, una vez ejecutado el return, se deja de ejecutar la función, en caso de tener instrucciones a continuación no se ejecutan.

Ejemplo de función que devuelve un entero:

```
fun functionSample():Int{
    var a=10
    a++
    return a
}
```

Ejemplo de funciones que no devuelven ningún valor:

```
fun functionSambleUnit():Unit{
    println(" no devuelve nada")
}

fun functionSambleUnit2(){
```

```
    println(" no devuelve nada")
}
```

4.1.3. Parámetros/argumentos

Es posible que la función necesite de datos para realizar su trabajo, por ejemplo, para indicar si un número es primo o no, se necesita conocer el número, a los datos que necesita una función para hacer su trabajo se le llama **parámetro**. Se ha de diferenciar dos tipos de parámetros en las funciones:

- Los parámetros formales son aquellos que se establecen en la definición de la función, estos parámetros han de posee un tipo de dato (primitivo/básico o compuesto) y un nombre, que será utilizado en el interior de la función.
- Los parámetros reales son aquellos que se facilitan a la función en cada llamada. También conocidos como argumentos. **No es necesario que el nombre coincida.**

La declaración de los parámetros formales se realiza dentro del paréntesis de la definición de la función, y en el cuerpo de esta, se hará referencia con el nombre del parámetro formal, cada parámetro se separa por “,”. Indicar que al igual que Java, en Kotlin los tipos primitivos/básicos se pasan por valor y en el caso concreto de Kotlin **son val**, y los objetos por referencia (no es exactamente así, pero se comporta igual), es decir los tipos primitivos/basícos no se pueden modificar, además si se pudiera las modificaciones no se verán fuera de la función, en cambio cuando se pasa por referencia, las modificaciones realizadas dentro de la función son visibles desde el exterior. Un ejemplo:

```
    fun function_sample( parametroPrimitivo:Int, parametroObjeto:Coordinate){
        parametroObjeto.x=66
        //fallo de compilación, los valores primitivos/básicos son de tipo "val" al pasarlo por valor
        //parametroPrimitivo=33
        println("DENTRO DE LA FUNCIÓN P1 vale $parametroPrimitivo  P2 vale$parametroObjeto")

    }
```

Al llamar a la función:

```
    var coordenada1=Coordinate(4,5)
    functionSample(3,coordenada1)
    println("Al salir de la funcion vales $coordenada1")
```

El valor de x después de la llamada tiene el valor de 66.

4.1.3.1 Nombre de los parámetros y valores por defecto

Los nombre de los parámetros se utilizan en el cuerpo de la función para referirse a ellos, ya sea para consultar su valor o modificarlos. Una de las características más destacadas de Kotlin es que permite definir valores por defecto de los parámetros, de forma que si al llamarse la función no se les da valor, toman el por defecto. También se tiene la característica de que no es necesario respetar el orden de definición de los parámetros, siempre y cuando se indique el nombre del parámetro y su valor. Estas dos características simplifican el código ya que no es necesario duplicar funciones/métodos y se puede adaptar más o menos las funciones dependiendo de las necesidades. Es especialmente útil en los constructors de clases.

```
fun functionExampleDefaultValue(a:Int=5,b:String,c:Coordinate):Unit{
    println("a -> $a")
    println("b -> $b")
    println("c -> $c")
}
```

Algunos ejemplos de llamadas a la función anterior:

```
    //llamando con todos los parámetros, sin nombre
    functionExampleDefaultValue(4,"Hola mundo",Coordinate(5,5))
    //llamando sin a, pero con nombre
    functionExampleDefaultValue(b="Hola luna",c=Coordinate(5,5))
    //cambiando el orden de los parámetros
    functionExampleDefaultValue(c=Coordinate(5,5),a=54, b="Hola luna")
```

4.1.3.2 Parámetros de longitud variable

Un tipo especial de parámetro es **vargargs**, se está indicando que la función tiene un número indeterminado de parámetros (es similar a pasar un vector/array). El uso clásico es la de definir comandos con parámetros variables a ejecutar desde el terminal.

```
fun functionVargars(vararg params:String){
    for (param in params){
        println(param)
    }
```

```
}
fun functionVargarsInt(vararg params:Int){
    for (param in params){
        println(param)
    }

}
```

Para llamar a las funciones se definen arrays, usando el * (paso por referencia, con punteros, al estilo C).

```
    val array_string= arrayOf("a","b","c","d","e","f","g","h","i")
    functionVargars(*array_string)
    val array_int= intArrayOf(1, 2, 3)
    functionVargarsInt(*array_int)
```

4.1.4. Función principal

Todo programa tiene un punto de inicio y a partir de este se comienza la ejecución del resto de mismo. En el caso de Java es un método estático de una clase, en el caso de Kotlin una función llamada main (tomado de entre otros lenguajes C) (pueden existir varios main en un programa, aunque totalmente desaconsejado, indicando desde línea de comandos o fichero Gradle, cúal de ellos es el que se tiene que iniciar).

La función main suele tener un único argumento (depende del lenguaje) de longitud variables (vararg) de tipo String, que son los parámetros de los comandos clásicos, como ls -la. En otros lenguajes (C,C++) es posible que main devuelva un entero para indicar si se ha ejecutado o no correctamente el programa. En los lenguajes basados en VM, el método main no suele devolver ningún valor. Un ejemplo de paso de parámetros de longitud variable:

```
fun main(vararg args: String) {
    if (args.isEmpty()) {
        println("No se han proporcionado argumentos.")
        return
    }

    // Analizando los parámetros
    for (arg in args) {
        when {
            arg.matches(Regex("-\\w+")) -> println("Opción: $arg")
            arg.matches(Regex("\\d+")) -> println("Número: $arg")
            else -> println("Texto: $arg")
        }
    }
}
```

La llamada y la salida por consola:

```
    $ kotlin MainKt -v 123 hello -o
    Opción: -v
    Número: 123
    Texto: hello
    Opción: -o
```

4.1.5. Variables como funciones/funciones como parámetro

Las variables no solo pueden almacenan datos (que al fin y al cabo son direcciones de memoria), sino también funciones, más concretamente direcciones de memoria en los que se encuentra una función, **siempre y cuando el número, orden, tipo de parámetros y tipo devuelto sean el mismo**.

Para definir una variable de tipo función se definen los parámetros y el valor devuelto, para su asignació, simplemente se da el nombre de la función a la que apuntará, y para llamar a la función apuntada por la variable, se sustituye el nombre la función por el nombre de la variable. **En el caso de variables a funciones no se pueden usar los valores por defectos, ni cambiar el orden de los paráemtros, ya que el compilador no los puede inferir.**

Ejemplo:

```
    fun functionExampleDefaultValue(a:Int=5,b:String,c:Coordinate):Unit{
        println("a -> $a")
        println("b -> $b")
        println("c -> $c")
    }
```

```
    fun main() {
      val f:(Int,String,Coordinate)->Unit?
        f=::functionExampleDefaultValue
        f(4,"ejemplo",Coordinate(5,5))
    }
    data class Coordinate(var x:Int,var y:Int)
```

De igual forma es posible definir funciones que reciban como parámetros variables que apunten a funciones:

```
fun main() {
    //variable de tipo función con 2 parámetros que devuelve vacio
    var f:(Int, String)->Unit?
    //se asocia la variable a la función
    f=::funcioncompoparametro
    //se pasa como parametro la variable de tipo función
    funcionejemplo(1,f)
    //mismo ejemplo pero pasando directamente
    funcionejemplo(5,::funcioncompoparametro)
}

//función que recibe un parámetro que es una función
fun funcionejemplo(parametro1:Int,parametro2:(a:Int,b:String )->Unit){
    println("Se pasa el parametro1: $parametro1 ")
    parametro2(parametro1,parametro1.toString());
}

//funcion normal
fun funcioncompoparametro(a:Int,b:String):Unit{
    println ("\tsoy una funcion pasada como parámetro $a $b\n\n")
}
```

