PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TEMA 3:

Introducción al lenguaje de programación Kotlin

ÍNDICE

PMDM

Introducción al lenguaje de programación Kotlin

1. ¿Qué es Kotlin?

1.- ¿Qué es Kotlin?

- Es un lenguaje de programación fuertemente tipado, desarrollado por JetBrains.
- Ha recibido influencias de lenguajes como Groovy, Scala o C#.
- Permite generar código para la JVM (Java Virtual Machine), JavaScript y en las últimas versiones también para ejecutables nativos.
- Posee muchas menos características que Scala pero es mucho más familiar y tiene mejor tiempo de compilación que este.

1.- ¿Qué es Kotlin?

HISTORIA DE KOTLIN

- Kotlin fue creado en 2010 por JetBrains, la empresa que desarrolló uno de los mejores IDE's que existen actualmente para Java, IntelliJ IDEA.
- Todos los IDE's de JetBrains están escritos en Java. Su mayor problema era su gran base de datos, ya que Java es un lenguaje genérico y poco conciso, por lo que pensaron que utilizar características de lenguajes modernos podrían ayudarles.
- Su mejor opción era Scala, pero lo descartaron por cuestiones de eficiencia y por ser demasiado potente para la solución que buscaban.

1.- ¿Qué es Kotlin?

HISTORIA DE KOTLIN

- Una vez descartado Java, no querían salir del mundo de JVM y decidieron que la mejor opción era crear su propia versión mejorada de este lenguaje, Kotlin.
- Inicialmente fue creado para desarrollar aplicaciones de escritorio (mercado de JetBrains), pero en la actualidad es un lenguaje multiplataforma además de ser lenguaje oficial de Android desde que lo anunció Google en 2017.

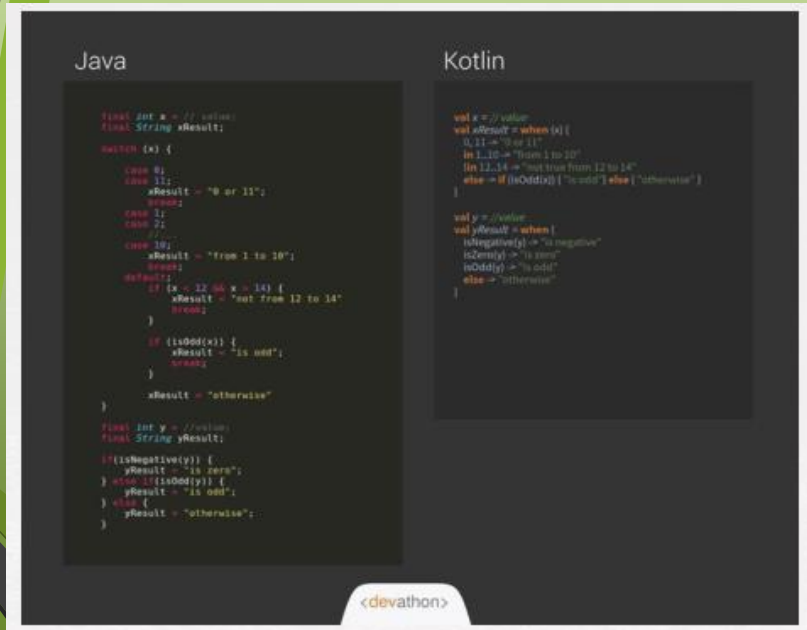
1.- ¿Qué es Kotlin?

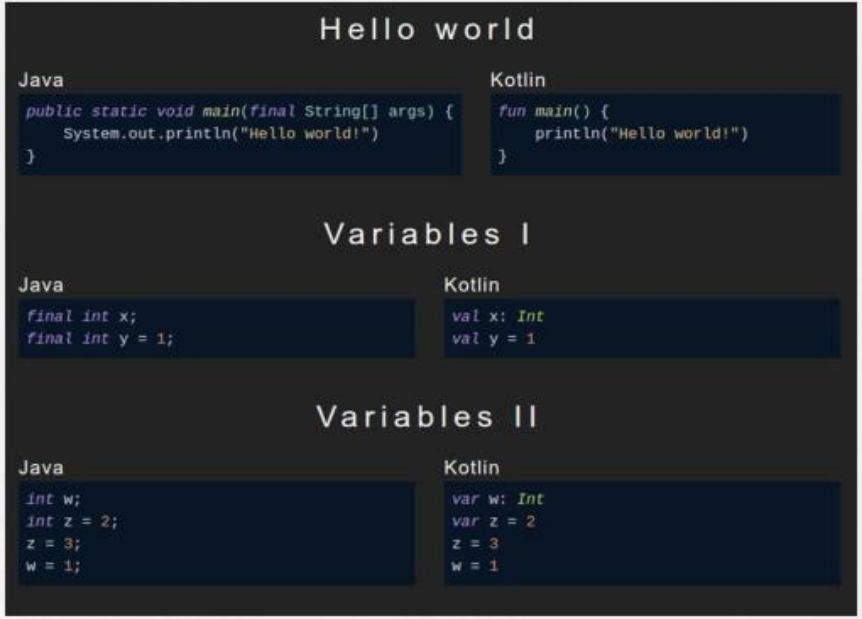
CARACTERÍSTICAS DE KOTLIN

- **Sintaxis:** Simplifica la lectura del código y su desarrollo respecto a Java. Esto ha influido en el incremento de su popularidad en el desarrollo de aplicaciones móviles.
- **Control de nulos:** Kotlin dispone de un mecanismo para controlar los "Null Pointer Exception", un error muy común en Java.
- **Curva de aprendizaje:** Su sintaxis simple y su parecido con otros lenguajes como Java hacen que la curva de aprendizaje sea bastante sencilla.
- **Comunidad:** La comunidad Kotlin está muy enfocada en el "open source", por lo que podemos encontrar gran cantidad de documentación y código libre.

1.- ¿Qué es Kotlin?

KOTLIN VS JAVA





1.- ¿Qué es Kotlin?