La salida:

```
$ Se pasa el parametro1: 1
    soy una funcion pasada como parámetro 1 1
$ Se pasa el parametro1: 5
    soy una funcion pasada como parámetro 5 5
```

4.2. Ámbito o alcance

El ámbio o alcance define las regiones del código donde los objetos, clases, funciones o variables son accesibles (se pueden modificar/ consultar en el caso de variables, o llamar a una función. Kotlin tine la característica de poder definir funciones dentro de funciones, que solo se puede llamar dentro del bloque de la función que la contiene, siendo este, por tanto su ámbito o alcance. Un ejemplo de uso:

```
    fun functionFirsLevel(c:Coordinate) {
        var d = Coordinate(7,5)

        fun functionSecondLevel(f:Coordinate) {
           f.x=f.x+d.x
            f.y=f.y+d.y
        }

        functionSecondLevel(c)
    }

    fun main() {
        var coordinate=Coordinate(1,1)
        functionFirsLevel(coordinate)
        //se encuentra fuera del ámbito, o es accesible ERROR de compilación
        //functionSecondLevel(coordinate)
        println(coordinate)

    }

    data class Coordinate(var x:Int,var y:Int)
```

4.3. Funciones genéricas

En la programación orientada a objetos, la generalización es ampliamente utilizada, el ejemplo clásico son las pilas, listas o colas

(colecciones en generales), su funcionamiento es el mismo independientemente de lo que almacenan (personas, tomates, naves espaciales...), pero se necesita conocer que tipo de clase almacenará en tiempo de compilación.

En Kotlin esta generalización se puede definir también en las funciones, de forma que se puede no especificar el tipo de dato en la definición de la función, pudiendo utilizarla por tanto con diferentes tipos de datos.

Un ejemplo de generalización:

- Se establecen 2 parámetros genericos.
- T que ha de implementar la interfaz comparable.
- T2 que ha de implementar la clase serializable.
- Devuelve un objeto de tipo T.

```
fun <T: Comparable<T>, T2: Serializable> funtionGeric(item1: T, item2: T2): T {
    return item1
}

fun main() {
    var coordinate1 = Coordinate(1, 1)
    var coordinate2 = Coordinate(2, 2)
    var coordinate3 = funtionGeric(coordinate1, coordinate2)
    println(coordinate3)
}

data class Coordinate(var x: Int, var y: Int) : Comparable<Coordinate>, Serializable {
    override fun compareTo(other: Coordinate): Int {
        return this.y.compareTo(other.y)
    }
}
```

4.4. Funciones en línea

La llamada de funciones en el código conllevan un consumo de tiempo y recursos, se ha de guardar el estado de los registros en la pila, cambiar el contador de programa a la dirección de memoria de la llamada, cuando finaliza se ha de desapilar la información apilada para continuar con la ejecución. Además en el caso de ejecutarse sobre una máquina virtual, el coste es mayor.

En ocasiones existen funciones que se han de ejecutar en el menor tiempo posible, para ello se definen las funciones en línea o “inline”, el compilador sustituye cada llamada a la función por **el código del cuerpo de la función**. De esta forma no forma se pierde el tiempo en realizar la llamada y volver al estado de ejecución previo a la llamada. Tiene la desventaja de generar programás más grande, por lo que se han de utilizar en:

```
-Funciones pequeñas.
-Necesidad de ejecución lo más rápida posible.
```

Se puede ver su uso en le siguiente ejemplo:

```
    inline fun funcionEnLinea(x:Int,y:Int):Int{
        return x+y
    }
```

Se dejan algunas caracteristicas sin tratar como la notación infija y la [optimización de funciones recursivas] (https://kotlinlang.org/docs/functions.html#tail-recursive-functions)

5. Programación Orientada a Objetos

En Kotlin se puede usar el paradigma de Programación Orientada a Objetos. Hay que recordar que la base de este paradigma son las denominadas **clases** que se usan para crear **objetos**.

5.1. Clases y atributos

Para crear una clase en Kotlin se usa la palabra reservada `class`. En el siguiente ejemplo se crea una clase llamada `Person` con tres atributos públicos: dos de tipo `String` y uno de tipo `Int`:

```
class Person {
    var name = ""
    var lastName = ""
    var age = 0
}
```

Los atributos son `public` por defecto y, como se ve en el ejemplo, hay que inicializarlos.

Las clases son simples planos que se pueden usar para crear objetos. En el siguiente ejemplo se puede ver cómo se usa la clase `Person` para crear un objeto y cómo se puede acceder a sus atributos:

```
var alice = Person()

alice.name = "Alice"
alice.lastName = "Ecila"
alice.age = 20

print("Soy ${alice.name} ${alice.lastName} y tengo ${alice.age} años")
```

5.2. Constructor principal

En el ejemplo anterior se ha usado el **constructor por defecto** ya que no se había declarado ningún constructor.

En Kotlin podemos añadir un constructor principal y constructores secundarios. Para añadir constructores secundarios se tiene que haber declarado el principal.

El **constructor principal** es declarado en la cabecera de la clase:

```
class Car(var brand: String, var model: String, var cv: Int, var age: Int)
```

Varias cosas a tener en cuenta antes de continuar:

- Las variables indicadas en el constructor son atributos de la clase
- Estos atributos, por defecto, son públicos
- Se pueden añadir valores por defecto a los atributos como en las funciones

Para crear objetos del tipo `Car`:

```
var hi10 = Car("Hyundai", "i10", 64, 3)

// No puedes crear coches "vacíos". Esto daría error porque hay un
// constructor principal que hay que usar y que espera 3 argumentos:
//var myCar = Car()

print("Mi coche es un ${hi10.brand} ${hi10.model} con ${hi10.cv} caballos")
print("Además, este coche tiene una antigüedad de ${hi10.age} años")
```

Llegados aquí, cualquier persona acostumbrada a la programación orientada a objetos y que desconozca el lenguaje Kotlin se preguntará: ¿dónde va el código de inicialización? En Java, por ejemplo, va dentro del cuerpo del constructor pero en Kotlin, como se ve en el ejemplo, el constructor principal no tiene cuerpo.

La respuesta está en el denominado **bloque de inicialización** en el que se usa la construcción del lenguaje `init`:

```
class Car(var brand: String, var model: String, var cv: Int, var age: Int) {

    val itvFrequency: Int

    init {
        when (age) {
            in 1..4 -> itvFrequency = 4
            in 5..9 -> itvFrequency = 2
            else -> itvFrequency = 1
        }
    }
}
```

En este último ejemplo hay varias novedades:

- En el constructor principal se declaran 4 atributos públicos y mutables
- Se ha declarado, además, un quinto atributo público e inmutable llamado `itvFrequency`
- Se pueden crear atributos a través del constructor principal y dentro del cuerpo de la clase
- Se usa un bloque `init` para asignar un valor al atributo `itvFrequency`

Por último, cabe señalar que se pueden introducir tantos bloques `init` como se quiera.

5.3. Constructores secundarios

Si se necesitara más constructores, Kotlin ofrece la posibilidad de añadir los denominados **constructores secundarios**.

Se usa la palabra reservada `constructor` como se ve en el siguiente ejemplo:

```
class Car(var brand: String, var model: String, var cv: Int, var age: Int) {

    val itvFrequency: Int

    init {
```

```
        when (age) {
            in 1..4 -> itvFrequency = 4
            in 5..9 -> itvFrequency = 2
            else -> itvFrequency = 1
        }
    }

    constructor(brand: String, model: String, cv: Int) : this(brand, model, cv, 0) {
        println("Se ha llamado al constructor principal")
    }
}
```

Varias cosas a señalar de este ejemplo:

- Los constructores secundarios usan el constructor principal
- Los constructores secundarios pueden tener cuerpo (en este ejemplo solo se imprime por pantalla un mensaje)

5.4. Métodos

A modo de recordatorio: si los **atributos** definen el **estado de un objeto**, los **métodos definen el comportamiento**. Los atributos y los métodos se denominan **miembros de una clase**.

En el ejemplo se han añadido varios métodos a la clase `Car` a modo de ilustración:

```
class Car(var brand: String, var model: String, var cv: Int, var age: Int) {

    val itvFrequency: Int

    init {
        when (age) {
            in 1..4 -> itvFrequency = 4
            in 5..9 -> itvFrequency = 2
            else -> itvFrequency = 1
        }
    }

    constructor(brand: String, model: String, cv: Int) : this(brand, model, cv, 0) {
        println("Se ha llamado al constructor principal")
    }

    fun printDetails() {
        println("Brand: $brand, Model: $model, CV: $cv, Age: $age years, ITV Frequency: $itvFrequency years")
    }

    fun increasePower(increase: Int) {
        cv += increase
        println("The power has been increased by $increase CV. New power: $cv CV")
    }

    fun calculateDepreciation(rate: Double = 0.15, currentValue: Double = 10000.0): Double {
        return currentValue * Math.pow(1 - depreciationRate, age.toDouble())
    }
}
```

5.5. Modificadores de visibilidad

Hasta ahora todos los miembros de una clase (atributos y métodos) eran `public` porque no se ha especificado nada de forma explícita y, por defecto, todos los miembros son `public`.

Los modificadores de visibilidad, en Kotlin, se pueden aplicar a: clases, objetos, interfaces, constructores, funciones y atributos.

Como ya se sabrá, estos modificadores de visibilidad indican el alcance de los elementos sobre los que se aplica en el programa, y son: `public`, `protected`, `private` e `internal`.

Por defecto, si no se indica nada, todo es `public`.

En el ámbito de las clases y la programacion orientada a objetos, estos modificadores tienen el siguiente impacto:

- `public`: visible en todos lados
- `protected`: visible en subclases
- `private`: visible en la clase
- `internal`: visible en el módulo o paquete

Aquí tienes la clase `Car` con modificadores de acceso:

```
class Car(
    private var brand: String,
    private var model: String,
    private var cv: Int,
    private var age: Int
) {

    private val itvFrequency: Int

    init {
        when (age) {
            in 1..4 -> itvFrequency = 4
            in 5..9 -> itvFrequency = 2
            else -> itvFrequency = 1
        }
    }

    constructor(brand: String, model: String, cv: Int) : this(brand, model, cv, 0) {
        println("Se ha llamado al constructor principal")
    }

    public fun printDetails() {
        println("Brand: $brand, Model: $model, CV: $cv, Age: $age years, ITV Frequency: $itvFrequency years")
    }

    private fun increasePower(increase: Int) {
        cv += increase
        println("The power has been increased by $increase CV. New power: $cv CV")
    }

    internal fun calculateDepreciation(rate: Double = 0.15, currentValue: Double = 10000.0): Double {
        return currentValue * Math.pow(1 - rate, age.toDouble())
    }
}
```

Varios detalles en los que fijarse en este último ejemplo:

- Todos los atributos son privados y, por tanto, no se puede acceder a ellos desde fuera de la clase
- `printDetails` es un método público sin argumentos y se puede acceder a él desde cualquierparte del programa
- `increasePower` es un método privado y, por tanto, solo se puede usar dentro de la clase. Además, tiene un argumento de tipo `Int`
- `calculateDeprecation` es un método con alcance dentro del paquete en que está la clase y tiene dos argumentos con valores por defecto

El siguiente código muestra cómo se puede usar la clase `Car` anterior. Se da por sentado que dicho código está en el mismo fichero, y por tanto en el mismo paquete, que dicha clase `Car`. Hay código comentado porque da error:

```
fun main() {
    var c1 = Car("Hyundai", "i10", 64)
    //println(c1.brand) <- error porque brand es privado

    // Se puede llamar a printDetails porque es public
    c1.printDetails()

    //c1.increasePower(10) <- error porque increasePower es privado

    // Se puede llamar a calculateDepreciation porque es internal y
    // este código está en el mismo fichero y paquete
    val depreciation = c1.calculateDepreciation()
    println("Depreciation = $depreciation")
}
```

5.6. Getter y setter

Los métodos *getter* y *setter* en Kotlin se indican en la declaración del atributo. Se usan en atributos para los que es necesario realizar algún tipo de cálculo, tanto para almacenar un valor como para devolverlo. Si son atributos sin dicha necesidad, en Kotlin se declaran públicos y ya está.

En el siguiente ejemplo se muestra una clase con métodos *getter*:

```
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() {
            return height == width
```

```
        }

    val area get() = this.height * this.width
}
```

- El atributo `isSquare` tiene un método `get` y, por tanto, cuando se acceda a dicho atributo para lectura se ejecuta el `get` - El atributo `area` también tiene un método `get` y, por tanto, cuando se acceda a dicho atributo para lectura se ejecuta dicho `get`

Hay que hacer notar que `isSquare` y `area` son inmutables y, por esa razón, no se pueden asignar valores a dichos atributos.

Ejemplo de uso de la clase `Rectangle`:

```
var r1 = Rectangle(10, 20)

if (r1.isSquare) { // Se ejecuta el código del get del atributo isSquare
    println("Es un cuadrado")
} else {
    println("Es un rectángulo")
}

println("Su área es de ${r1.area} metros cuadrados")

// Si se desomenta el código de abajo no compilaría porque porque isSquare,
// aunque es público, es inmutable.
//r1.isSquare = false

// Por la misma razón no se puede asginar valor a area.
//r1.area = 300
```

Para ver un ejemplo de método `set` se van a hacer varios cambios en la clase `Rectangle`:

- Se añaden métodos `set` a los atributos `width` y `height` para evitar valores menors que 1
- Se elimina el constructor primario
- Se añaden valores por defecto a los atributos `width` y `height`
- En los métodos `get` y `set` se usa la palabra reservada `field` que hace alusión al atributo en cuestión

```
class Rectangle() {
    val isSquare: Boolean
        get() {
            return height == width
        }

    val area get() = this.height * this.width

    var height: Int = 0
        get() = field
        set(value) {
            if (value < 1) {
                field = 1
            } else {
                field = value
            }
        }

    var width: Int = 0
        get() = field
        set(value) {
            if (value < 1) {
                field = 1
            } else {
                field = value
            }
        }
}
```

5.7. Herencia

La herencia en la programación orientada a objetos evita duplicidad en el código al permitir la reutilización de clases genéricas heredando de ellas.

Kotlin permite la herencia, pero hay que tener en cuenta varios detalles y particularidades del lenguaje:

- Todas las clases en Kotlin heredan de una clase común llamada `Any`
- Las clases en Kotlin son todas `final`

Quizás sea conveniente recordar que una clase `final` no puede tener clases derivadas. Dicho de otro modo, no se puede heredar de clases `final`

- Para permitir clases derivadas de una clase dada hay que usar el modificador open
- Lo mismo sucede para los miembros de una clase, que son todos `final` y, por tanto, si se desea que el miembro se pueda heredar hay que usar, aquí también, el modificador open
- En las clases derivadas hay que especificar con el modificar `override` los miembros que se sobreescribirán

Aquí se muestra un programa en Kotlin donde se usa la herencia:

```
open class Shape {
    open fun draw() {
        println("Shape")
    }

    fun fill() {
        println("Filled")
    }
}

class Circle : Shape() {
    override fun draw() {
        println("Circle")
    }
}

open class Rectangle : Shape() {
    final override fun draw() {
        println("Rectangle")
    }
}
```

¿Qué se puede observar en esta jerarquía de clases?