EMPRESAS QUE USAN KOTLIN

PMDM

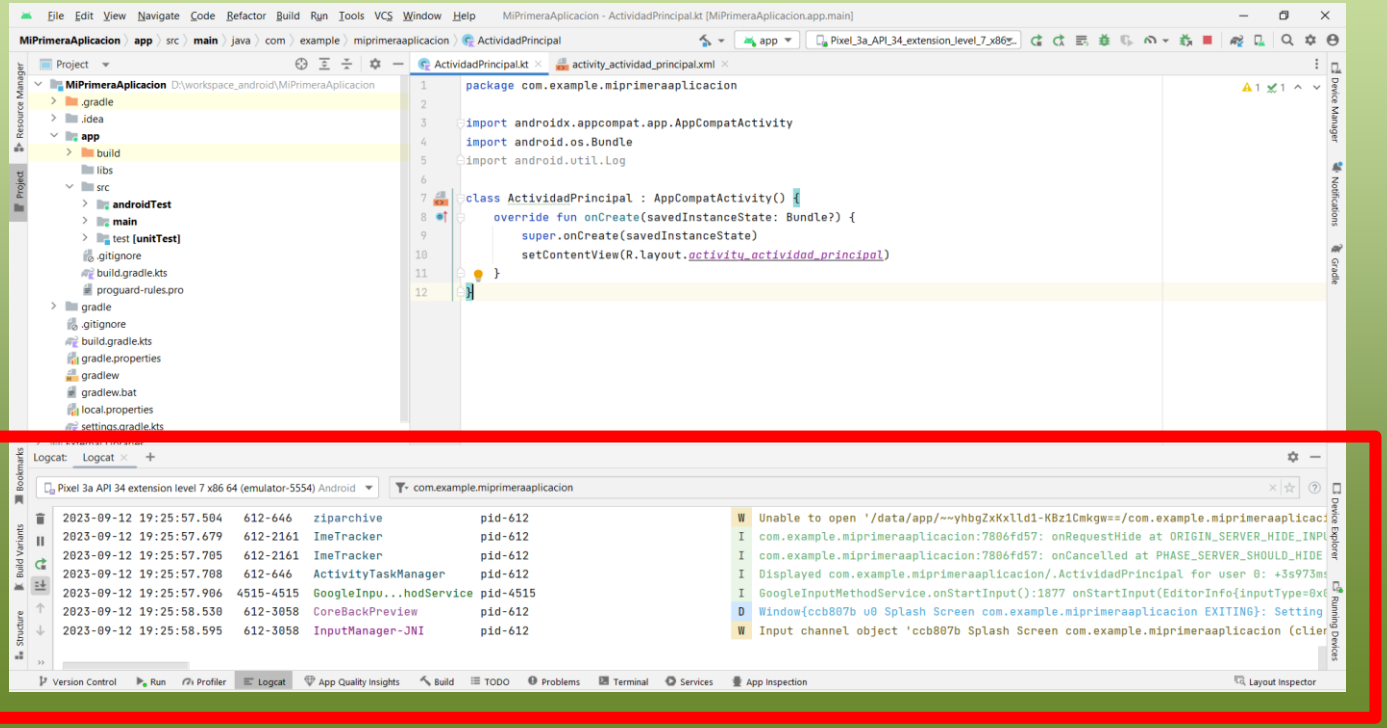
Introducción al lenguaje de programación Kotlin

2. Uso de Logcat en Android Studio

2.- Uso de Logcat en Android Studio

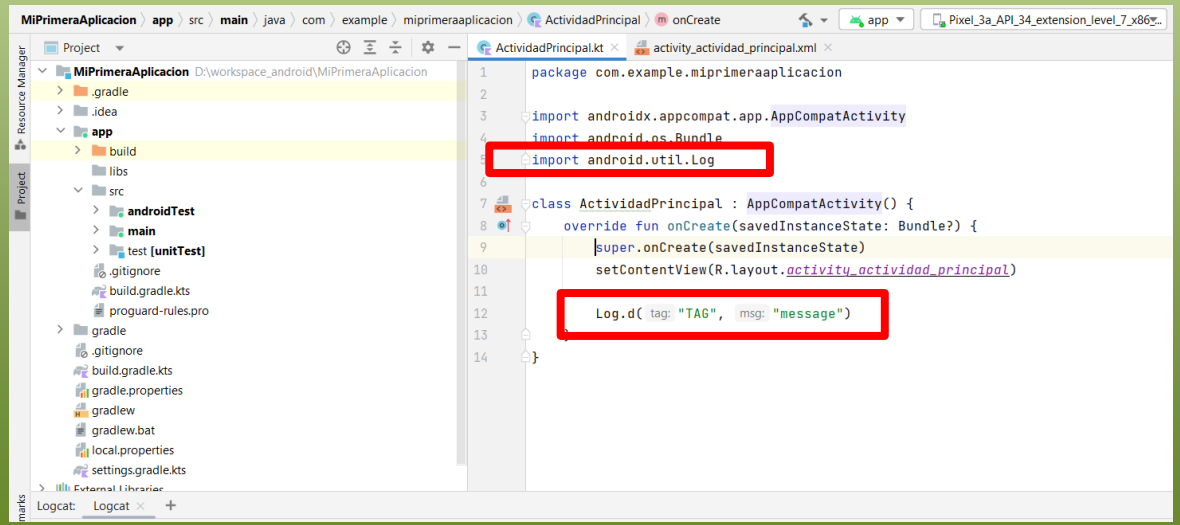
- **Logcat** es una herramienta de línea de comandos que posee Android Studio, la cual se encarga de volcar un registro de mensajes del sistema en una consola (ventana de Logcat) al ejecutar una aplicación, incluidos los que escribes desde tu app con la clase **Log**.
- Para abrir la ventana de Logcat en Android Studio haremos click en **View -> Tool Windows -> Logcat**.