- Shape es una clase “abierta” y, por tanto, se puede derivar de ella. Solo el método `draw` es sobreescribible por sus clases derivadas.
- `Circle` hereda de Shape y sobreescribe el método `draw`. No se puede heredar de esta clase `Circle` porque es `final` (cuando no se indica nada es `final`).
- `Rectangle` es una clase derivada de Shape y, además, es una clase “abierta” de la que se puede derivar otras clases. Sobreescribe el método `draw` de su padre y permite que, dicho método, sea sobreescribible por sus posibles clases derivadas.

El mismo ejemplo con atributos y constructor principal:

```
open class Shape(open val vertexCount: Int) {
    open var color: String = "Black"

    open fun draw() {
        println("Shape")
    }

    fun fill() {
        println("Filled")
    }
}

class Circle(val radius: Int) : Shape(0) {
    override var color: String = "Red"

    override fun draw() {
        println("Circle")
    }
}

open class Rectangle(val width: Int, val height: Int) : Shape(4) {
    open override var color: String = "Green"

    final override fun draw() {
        println("Rectangle")
    }
}
```

5.8. Interfaces

La idea de las interfaces es poder usar tipos genéricos, obligando a las clases que las implementan a un comportamiento determinado.

No está de más recordar que las interfaces no se pueden instanciar.

En Kotlin, las interfaces tienen características diferentes a Java pero, como en Java, y como se acaba de recordar, no se pueden crear instancias de interfaces. Se indican las características diferentes a Java:

- Los métodos de las interfaces pueden tener implementación: así hay métodos abstractos que deben ser implementados por las clases que usen la interfaz, y hay métodos con implementación por defecto.
- Las interfaces pueden tener atributos que, como en el caso anterior, pueden tener o no implementación por defecto.

Por ejemplo, en la interfaz Shape se tienen atributos abstractos, atributos con implementación, métodos abstractos y métodos con implementación (el código está comentado para facilitar la comprensión):

```
interface Shape {
    // Atributo abstracto: sin implementación.
    val name: String

    // Atributo con implementación
    val sides: Int
        get() = 0

    // Método abstracto (sin implementación)
    fun area(): Double

    // Método con implementación
    fun describe() {
        println("El nombre del área es $name y tiene $sides lados.")
    }
}
```

La siguiente clase `Rectangle` implementa la interfaz Shape y, por tanto, tiene que implementar los métodos y atributos abstractos:

```
class Rectangle(val width: Double, val height: Double) : Shape {
    // Implementación de la propiedad abstracta
    override val name: String = "Rectangle"

    // Sobrescribiendo la propiedad con implementación
    override val sides: Int
        get() = 4

    // Implementación del método abstracto
    override fun area(): Double {
        return width * height
    }
}
```

> Un detalle a tener en cuenta es que al implementar la interfaz se escribe `: Shape` y no `: Shape()` como en la herencia. ¿Por qué? Porque las interfaces no se pueden crear, se implementan, y las clases sí se crean, por eso tienen el operador *llamada* o *()*.

5.9. Clases abstractas

Una de las grandes diferencias entre las interfaces y las clases abstractas es que estas últimas pueden definir métodos (tienen métodos con implementación) y atributos. Pero en Kotlin, como ya se ha visto más arriba, las interfaces pueden tener métodos con implementación y atributos. Así que, en ese sentido no hay diferencia.

No obstante siguen habiendo diferencias que se tienen que conocer para saber qué usar en cada caso:

Interfaces	Clases abstractas
Una clase puede implementar varias interfaces	Una clase solo puede heredar otra clase, sea o no abstracta /td>
No pueden tener constructores	Pueden definir constructores
No tienen estado: atributos con valores	Sí pueden tener estado: atributos con valores
Los miembros solo pueden ser `public` (por defecto) e `internal`	Los miembros pueden ser `public` (por defecto), `protected`, `private` e `internal`
Se usan para definir contratos, permitir "herencia" múltiple o desacoplar dependencias	Se usan para compartir código, cuando se necesita constructores y evitar duplicidad en el código

Dicho lo cual, aquí tienes un programa en el que se usan clases abstractas en Kotlin (el código está comentado para que se entienda mejor):

```
// Las clase abstracta tiene constructor primario
abstract class Vehicle(val name: String) {
```

```
    // Método abstracto que las subclases deben implementar
    abstract fun startEngine()

    // Método concreto que puede ser utilizado por las subclases
    fun stopEngine() {
        println("$name engine stopped.")
    }

    // Método concreto con implementación
    fun stop() {
        println("$name is stopping.")
    }

    // Método abstracto que las subclases deben implementar
    abstract fun drive()
}
```

Y, ahora, un par de clases que extienden de la clase abstracta anterior:

```
// Se usa el constructor de la clase abstracta para crear la clase específica Car
class Car(name: String) : Vehicle(name) {

    // Este método es abstracto, así que hay que implementarlo en la clase derivada
    override fun startEngine() {
        println("$name engine started.")
    }

    // Este método es abstracto, así que hay que implementarlo en la clase derivada
    override fun drive() {
        println("$name is driving.")
    }
}

// Se usa el constructor de la clase abstracta para crear la clase específica Bike
class Bike(name: String) : Vehicle(name) {

    // Este método es abstracto, así que hay que implementarlo en la clase derivada
    override fun startEngine() {
        println("$name engine started.")
    }

    // Este método es abstracto, así que hay que implementarlo en la clase derivada
    override fun drive() {
        println("$name is driving.")
    }
}
```

5.10. Data classes (o clases para datos)

Esta es una novedad con respecto a Java pero es un concepto o idea muy utilizada en otros lenguajes como Python.

Los *data classes* son clases que se usan básicamente para mantener datos relacionados. Kotlin ofrece una sintaxis muy compacta para crear este tipo de clases en las que, el propio compilador, añade automáticamente la implementación de estos métodos:

- Los métodos `.equals` y `.hashCode` para poder comparar objetos del *data class*
- El método `.toString()` para poder representar el *data class* en forma de String
- El método `.copy()` para copiar objetos con la posibilidad de cambiar algún atributo en el proceso de copia
- Los métodos `.componenetN()` para desestructurar atributos (lee la documentación oficial)

Se usa el modificador `data` para crear este tipo de clases:

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

En una única línea tenemos un *data class* para mantener los atributos `name` y `age` agrupados. La clase equivalente, si no existieran los *data class*, sería esta:

```
class User(val name: String, val age: Int) {

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is User) return false

        if (name != other.name) return false
        if (age != other.age) return false
```

```
        return true
    }

    override fun hashCode(): Int {
        var result = name.hashCode()
        result = 31 * result + age
        return result
    }

    override fun toString(): String {
        return "User(name='$name', age=$age)"
    }

    fun copy(name: String = this.name, age: Int = this.age): User {
        return User(name, age)
    }

    operator fun component1(): String = name
    operator fun component2(): Int = age
}
```

La ventaja de los *data class* es evidente.

5.11. Enumeraciones

En Kotlin también se tienen enumeraciones y funcionan de manera muy parecida a Java.

en la documentación oficial de Kotlin se tiene toda la información necesaria.

A modo de ilustración, se muestra un programa en el que se usan enumeraciones, pero también *data classes*, y debería comprensible si se han seguido todos los apartados anteriores:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}

data class Point(var x: Int, var y: Int)

fun move(point: Point, direction: Direction): Point {
    when (direction) {
        Direction.NORTH -> point.y += 1
        Direction.SOUTH -> point.y -= 1
        Direction.WEST -> point.x -= 1
        Direction.EAST -> point.x += 1
    }
    return point
}

fun main() {
    val point = Point(0, 0)

    println("Punto inicial: $point")

    move(point, Direction.NORTH)
    println("Después de moverse al norte: $point")

    move(point, Direction.EAST)
    println("Después de moverse al este: $point")

    move(point, Direction.SOUTH)
    println("Después de moverse al sur: $point")

    move(point, Direction.WEST)
    println("Después de moverse al oeste: $point")
}
```

6. Funciones lambda

En el mundo de la programación existen unas construcciones denominadas **funciones lambda** o, también, denominadas **funciones anónimas**.

Las funciones lambda son funciones sin nombre, y se suelen utilizar como argumentos de otras funciones a las que se conoce como

funciones de orden superior (en inglés **higher-order functions**).

> Las funciones lambda se usan para pasar bloques de código a otras funciones o cuando se usan una o muy pocas veces.

Este es un concepto que viene de la programación funcional que, en los últimos años, se ha popularizado e introducido en todo tipo de lenguajes como Java o C++.

6.1. Funciones lambda en Kotlin

Kotlin incorpora funciones lambda y se suelen usar para pasarlas como argumentos de funciones de orden superior.

Las funciones lambda son, esencialmente, funciones anónimas que pueden ser tratadas como valores: se pueden, por ejemplo, pasar como argumentos de otras funciones, devolverlas como resultados de otras funciones, o hacer cualquier cosa que se podría hacer con una variable u objeto.

Así pues, en un lenguaje como Kotlin, necesitamos una sintaxis para indicar el tipo de un lambda, su *signatura* con **el tipo de los argumentos** y el **tipo de retorno**:

`(<Tipo Parámetro 1>, ...., <Tipo Parámetro N>) -> <Tipo de Retorno>`.

A continuación, se muestran declaraciones de variables del tipo *lambda expression*:

```
// Variable a la que se le puede asignar una función
// que no recibe argumentos ni devuelve valor.
var fun1: () -> Unit = {}

// Variable a la que se le puede asignar una función
// que recibe un número entero y no devuelve nada.
var fun2: (Int) -> Unit = {}

// Variable a la que se le puede asignar una función
// que recibe dos argumentos (el primero de tipo String
// y el segundo de tipo Int) y devuelve un valor de tipo
// String.
var fun1: (String, Int) -> String = {}
```

6.2. Ejemplos comentados

En el siguiente ejemplo se puede ver el uso de una función lambda que calcula el producto de dos números.

```
// Función anónima que recibe dos números y devuelve el producto de dichos números.
var multiply: (Int, Int) -> Int = { n1: Int, n2: Int -> n1 * n2 }

// Imprime por pantalla 6
println(multiply(2, 3))

// Imprime por pantalla 16
println(multiply(4, 4))
```

Dado que en Kotlin existe la inferencia de tipos, aquí se muestra el ejemplo de arriba pero sin indicar el tipo de la lambda:

```
// Queda claro, y se puede inferir, que el tipo de la variable
// multiply es una lambda que recibe dos números enteros y devuelve
// su multiplicación (otro número entero).
var multiply = { n1: Int, n2: Int -> n1 * n2 }

println(multiply(2, 3))
println(multiply(4, 4))
```

6.3. Funciones de orden superior

Dado que es una de los principales usos de las lambda, aquí se muestra el ejemplo de un programa que recibe un número y una funcion lambda para operar con dicho número.

```
fun operation(num: Int, op: (Int) -> Int): Int {
    val result: Int = op(num)
    return result
}

fun main() {
    val r1 = operation(2, { n -> n * 2 })
    println(r1)

    val r2 = operation(5, { n -> n + 5 })
    println(r2)
```

```
}
```

Descripción del funcionamiento del programa:

- Se ha definido una función llamada `operation` que recibe un número entero llamado `num` y una función labmda llamada op. Esta función lambda, a su vez, recibe un número entero y devuelve otro número entero.
- En la funcion `main` se está llamando a la función de orden superior `operation` dos veces:
  - En la primera se pasa el número 2 y una función lambda que recibe un número y devuelve el producto de este número por 2.
  - en la segunda se pasa el número 5 y una función lambda que recibe un número y devuelve la suma de dicho número y 5.

6.4. Nombre implícito para funciones lambda con un solo argumento: it

Dado que muchas funciones lambda reciben un único argumento, para hacer más compacto el código, en Kotlin puedes obviar la declaración de dicho argumento y se llamará `it`.

Aquí se muestra el ejemplo anterior donde se omite la declaración del argumento n de la función lambda y se usa el argumento por defecto `it` en su lugar:

```
fun operation(num: Int, op: (Int) -> Int): Int {
    val result: Int = op(num)
    return result
}

fun main() {
    val r1 = operation(2, { it * 2 })  // Se usa argumento por defecto: it
    println(r1)

    val r2 = operation(5, { it + 5 })  // Se usa argumento por defecto: it
    println(r2)
}
```

7. Colecciones

Las colecciones son grupos de variables relacionados. Es un término que a los programadores que vengan de Java resultará familiar.

Las colecciones relevantes en Kotlin son:

- `List`: es una colección ordenada con acceso a sus elementos por medio de índices (números enteros que reflejan la posición del elementos, comenzando por 0). El orden de los elementos, en las `List`, es importante. Además, los elementos se pueden repetir.

  Cuando se dice que los elementos de una lista están ordenados, se refiere a que se mantiene el orden en que fueron introducidos a la lista.
- `Set`: es una colección de elementos únicos (un elemento no puede aparecer en la colección más de una vez). Esta colección es una abstracción del concepto de conjunto en matemáticas: grupo de elementos sin repetición. En los `Set` el orden de los elementos no es importante.
- `Map`: a veces también conocido como diccionario, es un conjunto de pares claves-valor (key-value). Las claves son únicas y los valores se pueden repetir.

7.1. Colecciones mutables e inmutables

La librería estándar de Kotlin ofrece interfaces para la creación de colecciones que pueden ser: inmutables o mutables.

Las **colecciones mutables** pueden ser modificadas: se pueden añadir, modificar y eliminar elementos.

Las **colecciones inmutables** quedan como fueron inicializadas y no se pueden modificar.

Esto es independiente que uses el modificador `val` o `var` para crear las colecciones. Con `val` consigues que la referncia de la colección no cambie, mientras que con `var` permites que la referencia de la colección pueda cambiar.

Un cosa es **el objeto en sí**, la colección, y otra **los elementos de dicha colección**. Así pues, cuando se habla de que una coleccion es mutable es porque sus elementos puedan cambiar, y cuando se habla de que una colección es inmutable es porque sus elementos no pueden ser modificados.

7.1.1. Aclaraciones en código

Con la lectura del siguiente código (acompañado por comentarios) debería quedar claro:

- Cómo se crean colecciones
- Qué diferencia hay entre colecciones mutables e inmutables
- En qué afecta el uso de `val` o `var` sobre las colecciones

```
// Crea una lista de números mutables
// Además, la lista numbers no puede ser modificada
val numbers = mutableListOf(1, 2, 3, 4, 5)

// Pero sí sus elementos porque numbers es mutable
// Aquí se cambia el valor de la posición 1 por el valor 10
numbers[0] = 10

// Muestra: [10, 2, 3, 4, 5]
// Hemos cambiado el primer valor de 1 a 10
println(numbers)

// Esta daría error: una variable val no puede ser reasignada
//numbers = mutableListOf(10, 9, 8, 7)