2.- Uso de Logcat en Android Studio



2.- Uso de Logcat en Android Studio

- El envío de mensajes a la consola de Logcat se realiza mediante la clase **Log** del paquete **android.util**. Es necesario importar la clase Log en la clase Kotlin donde la usemos:



2.- Uso de Logcat en Android Studio

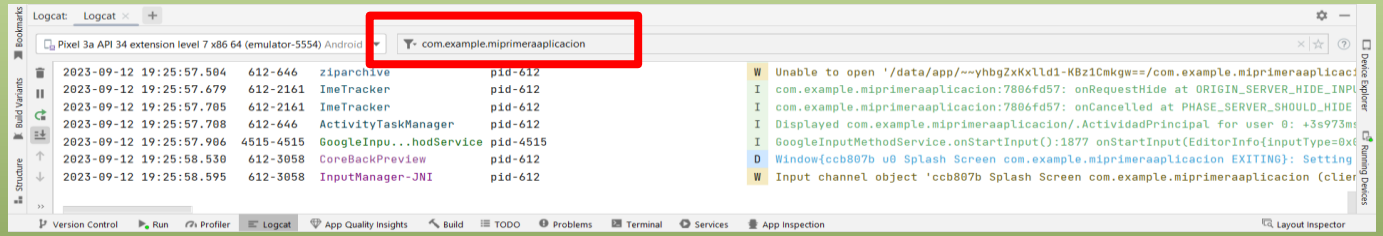
- La clase Log dispone de varios métodos en función del tipo de mensaje que queramos mostrar en la consola.
- Ejemplos:
  - *Log.d ("TAG", "Este es un mensaje de debug")*
  - *Log.i ("TAG", "Este es un mensaje de info")*
  - *Log.e ("TAG", "Este es un mensaje de error")*
- A cualquier método que usemos le debemos pasar dos parámetros:
  - La **etiqueta** o "**tag**" asociada al mensaje (de tipo String?)
  - El **mensaje** o **"msg"** que queremos mostrar (de tipo String)

2.- Uso de Logcat en Android Studio

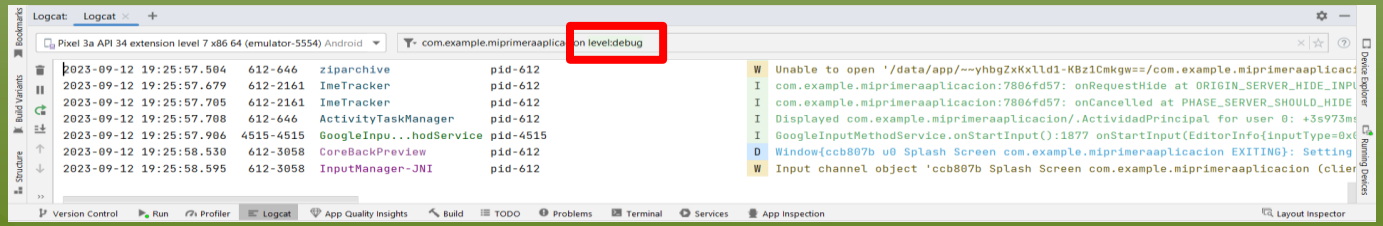
- El método de la clase Log que usemos dependerá de la prioridad del mensaje que queramos mostrar.
- Las diferentes prioridades existentes ordenadas de la prioridad más baja a la prioridad más alta son las siguientes:
  - V: Verbose (prioridad más baja)
  - D: Debug
  - I: Info
  - W: Warning
  - E: Error
  - F: Fatal
  - S: Silent (prioridad más alta en la que nunca se imprime nada)

2.- Uso de Logcat en Android Studio

- Para visualizar los mensajes que lanzamos desde una clase Kotlin, debemos indicar el paquete donde se encuentra nuestra clase en el campo de filtrado de la ventana de Logcat:

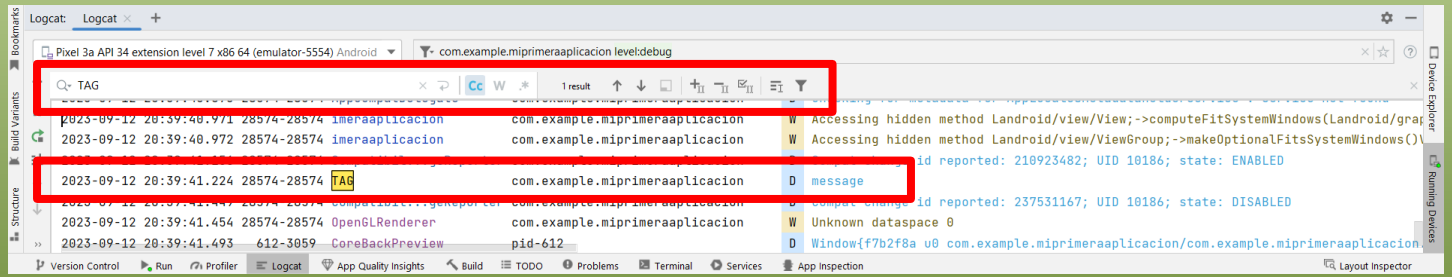


- Adicionalmente, también podemos filtrar los mensajes según el nivel de prioridad que queramos mostrar con la sintaxis **level:prioridad**:



2.- Uso de Logcat en Android Studio

- Algo que nos puede resultar muy útil es la búsqueda de nuestros mensajes en la consola a través de la etiqueta o tag que le hemos asociado previamente, la cual hemos pasado por parámetros al método usado de la clase Log.
- Para abrir el **buscador** en la ventana de Logcat debemos pulsar **Ctrl + F**:



- *<u>Nota</u>: Es recomendable limpiar la consola de Logcat antes de ejecutar de nuevo una aplicación para que no se mezclen los mensajes de ejecuciones diferentes.*

PMDM

Introducción al lenguaje de programación Kotlin

3. Programación básica en Kotlin

3. Programación básica en Kotlin

COMENTARIOS

- Para comentar una línea de código en Kotlin se utiliza **//**.
- Ejemplo:
  - *//Este log muestra un saludo*
  - *Log.d ("TAG", "¡Hola!")*
- En el caso de que queramos comentar un bloque completo de código utilizaremos **/* */.**
- Ejemplo:
  - */* Log.d ("TAG", "primer mensaje")*
    *Log.d ("TAG", "segundo mensaje") */*

3. Programación básica en Kotlin

VARIABLES Y CONSTANTES

- Una variable se declara mediante la palabra reservada **var,** como se muestra en los siguientes ejemplos:
  - *var nombre: String*
  - *var edad: Int*
- Una vez que tenemos la variable declarada podemos asignarle un valor con el símbolo **=**:
  - *nombre = "Pepe"*
  - *edad = 25*
- También se puede asignar el valor a la variable en el momento de su declaración. Si se realiza de esta forma, Kotlin permite que no se indique el tipo de la variable (aunque siempre es recomendable hacerlo):
  - *var nombre: String = "Pepe" //Indicamos el tipo de la variable*
  - *var nombre = "Pepe" //Kotlin interpreta automáticamente que la variable nombre es de tipo String*

3. Programación básica en Kotlin

VARIABLES Y CONSTANTES

- Para declarar una constante utilizaremos la palabra reservada **val**.
- Una constante es una variable a la que no se le puede modificar el valor que almacena una vez se ha inicializado con un valor. En Kotlin es la única diferencia que hay que tener en cuenta entre variables y constantes. El resto de operaciones con constantes (declaración, asignación de valores o inicialización) las haremos de la misma forma que las hacemos con una variable.
- Ejemplos:
  - *val nombre: String*
  - *nombre = "Pepe"*
  - *val edad: Int = 25*
  - *val edad = 25*

3. Programación básica en Kotlin

VARIABLES Y CONSTANTES

- Si queremos imprimir una variable o constante en la consola de LogCat, debemos pasarla por parámetros en el argumento msg del método Log que utilicemos:
  - *Log.d ("TAG", nombre)*
  - *Log.d ("TAG", edad.toString())* *//El mensaje que le pasamos a un método Log es de tipo String, por lo //que si queremos imprimir una variable de otro tipo tenemos que //utilizar el método toString()*
- También podemos pasar una variable o constante implícita en el mensaje usando el símbolo **$**:
  - *Log.d ("TAG", "La edad es $edad")*
  - *Log.d ("TAG", "La edad es ${edad.toString()}")*

3. Programación básica en Kotlin

TIPOS DE DATOS

- Los tipos de datos en Kotlin son los siguientes:
  - **Números enteros**:
    - Byte -> 8 bits (1 byte). Rango: -128 a 127
    - Short -> 16 bits (2 bytes). Rango: -32768 a 32767
    - Int -> 32 bits (4 bytes). Rango: -2.147.483.648 a 2.147.483.647
    - Long -> 64 bits (8 bytes). Rango: -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
  - **Números decimales**:
    - Float -> 32 bits (4 bytes). Precisión: hasta 7 decimales
    - Double -> 64 bits (8 bytes). Precisión: hasta 16 decimales
  - **Valores booleanos:**
    - Boolean **->** true o false
  - **Cadenas de texto:**
    - String

3. Programación básica en Kotlin

TIPOS DE DATOS

- Ejemplos:
  - *var nombre: String = "Pepe"*
  - *var edad: Int = 5*
  - *var precio: Double = 15.50*
  - *var altura: Float = 1.73f* *//El tipo de datos float debe indicarse con f o F*
  - *var mayorDeEdad: Boolean = true*

3. Programación básica en Kotlin

OPERADORES

- Dentro de la variedad de operadores que nos proporciona Kotlin, destacamos los siguientes:
  - **Aritméticos**: +, -, *, /, %
  - **Incremento y Decremento**: ++, --
  - **Relacionales**: ==, !=, <, >, <=, >=
  - **Lógicos**: &&, ||, !

3. Programación básica en Kotlin

- **Ejercicio 01:** Realiza un programa en Kotlin que muestre por pantalla el nombre de una asignatura, la nota del primer examen, la nota del segundo examen y la nota media de la asignatura. Para ello deberás almacenar previamente el nombre y las notas de la asignatura en variables del tipo de datos adecuado.

- Muestra el resultado en la consola de Logcat como en el siguiente ejemplo:

Asignatura: Matemáticas
Nota media: 8.5

3. Programación básica en Kotlin

- **Ejercicio 02:** Suponemos que llevas en tu cartera 130 euros en billetes de 50 euros y de 10 euros. Realiza un programa en Kotlin que, partiendo de almacenar los 130 euros de tu cartera en una variable, sea capaz de decirte cuantos billetes tienes de cada tipo.

- Muestra el resultado en la consola de Logcat como en el siguiente ejemplo:

130 euros hacen un total de: 2 billetes de 50 euros y 3 billetes de 10 euros

3. Programación básica en Kotlin

NULL SAFETY

- Como hemos mencionado anteriormente, Kotlin proporciona un mecanismo para evitar que las variables que no tengan valor o tengan valor nulo causen excepciones del tipo NullPointerException, algo que pasa muy a menudo en otros lenguajes de programación como Java.
- En Kotlin, ninguna variable que declaremos como lo hemos hecho hasta ahora puede inicializarse o tomar un valor nulo. Los siguientes ejemplos darían errores de compilación:
  - *var nombre: String = null*
  - *var edad: Int = null*
- Para que una variable pueda almacenar valores nulos es necesario indicarlo con el símbolo **?** detrás del tipo de datos, en el momento de su declaración. Los siguientes ejemplos no darían errores de compilación:
  - *var nombre: String? = null*
  - *var edad: Int? = null*

3. Programación básica en Kotlin

NULL SAFETY

- Si queremos operar con una variable que hemos declarado como "posible" nullable, el compilador nos avisará para que controlemos este caso y evitar así que se pueda producir una excepción de tipo NullPointerException.
- ¿Cómo controlamos estos casos? El mecanismo que nos proporciona Kotlin es el siguiente:

```
nombre?.let {
    Log.d ("TAG", nombre)  //Se ejecuta en el caso de que nombre sea distinto de null
} ?: run {
    Log.d ("TAG", "La variable nombre no tiene valor o tiene un valor nulo")
    //Se ejecuta en el caso de que nombre sea null
}
```

3. Programación básica en Kotlin

- **Ejercicio 03:** Realiza un programa en Kotlin que almacene en una variable un nombre de usuario y en otra variable nullable un password. El programa debe controlar, mediante el mecanismo de null safety, que se almacene un valor en la variable del password para dar la bienvenida al usuario.