// Crea una lista de strings inmutable
var names = listOf("Alice", "Bob")

// Esto no se puede hacer porque la lista es inmutable
// Si se descomenta dará error
//names[0] = "Mary"

// Sí se puede reasignar un nuevo valor a la variable names
// porque se usó var
names = listOf("Alice", "Bob", "Mary", "Jon")
```

7.2. Funciones y manejo de colecciones

Las tres colecciones básicas de Kotlin pueden ser creadas y manejadas por medio de una gran cantidad de funciones e interfaces. Aquí se resumen algunas importantes con el objetivo de que se pueda comprender el funcionamiento de esta parte en Kotlin.

Se pueden **crear colecciones inmutables** con las funciones: `listOf`, `setOf` y `mapOf`. Hay ejemplos en la web oficial de Kotlin:

- Ejemplos para listas
- Ejemplos para conjuntos
- Ejemplos para mapas

Para **crear colecciones mutables** se encuentran las funciones análogas: `mutablesListOf`, `mutableSetOf` y `mutableMapOf`.

Cuando se crean colecciones vacías (inicialización vacía) se debe usar el **operador de tipo genérico** porque el compilador no puede deducir el tipo. Por ejemplo:

```
// Creación de una lista mutable de enteros vacía
var numbers = mutableListOf<Int>()
```

Por último, en la documentación oficial tienes documentada las funciones que se pueden usar sobre las colecciones, es decir la API de cada tipo de colección:

- API de las listas
    - Algunos ejemplos de funciones sobre listas
- API de los conjuntos
    - Algunos ejemplos de funciones sobre conjuntos
- API de los mapas
    - Algunos ejemplos de funciones sobre mapas

7.3. Construir colecciones

Es muy importante dominar la construcción de colecciones para terminar de entenderlas. En la documentación oficial de Kotlin tienes una página con ejemplos sobre cómo construir colecciones cuya lectura es más que obligatoria.

7.4. Recorrer colecciones

En Kotlin se pueden recorrer colecciones por medio del mecanismo conocido como **iterador**. Un **iterador** (*Iterator*) es un objeto que proporciona acceso a los elementos de la colección ocultando los detalless de la misma.

Para obtener el iterador de un colección solo hay que llamar a la función `iterator()`. Esta función devuelve la posición al primer elementos de la colección.

La función `next` de un iterador devuelve el elemento de la posición a la que apunta y se mueve al siguiente elemento si lo hubiera.

La función `hasNext` devuelve `true` si el iterador apunta a un elemento válido y `false` si se acabó y el iterador ha salido de la colección.

7.4.1. While

En este ejemplo se usa este mecanismo para mostrar los elementos de una lista en un bucle `while`:

```
// Crea una lista inmutable de strings con nombres de personas,
val people = listOf("Alice", "Bob", "Mary", "Jon")

// Obtiene el iterator de la lista,
// Al obtener el iterator de una colección se tiene la referencia a la
// primera posición de la colección,
val it = people.iterator()

// Cada vez que se llama a next se devuelve el elemento sobre el que apunta
// el iterator y se mueve una posición adelante.
// Para comprobar si hay más elementos se llama a la función hasNext,
while (it.hasNext()) {
    println(it.next())
}
```

7.4.2. For

El uso del bucle `for` es más adecuado para recorrer colecciones porque el manejo del objeto iterador lo hace internamente. El mismo ejemplo pero con bucle `for`:

```
val people = listOf("Alice", "Bob", "Mary", "Jon")
for (name in people) {
    println(name)
}
```

El bucle `for` va colocando en `name` el siguiente elemento de la lista y cuando no quedan más sale directamente.

7.4.3. For Each

Una alternativa al bucle `for` es la de usar *programación funcional* y, en este caso concreto, la función `forEach` de las colecciones. Esta función espera una función lambda como argumento a la que le pasa el siguiente elemento de la colección:

```
val people = listOf("Alice", "Bob", "Mary", "Jon")
people.forEach { name ->
    println(name)
}
```

7.5. Operaciones map, filter y reduce

Se puede decir que estas son las tres operaciones básicas sobre colecciones que se usan en la programación funcional, y Kotlin las incorpora. Se trata de funciones de orden superior, pues todas ellas reciben como argumento una función anónima o lambda:

- La operación **map** se usa para transformar los elementos de una colección. El resultado es una colección con el mismo número de elementos pero con alguna transformación.
- La operación **filter** se usa para filtrar y quedarse con un subconjunto de los elementos de una colección. El resultado es otra colección con, solo, los elementos que cumplan una determinada condición.
- La operación **reduce** se usa para obtener un resultado tras aplicar algún tipo de operación sobre los elementos de la colección. El resultado es un único elemento.

  No confundir la **operación map** con la colección **Map**.

7.5.1. La operación map

Ejemplo típico de `map` donde se tiene una colección de números que se transforman de algún modo. En este caso concreto se multiplican todos por 2 y se obtiene la misma lista de números pero multiplicados por 2. La función map recibe una función lambda que recibe un valor y realiza alguna transformación. Internamente, recorre todo la colección y devuelve el resultado de aplicar dicha lambda a cada valor.

> La lista original se mantiene intacta y se devuelve una copia con los valores transformados.

```
// Lista de enteros
val numbers = listOf(1, 2, 3, 4, 5)

// Usando la función map para duplicar cada elemento
val doubledNumbers = numbers.map { it * 2 }

// Imprimiendo la lista transformada: [2, 4, 6, 8, 10]
println(doubledNumbers)
```

7.5.2. La operación filter

Ejemplo típico de `filter` en el que se tiene un colección de números y se obtienen solo los pares. Lo que recibe la función `filter` es una función lambda que realice alguna acción devolviendo un valor booleano. Internamente se aplica dicha lambda sobre todos los valores de la colección y devuelve otra colección con los valores para los que la lambda devuelve `true`.

> La lista original se mantiene intacta y se devuelve una nueva lista con los valores que pasan el filtro.

```
// Lista de enteros
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Usando la función filter para obtener solo los números pares
val evenNumbers = numbers.filter { it % 2 == 0 }

// Imprimiendo la lista filtrada: [2, 4, 6, 8, 10]
println(evenNumbers)
```

7.5.3. La operación reduce

Ejemplo típico de `reduce` en el que se toma una lista de enteros y se devuelve la suma de los mismos. La función de orden superior `reduce` recibe una función lambda que recibe dos argumentos: el valor acumulado y el nuevo valor. Esta función lambda realizará una operación devolviendo el nuevo valor acumulado:

```
// Lista de enteros
val numbers = listOf(1, 2, 3, 4, 5)

// Usando la función reduce para sumar los números
val sum = numbers.reduce { acc, number -> acc + number }

// Imprimiendo la suma de los números: 15
println(sum)
```

8 Prácticas y ejercicios.

8.1 Introducción.

Ejercicio 1

¿Qué dos filosofias implementan las principales plataformas de desarrollo de aplicaciones multiplataforma? ¿Qué pros y contras poseen cada una de ellas?

Ejercicio 2

Realizar una comparativa usando los portales de empleo sobre las ofertas que demandan conocer las siguientes tecnologias/lenguajes. * Kotlin. * PHP. * NodeJs. * Laravel. * Java. * Spring.

Ejercicio 3

¿Que necesidades cubre Kotlin?

Ejercicio 4

A partir de la entrada en la Wikipedia LLVM, indicar:

- ¿Qué se desarrolla con el?
- ¿Qué hace LLVM con el código IF?
- ¿Cómo se consigue compilar para plataformas nativas?

Preguntas

¿Sobre que máquina se ejecuta Java? ¿Qué lenguaje entiende la máquina anterior? ¿Cuál es su equivalente en la plataforma .NET? ¿Qué aporta la compilación JIT? ¿Y la AOT? ¿Cuál es mejor? ¿Qué lenguajes a influenciado a Kotlin? ¿Para que arquitecturas compila el compilador de Kotlin?

Ejercicio 5

Descargar el compilador. Añadir al path la ruta de los ejecutables de Kotlin Usando un editor de texto sencillo, crear un programa que muestre un cuadrado con asteriscos de 3x3 Compilar Ejecutar Generar un fichero jar Ejecutar el Jar

Ejercicio 6

Abrir la consola interactiva Crear una variable:

```
 var a=10
```

Imprimir la variable:

```
println(a)
```

Incremetar la variable en 2 Volver a imprimir Salir de la consola interactiva

Ejercicio 7

Usando Gradle, crear un programa en Kotlin que imprima una X, con asteriscos, de 3x3 Ejecutar el programa. ¿Qué tareas se pueden ejecutar? ¿En que fichero se encuentra la configuración de Gradle?

Ejercicio 8

Iniciar el programa del ejercicio 6 en modo depuración y establecer al menos un punto de ruptura. Ejecutar el programa saltando entre puntos de ruptura.

8.2. Variables y estructuras de control.

Cuestiones.

- ¿Qué tipos de variables se pueden encontrar en Kotlin? ¿Cuál es la diferencia entre ellas?
- Indicar al menos un ejemplo de representación de datos que use cada una de ellas, por ejemplo, PI
- ¿En Kotlin existen tipos de datos primitivos de la misma forma que en Java?
- ¿Qué es la inferencia de tipos? ¿Quién decide de que tipo es una variable a la que no se le ha dado explicitamente?
- Características de los multiline strings.
- Diferencia entre el operador == y el operador ===
- En Kotlin, por defecto, ¿puede inicializarse una variable a NULO?
- ¿Qué operador se utiliza para poder definir una variable a NULO?
- Indicar en las siguientes declaraciones de variables, si es posible que tengan el valor NULO, o

```
var nombre: String? = "Juan"
var nombre2: String = "Juan"
var nombre3:String=nombre?:"Paco"
val nombre4:String?="Pedro"
```

- Comparar la estructura de control switch de Java con la sentencia de control when de Kotlin
- Indicar las diferencias entre un bucle for en Java y un bucle for en Kotlin
- Escribir la sentencia for para realizar una acción 10 veces.
- ¿Cómo se ejecutaria un bucle que va desde 1000 hasta 500 en saltos de 3?

Es imprescindible documentar el código.

Ejercicio 1

Se necesita realizar un estudio de las ocurrencias de cada carácter que aparece en un texto, implementar un programa que muestre el número de ocurrencias de cada letra en minúsculas (ver como transformar en minúsculas) en el siguiente fragmento de código (Eliminar los acentos y ñ, SOLO SE TIENE EN CUENTA LOS CARACTERES DE LA a-z). Para crear un array en Kotlin, ver la documentación oficial

Artículo 14. Los españoles son iguales ante la ley, sin que pueda prevalecer discriminación alguna por razón de nacimiento, raza, sexo, religión, opinión o cualquier otra condición o circunstancia personal o social .

Ejercicio 2

Crear un programa que a partir de una palabra definida como constante indique si es palíndromo o no. Una palabra es palíndromo si se lee igual de derecha a izquierda que de izquierda a derecha, por ejemplo rodador (Cuidado con el ejercicio, esta solucionado en Internet, pero si se pregunta en un examen escrito o alguna variación del mismo...)

Ejercicio 3

Implementar un algoritmo que indique si un cuadrado es mágico. Un cuadrado mágico es aquel en el que las filas y columnas suman lo mismo, realizar pruebas con diferentes cuadrados y dimensiones: Cuadrado mágico

Ejercicio 4

La empresa “TecCalzado” ha detectado que algunos de sus correos electrónicos han sido leídos por la competencia y han decidido utilizar un sistema de codificación para evitar la lectura de los mensajes, han decido implementar un método por sustitución, a pesar de que tiene problemas de seguridad.

El método de codificación es Polibios. Data del año 150 a. C. siendo uno de los algoritmos de sustitución más antiguo del cual se tiene conocimiento y recibe el nombre de Polibios, nombre que se le dio en reconocimiento al historiador griego del mismo nombre y de quien se considera fue su creador.

Proceso de cifrado:

Para llevar a cabo el proceso de cifrado se consideran la primera columna y el primer renglón de la tabla anterior como el par criptográfico correspondiente a cada letra dentro de la matriz de 5 X 5 mostrada en la tabla, de manera que justo en ese orden,

renglón-columna, son las dos letras que sustituyen a cada una de las letras que pueden conformar el mensaje en claro.

matriz

Así, por ejemplo para la letra M el criptograma correspondiente es CB, en tanto que para la U es el par DE, de manera que de acuerdo con este algoritmo se puede observar que se sustituye el alfabeto {A, B , C, D, E, F, ...X, Y, Z} por el alfabeto de cifrado {AA, AB, AC, AD, AE, BA,..., EC, ED, EE}, entre lo que destaca de manera importante que el criptograma correspondiente a un mensaje en claro cifrado con este algoritmo siempre contendrá el doble de caracteres que el texto plano, característica que no es precisamente lo más deseado ya que los criptogramas pueden alcanzar dimensiones muy grandes, complicados de manipular, y con la necesidad de un espacio de almacenaje duplicado al tamaño original.

Para ejemplificar el proceso de sustitución con base en la tabla anterior:

Mcla P O L Y B I O S E S G R I E G O Sustitución CE CD CA ED AB BD CD DC AE DC BB DB BD AE BB CD

Cripto = CECDCAEDABBDCDDCAEDCBBDBBDAEBBCD

Proceso de descifrado:

Para llevar a cabo el proceso inverso, esto es, el proceso de descifrado, se parte el criptograma a descifrar, leyendo éste de izquierda a derecha y tomando en cada ocasión un par de caracteres para llevarlos a la tabla de descifrado de manera que el primero lo ubicamos con su similar en la primera columna de la tabla y el segundo con su símil del primer renglón, entonces extendemos una línea imaginaria sobre la columna y el renglón identificados y en la celda donde éstas se intersecan se encuentra el carácter correspondiente al mensaje en claro de ese par de elementos del criptograma.

Por ejemplo, si se desea descifrar el criptograma AEDCDDDEADBDCD, de izquierda a derecha se toman los elementos que lo conforman de dos en dos, y se llevan a ala tabla de cifrado. Después de desarrollar el descifrado para el criptograma se obtiene: Cripto: AE DC DD DE AD BD CD

Mcla: ESTUDIO

Crear un programa que solicite una frase por terminal, indique si se quiere codificar o descodificar, y si se realiza con número o letras, LA CODIFICACIÓN Y DECODIFICACIÓN DEPENDE DE LA DEFINICIÓN DE LA TABLA, PERO HA DE AL DESCODIFICAR SE HA DE OBTENER EL MISMO ORIGINAL:

```
>Introducir frase: Sábete Sancho, que no es un hombre más que otro si no hace más que otro
>Codificar(1)/Descodificar(0): 1
>Números(1)/Letras(0):1
>Frase codificada: 4312154415 431133132334 414515 3334 1543 4533 233432124215 3243 414515 34444234 4324 3334 23111315 32
```

Para leer el terminal consultar la siguiente web

8.3. Funciones.

Ejercicio 1

Crear una función llamada mayor que indique si un número entero es mayor que otro.

Ejercicio 2

Realizar 2 llamadas a la función anterior con los valores 4,5 y 6,2.

Ejercicio 3

Crear una función llamada mayor que devuelve el mayor de 3 caracteres en el mismo fichero que la función del ejercicio 1.

Ejercicio 4

Usar la función del ejercicio 3 con los números 5,6,1 y 4,88,64.

Ejercicio 5.

Crear una función que descomponga un número en factores primos, es necesario además crear otra función que indique si un número es primo o no, por ejemplo al descomponer 45, se empieza por el 2 que es primo (llamara a esprimo(2)) pero 45 no es divisible por 2, pasando al 3, que es primo (se comprueba) y 45 es divisible por 3, quedando 15 y mostrando el 3, a continuación se pasa a 4 y se comprueba si es primo (no lo es), con lo que se pasa al 5....

Ejercicio 6

Probar la función anterior con los números 15, 56 y 678.

Ejercicio 7

Crear una función que devuelve el área de un círculo con valores reales y probarlo.

Ejercicio 8

Implementar una función que imprima un vector de enteros, recordar que no se tiene el atributo length en C.

Ejercicio 9

Desarrollar una función que ordene un vector, llamar a la función e imprimir a continuación el vector ordenado.

Ejercicio 10

Crear una función recursiva que calcule eleve un número a otro y probar. Definir en primer lugar caso base y caso recursivo en papel.

Ejercicio 11

Definir una función que indique si un número es primo o no, de forma recursiva. Definir en primer lugar caso base y caso recursivo en papel.

Ejercicio 12

Definir una función que muestre el mensaje “Hola $nombre”, donde el nombre se ha de pasar como parámetro, en caso de no pasarlo se mostrará la palabra “Hola Caracola”. :stuck_out_tongue_winking_eye:

Ejercicio 13

Se necesita una función que muestre una tabla de multiplicar, tiene 3 parámetros: número inferior (por defecto 0), número superior (defecto 10) y multiplicador (por defecto 0). Crear la función y probarla sin pasar valores, pasando solo el superior y el multiplicador, pasar solo el multiplicador y el inferior y pasándolos todos.

Ejercicio 14.

Se tiene una función principal que recibe un conjunto de parámetros variable, el programa únicamente muestra el número de parámetros y una lista de los mismos. Definirla y probar pasando desde la línea de comandos un número variable de parámetros.

Ejercicio 15.

Se tiene la clase Pen:

```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, BLACK
}

data class Pen(
    val duration: Int,
    val thicknessgrosor: Int,
    val color: Color
)
```

Y se ha implementado un algoritmo de ordenación, pero se desea que unas veces se ordene por duración y otras por grosor. Usando las variables como funciones y funciones como parámetros, crear dos funciones, la primera devuelve la diferencia de grosor y la segunda la diferencia de duración. Se facilita el algoritmo de ordenación y un array de Pen :exclamation: :

```
fun main() {
    var array = arrayOf(
        Pen(2, 4, Color.RED),
        Pen(6, 1, Color.BLUE),
        Pen(4, 5, Color.BLACK)
    )
    // usando bubbleSort ordenar por los dos campos usando variables a funciones

}

fun bubbleSort(arr: Array<Pen>, f: (p1: Pen, p2: Pen) -> Int) {
    val n = arr.size
    for (i in 0 until n - 1) {
        for (j in 0 until n - i - 1) {
            if (f(arr[j], arr[j + 1]) > 0) {
                // Swap the elements
                val temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
            }
        }
    }
}
```

La salida ordenando por los dos atributos con los datos facilitados es:

```
Ordenado por duración
Pen(duration=2, thicknessgrosor=4, color=RED)
```

```
Pen(duration=4, thicknessgrosor=5, color=BLACK)
Pen(duration=6, thicknessgrosor=1, color=BLUE)
.........................
Ordenado por grosor
Pen(duration=6, thicknessgrosor=1, color=BLUE)
Pen(duration=2, thicknessgrosor=4, color=RED)
Pen(duration=4, thicknessgrosor=5, color=BLACK)
```

Ejercicio 16.

Piden mejorar el programa anterior, de forma que sea capaz de ordenar cualquier clase y pudiendo cambiar que se entiende por mayor o menor para cada clase (igual que el ejercicio anterior, pero con cualquier tipo, como Brush,Draft ...) :pencil2: . Para ello se usan las funciones genericas La función de ordenación recibe un array de tipo T y una variable de tipo función, esta función recibe 2 elementos de tipo T (genericios) y devuelve un entero. La cabecera de la función de ordenación:

```
fun <T>bubbleSort(arr: Array<T>, f: (p1: T, p2: T) -> Int)
```

Se puede probar añadiendo la clase:

```
enum class Shape{
    CIRCLE, RECTANGLE, ROUNDED
}
data class Brush(
    val color: Color,
    val hardness:Int,
    val shape:Shape,
    val separation:Int
)
```

8.4. POO.

Ejercicio 1.

Escribir una **data class** en Kotlin llamada `Coordinate` que se pueda usar para almacenar las coordenadas **x** e **y** de una espacio bidimensional. Las coordenads `x` e `y` serán de tipo `Float` y se inicializarán con el valor `0f`.

Ejercicio 2.

Escribir una **data class** en Kotlin llamada `Size` que permita almacenar el ancho y el alto de un elemento cualquiera. Llamar al atributo que almacena el ancho `w` y al atributo que almacena el alto `h`.

Ejercicio 3.

Crear una clase abstracta de Kotlin llamada `GameElement` con los siguiente atributos:

- Atributo protegido llamado `c`, de tipo `Coordinate` (del ejercicio 1) y mutable.
- Atributo protegido llamado `s` del tipo `Size` (del ejercicio 2) y mutable.
- Atributo protegido llamado `image` de tipo `Array<Array<Char>>` e inmutable.

Esta clase abstracta tendrá dos funciones abstractas:

- `paint` que recibe un objeto de tipo `Screen` (de la librería `com.googlecode.lanterna.screen.Screen`) llamado `s` y no devuelve nada.
- `update` que no recibe ningún argumento ni devuelve ningún valor.

Por último, tendrá tres métodos con implementación:

- `getPosition` que no recibe nada y duelve `c`.
- `getSize` que no recibe nada y devuelve `s`
- `evalCollision` que recibe un argumento llamado `element` del tipo `GameElement` y devuelve `true` si hay colisión entre este elemento y el elemento pasado como parámetro.

Con respecto al último método, `evalCollision`: dos `GameElement` colisionan si en el espacio bidimensional coinciden en algún punto.

8.5. Colecciones y lambdas.

Ejercicio 1.

Dada una colección de nombres (de tipo `String`) escribir un programa en Kotlin que muestre por pantalla una colección con las dimensiones de los nombres. Por ejemplo, dada la lista `["Alice", "Bob", "Mary", "Jon"]` el programa mostrará por pantalla la lista `[5, 3, 4, 3]`. Usa la función de orden superior `map`.

Ejercicio 2.

Escribir un programa en Kotlin que construya una lista de frases y las muestre del revés. Usa la función de orden superior `map` para ello.

Ejercicio 3.

Escribir un programa en Kotlin que, dada una lista de palabras, utilice `filter` para crear una nueva lista que contenga solo las palabras que tienen más de 5 caracteres.

Ejercicio 4.

Escribir un programa en Kotlin que, dada una lista de cadenas, utilice `filter` para obtener una nueva lista que contenga solo las cadenas que contienen la letra “a”.

Ejercicio 5.

Dada la siguiente **data class** de Kotlin:

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

Escribir un programa que, dado una lista de `Person`, utilice `filter` para obtener una nueva lista de `Peron` que sean mayores de edad (que tengan 18 años o más).

Ejercicio 6.

Dada la *data class* `Person` anterior, escribir un programa que, dado una lista de `Person`, utilice reduce para obtener la suma de las edades de las personas de dicha lista.