- Muestra el resultado en la consola de Logcat de la siguiente forma (prueba ambos casos):

1. Si password contiene un valor:

   Bienvenido nombreUsuario
   El password introducido es passwordUsuario

2. Si password es "null":

   Error: nombreUsuario no ha introducido el password requerido

3. Programación básica en Kotlin

FUNCIONES

- Una función en Kotlin se declara utilizando la palabra reservada **fun**. La sintaxis para declarar una función es la siguiente:

```
fun nombreFuncion () {
    //código de la función
}
```

- La función que hemos declarado anteriormente tiene un ámbito público, por lo que se podría invocar desde cualquier parte de la aplicación. Si queremos que la función únicamente se pueda invocar desde la clase en la que la hemos declarado debemos utilizar el modificador **private**:

```
private fun nombreFuncion () {
    //código de la función
}
```

- La invocación de una función se realiza con el nombre de dicha función:

```
nombreFuncion ()
```

3. Programación básica en Kotlin

FUNCIONES

- El paso de parámetros a una función se indica en la declaración mediante el nombre del parámetro y el tipo. Si la función espera más de un parámetro, estos se separan por comas:

```
private fun datosAlumno (nombre: String, edad: Int) {
    Log.d ("TAG", "Nombre: $nombre")
    Log.d("TAG", "Edad: $edad")
}
```

- Si queremos que la función devuelva un valor debemos indicar el tipo del valor retornado en la cabecera de la función:

```
private fun calcularEdad (anioNacimiento: Int): Int {
    return 2023 - anioNacimiento
}
```

3. Programación básica en Kotlin

FUNCIONES

- El resultado de la función anterior se podría almacenar en una variable del mismo tipo retornado por la función:

  var edad: Int = calcularEdad (1996)

- Kotlin permite simplificar la sintaxis en la declaración de funciones que retornan algún valor y tienen una sola línea de código en el cuerpo de la función. Por tanto, la función *calcularEdad* que hemos visto anteriormente se podría declarar de la siguiente forma:

  private fun calcularEdad (anioNacimiento: Int): Int = 2023 - anioNacimiento

- Esta función se invocaría de la misma forma que antes:

  var edad: Int = calcularEdad (1996)

3. Programación básica en Kotlin

Ejercicio 04: Crea tres variables que almacenen números enteros (num1, num2 y num3). Posteriormente, implementa en Kotlin las siguientes funciones y prueba su funcionamiento con las variables que te has creado previamente:

1. Función **mostrarNumeros** que reciba num1, num2 y num3 por parámetros y los muestre por pantalla.
2. Función **sumaNumeros** que reciba num1, num2 y num3 por parámetros, almacene la suma de los tres números en una variable y la muestre por pantalla.
3. Función **calcularMedia** que reciba num1, num2 y num3 por parámetros y devuelva la media aritmética de los tres números. Este valor se debe almacenar en una variable que se mostrará posteriormente por pantalla.
4. Función **operaciones** que reciba la media de los tres números y num3 por parámetros, almacene la media multiplicada por num3 en una variable y la muestre por pantalla.

3. Programación básica en Kotlin

CLASES

- Una clase la declaramos utilizando la palabra reservada **class**:

```
class Persona () {
    //variables y constantes
    //funciones
}
```

- Para crear una variable o constante de la clase que hemos declarado anteriormente, lo haríamos de la siguiente forma:

```
var alumno = Persona()
```

- Podemos crearnos tantas variables o constantes como queramos de una misma clase.

3. Programación básica en Kotlin

CLASES

- Dentro de una clase podemos declarar funciones para implementar las funcionalidades de esa clase:

```
class Persona () {
        fun presentacion() {
                Log.d ("TAG", "¡Hola! Me presento")
        }
}
```

- *Nota: Las funciones que implementamos en una clase deben ser públicas para poder acceder a ellas desde fuera de la clase.*
- También podemos declarar variables y constantes en el cuerpo de una clase. Estas variables o constantes pertenecen al ámbito de esta clase, por lo que se podrán utilizar en cualquier función pero únicamente dentro de esa misma clase:

```
class Persona () {
        val logTag: String = "TAG"
        fun presentacion() {
                Log.d (logTag, "¡Hola! Me presento")
        }
}
```

3. Programación básica en Kotlin

CLASES

- Para realizar el paso de parámetros que utilizarán las funciones de una clase tenemos dos opciones:
  - Pasar los parámetros directamente a la función: Lo haremos de esta forma cuando los parámetros solo se vayan a utilizar en esa función. Ejemplo:

```
class Persona () {
    fun presentacion(nombre: String, edad: Int) {
        Log.d ("TAG", "¡Hola! Mi nombre es $nombre y mi edad es $edad")
    }
}
```

3. Programación básica en Kotlin

CLASES

- Pasar los parámetros a la clase: Lo haremos de esta forma cuando los parámetros sean comunes para todas las funciones de la clase. Ejemplo:

```
class Persona (private val nombre: String, private val edad: Int) {
    fun presentacion() {
        Log.d ("TAG", "¡Hola! Mi nombre es $nombre y mi edad es $edad")
    }
    fun mostrarEdad() {
        Log.d ("TAG", "Tengo $edad años")
    }
}
```

*Nota: Si lo hacemos de esta forma tenemos que indicar la palabra clave **var/val** junto a cada parámetro, además del modificador **private** para especificar que esos parámetros solo se van a utilizar en esa clase.*

3. Programación básica en Kotlin

CLASES

- En los siguientes ejemplos podemos ver cómo invocar a las funciones que hemos implementado en nuestras clases:

➢ Pasando los parámetros a la función:

```
val alumno = Persona ()
val nombreAlumno = "Pepe"
val edadAlumno = 26
alumno.presentacion (nombreAlumno, edadAlumno)
```

➢ Pasando los parámetros a la clase:

```
val nombreAlumno = "Pepe"
val edadAlumno = 26
val alumno = Persona (nombreAlumno, edadAlumno)
alumno.presentacion()
```

3. Programación básica en Kotlin

DATA CLASSES

- En Kotlin tenemos otro tipo de clases, llamadas data classes, cuyo objetivo principal es el de almacenar datos a modo de paquete. Para declarar una clase de datos utilizamos la palabra reservada **data class**:

```
data class DatosPersona (
    val nombre: String,
    val edad: Int
)
```

- Para crear una variable o constante de la clase de datos que hemos declarado anteriormente lo haríamos de la siguiente forma :

```
val datosAlumno = DatosPersona ("Pepe", 26)
```

- Igual que ocurre con las clases, podemos crearnos tantas variables o constantes como queramos de una misma clase de datos.

3. Programación básica en Kotlin

DATA CLASSES

- Por ejemplo, podemos usar los data classes para agrupar los datos que pasamos por parámetro a una clase o a una función:

```
//Declaración de la clase

class Persona (private val data: DatosPersona) {
        fun presentacion() {
                Log.d ("TAG", "Mi nombre es ${data.nombre} y mi edad es ${data.edad}")
        }


//Utilización

val datosAlumno = DatosPersona ("Pepe", 26)
val alumno = Persona (datosAlumno)
alumno.presentación()
```

3. Programación básica en Kotlin

INTERFACES

- Una interfaz es una clase abstracta que sirve para extender la funcionalidad de una clase. En una interfaz únicamente vamos a declarar las funciones (solo la cabecera de cada función) sin añadir la implementación.
- La declaración de una interfaz la hacemos utilizando la palabra reservada **interface**:

```
interface InterfazPersona {
    fun mostrarEdad()
}
```

3. Programación básica en Kotlin

INTERFACES

- Una vez creada la interfaz podemos indicar que cualquiera de las clases que nos hemos creado la implemente. Esto nos obliga a implementar en estas clases todas las funciones que hemos declarado en la interfaz, sobrescribiéndolas mediante la palabra reservada **override**:

```
class Persona (private val nombre: String, private val edad: Int): InterfazPersona {
    fun presentacion() {
        Log.d ("TAG", "¡Hola! Mi nombre es $nombre y mi edad es $edad")
    }
    override fun mostrarEdad() {
        Log.d ("TAG", "Tengo $edad años")
    }
}
```

- **Ejercicio 05:** Crea un nuevo proyecto con el nombre de **ProgramadorApp**. En esta aplicación, implementa en Kotlin:

1. Una clase llamada **Programador**.
2. Una interfaz llamada **ProgramadorInterface**.
3. Una clase de datos llamada **ProgrammerData.**

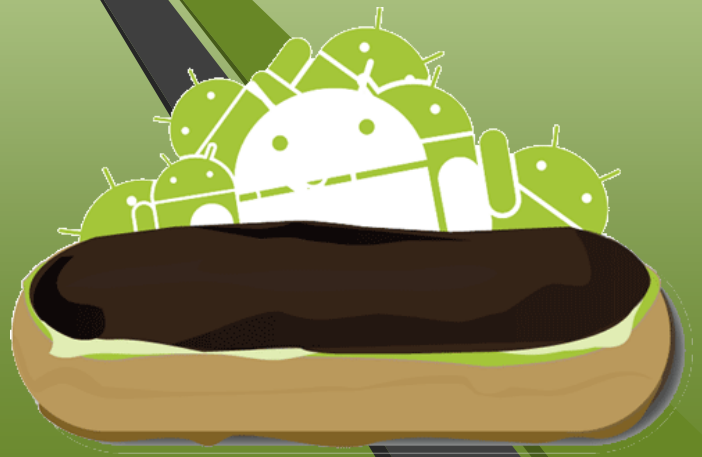
- La aplicación debe seguir las siguientes reglas:

1. La clase de datos debe incluir tres constantes llamadas **name, age** y **language**.
2. La interfaz debe tener una función llamada **getProgrammerData,** que devolverá un objeto de tipo ProgrammerData.
3. La clase Programador debe implementar la interfaz anterior. En esta clase también se deben implementar tres funciones privadas que se encarguen de devolver el nombre, la edad y el lenguaje de programación del programador.
4. La función getProgrammerData debe devolver un objeto de tipo ProgrammerData con los datos de las tres funciones privadas creadas anteriormente.

- *Nota: Si hemos realizado correctamente la implementación, desde un objeto de la clase Programador únicamente podremos acceder a la función getProgrammerData.*

PMDM

Introducción al lenguaje de programación Kotlin



4. Estructuras de control

4. Estructuras de control

ESTRUCTURAS CONDICIONALES

- A estas estructuras también se les conoce como estructuras de selección. Nos permiten variar el flujo de un programa en función de unas determinadas condiciones.
- En Kotlin, estas estructuras pueden ser de tres tipos:
  - Simples*: **if***
  - Dobles*: **if…else***
  - Múltiples*: **when***

4. Estructuras de control

ESTRUCTURAS CONDICIONALES SIMPLES

- En este tipo de estructuras condicionales la sentencia (o sentencias) únicamente se ejecutarán si se cumple la condición (o condiciones). En caso contrario, el programa sigue su curso sin ejecutar esta sentencia. Se utiliza la palabra reservada **if**.
- Ejemplo:

```
if (edad >= 18) {
    Log.d ("TAG", "Es mayor de edad")
}
```

- *Nota: Si el if tiene una sola sentencia es opcional el uso de las llaves.*

4. Estructuras de control

ESTRUCTURAS CONDICIONALES DOBLES

- En este tipo de estructuras condicionales, se ejecutarán unas sentencias u otras en función de si se cumple la condición o no. Se utilizan las palabras reservadas **if…else**.
- Ejemplo:

```
if (lenguaje == "Kotlin") {
    Log.d ("TAG", "El lenguaje de programación elegido es Kotlin")
} else {
    Log.d ("TAG", "El lenguaje de programación no es Kotlin")
}
```

4. Estructuras de control

ESTRUCTURAS CONDICIONALES DOBLES

- Kotlin permite asignar el valor resultado de este tipo de estructuras (if...else) directamente a una variable.
- Ejemplo:

```
var edad: Int = 25

var esMayorDeEdad: Boolean = if (edad >= 18) {
    true
} else {
    false
}
```

4. Estructuras de control

ESTRUCTURAS CONDICIONALES MÚLTIPLES

- Cuando tenemos muchas condiciones y queremos evitar el uso de varios if...else anidados, se utiliza una estructura condicional múltiple. En Kotlin, este tipo de estructuras se definen con **when**.
- Ejemplo:

```
var lenguaje: String = "Kotlin"
when (lenguaje) {
    "Kotlin" -> Log.d ("TAG", "Se ha seleccionado Kotlin")
    "Java" -> Log.d ("TAG", "Se ha seleccionado Java")
    "Python" -> Log.d ("TAG", "Se ha seleccionado Python")
    else -> Log.d ("TAG", "Se ha seleccionado otro lenguaje")
}
```

4. Estructuras de control

ESTRUCTURAS CONDICIONALES MÚLTIPLES

- Si queremos añadir más de una sentencia en una condición de la estructura when debemos hacerlo entre llaves.
- Ejemplo:

```
var lenguaje: String = "Kotlin"
when (lenguaje) {
    "Kotlin" -> {
        Log.d ("TAG", "Se ha seleccionado Kotlin")
        Log.d ("TAG", "Esta es su primera opción")
    }
    "Java" -> Log.d ("TAG", "Se ha seleccionado Java")
}
```

4. Estructuras de control

ESTRUCTURAS CONDICIONALES MÚLTIPLES

- La estructura when es mucho más flexible que estructuras condicionales múltiples de otros lenguajes, como por ejemplo switch en Java. Algunas posibilidades que nos ofrece esta estructura son:
  - Podemos añadir varios valores en una misma condición separados por coma:

```
var lenguaje: String = "Kotlin"
when (lenguaje) {
    "Kotlin", "Java" -> Log.d ("TAG", "Se ha seleccionado Kotlin o Java")
    "Python" -> Log.d ("TAG", "Se ha seleccionado Python")
}
```

4. Estructuras de control

ESTRUCTURAS CONDICIONALES MÚLTIPLES

- Si utilizamos tipos numéricos se pueden establecer rangos en una condición:

```
var numero: Int = 10
when (numero) {
    0 -> Log.d ("TAG", "El número es 0")
    1 -> Log.d ("TAG", "El número es 1")
    in 5..15 -> Log.d ("TAG", "El número está entre 5 y 15")
}
```

4. Estructuras de control

ESTRUCTURAS CONDICIONALES MÚLTIPLES

- También podemos usar when como un if para establecer comparaciones en las condiciones:

```
var numero: Int = 10
when {
    numero == 0 -> Log.d ("TAG", "El número es 0")
    numero > 1 -> Log.d ("TAG", "El número es mayor que 1")
    numero in 5..15 -> Log.d ("TAG", "El número está entre 5 y 15")
}
```

4. Estructuras de control

- **Ejercicio 06:** Crea un programa en Kotlin que almacene en una variable la nota de un alumno (número entero entre 0 y 10) y muestre en la consola de Logcat su calificación según el valor de la nota almacenada:
  - 0 a 4 = Suspenso.
  - 5 a 6 = Bien.
  - 7 a 8 = Notable.
  - 9 a 10 = Sobresaliente.

4. Estructuras de control

- **Ejercicio 07:** Diseña un programa en Kotlin que almacene un número del 1 al 4 en una variable (opción) y dos números enteros (num1 y num2). Posteriormente el programa, en función de la opción almacenada, realizará las siguientes operaciones:
  - 1 – Sumar num1 y num2.
  - 2 – Restar num1 y num2.
  - 3 – Multiplicar num1 y num2.
  - 4 – Dividir num1 entre num2.
- *Nota: Implementa cada una de las operaciones anteriores en una función.*

4. Estructuras de control

LISTADOS

- Un listado es un conjunto de valores ordenados que tienen el mismo tipo de datos. Cada elemento del listado tiene asociado un índice que identifica de manera unívoca su posición dentro del listado.
- Hay dos tipos de listados principales en Kotlin que se utilizan en Android:
  - *List -> Listados inmutables (no se puede modificar su contenido)*
  - *ArrayList -> Listados mutables (se pueden añadir y modificar valores, ordenar, etc.).*

4. Estructuras de control

LISTADOS (LIST)

- Un listado de tipo **List** se puede declarar de dos formas:
  1. Indicando el tipo de listado:

     var list: List<String> = listOf()

  2. Indicando el tipo de valores del listado:

     var list = listOf<String>()

- Para inicializar con valores este tipo de listados utilizamos el método *listOf*:
  1. *var list: List<String> = listOf("Uno", "Dos", "Tres")*
  2. *var list = listOf<String>("Uno", "Dos", "Tres")*

4. Estructuras de control

LISTADOS (LIST)

- Si queremos obtener un valor de un listado de tipo List lo podemos hacer:
  1. Accediendo al valor indicando su índice entre corchetes:

     var listValue: String = list [o]
  2. Utilizando el método **get(index):**

     var listValue: String = list.get(o)
- Otros métodos que podemos usar con este tipo de listados son:
  - ***size*** *-> Devuelve el tamaño del listado*
  - ***indexOf(element)*** *-> Devuelve el índice del valor especificado.*
  - ***contains(element)*** *-> Devuelve true si el valor especificado está en el listado, false en caso contrario.*
  - *Etc.*

4. Estructuras de control

LISTADOS (ARRAYLIST)

- Un listado de tipo **ArrayList** se puede declarar de dos formas:
  1. Indicando el tipo de listado:

     var arrayList: ArrayList<String> = arrayListOf()

  2. Indicando el tipo de valores del listado:

     var arrayList = arrayListOf<String>()

- Para inicializar con valores este tipo de listados utilizamos el método *arrayListOf*:
  1. *var arrayList: ArrayList<String> = arrayListOf("Uno", "Dos", "Tres")*
  2. *var arrayList = arrayListOf<String>("Uno", "Dos", "Tres")*

4. Estructuras de control

LISTADOS (ARRAYLIST)

- Si queremos obtener un valor de un listado de tipo ArrayList lo podemos hacer:
  1. Accediendo al valor indicando su índice entre corchetes:

     var arrayListValue: String = arrayList [o]
  2. Utilizando el método **get(index):**

     var arrayListValue: String = larrayList.get(o)
- Si queremos añadir un valor en este tipo de listados lo podemos hacer:
  1. Insertando el valor indicando su índice entre corchetes:

     arrayList [o] = "Cero"
  2. Utilizando el método **add(index, element):**

     arrayList.add(o, "Cero")

4. Estructuras de control

LISTADOS (ARRAYLIST)

- Otros métodos que podemos usar con este tipo de listados son:
  - ***size*** *-> Devuelve el tamaño del listado*
  - ***remove(element)*** *-> Elimina el valor especificado del listado*
  - ***removeAt(index)*** *-> Elimina el valor de la posición indicada.*
  - ***isEmpty()*** *-> Devuelve true si el listado está vacío, false en caso contrario.*
  - ***set(index, element)*** *-> Modifica el valor especificado en la posición indicada.*
  - ***contains(element)*** *-> Devuelve true si el valor especificado está en el listado, false en caso contrario.*
  - *Etc.*

4. Estructuras de control

ESTRUCTURAS REPETITIVAS

- A estas estructuras también se les conoce como bucles. Nos permiten ejecutar una sentencia (o sentencias) mientras se cumpla una determinada condición.
- En Kotlin, estas estructuras pueden ser de tres tipos:
  - ***for***
  - ***while***
  - ***do-while***

4. Estructuras de control

ESTRUCTURAS REPETITIVAS (FOR)

- Este bucle lo utilizaremos cuando queramos que las sentencias se ejecuten un número conocido y fijo de veces.
  - Si queremos acceder directamente a todos los elementos de un listado utilizaremos el modo conocido en otros lenguajes, por ejemplo en Java, como **for-each**:

```
for (nombre: String in arrayList) {
    Log("TAG", "Nombre: $nombre")
}
```

Nota: Esta forma de recorrer un listado se utiliza mucho con tipos String.

4. Estructuras de control

ESTRUCTURAS REPETITIVAS (FOR)

- Para recorrer un listado por posición debemos especificar un intervalo. En el siguiente ejemplo se recorrería el listado arrayList desde la posición *0* a la *5* (ambas incluidas):

```
for (position: Int in 0..5) {
    Log("TAG", "Nombre: ${arrayList.get(position)}")
}
```

- Si no queremos llegar hasta el tamaño del listado y así evitar que se produzcan excepciones al intentar acceder a la posición de un índice que no existe, se puede utilizar la palabra reservada **until**. En el siguiente ejemplo se recorrería el listado arrayList desde la posición *0* hasta *arrayList.size – 1:*

```
for (position: Int in 0 until arrayList.size) {
    Log("TAG", "Nombre: ${arrayList.get(position)}")
}
```

4. Estructuras de control

ESTRUCTURAS REPETITIVAS (FOR)

- También podemos recorrer un listado estableciendo saltos en un intervalo, utilizando la palabra reservada **step**. En el siguiente ejemplo se recorrería el listado arrayList desde la posición *0* a la *10* (ambas incluidas) de dos en dos:

```
for (position: Int in 0..10 step 2) {
    Log("TAG", "Nombre: ${arrayList.get(position)}")
}
```

- Mediante la palabra reservada **downTo,** Kotlin nos permite recorrer un listado de atrás hacia adelante. En el siguiente ejemplo se recorrería el listado arrayList desde la posición *10* hasta la *0* (ambas incluidas)*:*

```
for (position: Int in 10 downTo 0) {
    Log("TAG", "Nombre: ${arrayList.get(position)}")
}
```

4. Estructuras de control

ESTRUCTURAS REPETITIVAS (WHILE)

- Este bucle lo utilizaremos cuando queramos que las sentencias se ejecuten cero o más veces en función de una condición. La sintaxis del bucle while en Kotlin la podemos ver en el siguiente ejemplo:

```
var numero: Int = 1
while (numero <= 10) {
    Log("TAG", "Número: $numero")
    numero++
}
```

- En el ejemplo anterior se imprimen los números del 1 al 10.

4. Estructuras de control

ESTRUCTURAS REPETITIVAS (DO-WHILE)

- Este bucle lo utilizaremos cuando queramos que las sentencias se ejecuten al menos una vez y continúen ejecutándose mientras que la condición sea verdadera. La sintaxis del bucle do-while en Kotlin la podemos ver en el siguiente ejemplo:

```
var numero: Int = 1
do{
    Log("TAG", "Número: $numero")
    numero++
} while (numero <= 10)
```

- En el ejemplo anterior se imprimen los números del 1 al 10.

4. Estructuras de control

- **Ejercicio 08:** Escribe un programa en Kotlin que rellene un listado con los 80 primeros números primos. Posteriormente, muestra el listado en la consola de Logcat.

- Ejemplo: Contenido de un listado con los 80 primeros números primos

[0] -> 1; [1] -> 2; [2] -> 3; [3] -> 5; [4] -> 7; [5] -> 11; [6] -> 13…

4. Estructuras de control

- **Ejercicio 09:** Sabiendo que tenemos una clase de 6 alumnos (Pepe, Juan, Ana, Marta, Pedro y María), realiza un programa en Kotlin que cree un listado que almacene la nota media de cada alumno y, según la opción elegida almacenada previamente en una variable, realice las siguientes operaciones:
  - 1. Mostrar las notas almacenadas en el listado.
  - 2. Mostrar el nombre y la nota del alumno que tiene la media más alta (necesitarás utilizar otro listado con el nombre de los alumnos).
- *Nota: Implementa cada una de las operaciones de este ejercicio en una función.*

PMDM

Introducción al lenguaje de programación Kotlin

5. Control de errores

5. Control de errores

- El control de errores, o también conocido como **control de excepciones**, es un mecanismo que nos permite controlar los errores que se producen en nuestras apps en tiempo de ejecución.
- En Kotlin, la sintaxis para realizar un control de excepciones es la siguiente:

```
try {
    //Sentencia (o sentencias) que queremos proteger
} catch (e1: excepcion_1) {
    //Control de la excepción 1
} catch (e2: excepcion_2) {
    //Control de la excepción 2
} catch (e3: excepcion_n) {
    //Contron de la excepción n
} finally {
    //OPCIONAL; Este código se ejecuta siempre
}
```

5. Control de errores

- En el siguiente ejemplo se produce una excepción de tipo ***IndexOutOfBoundsException*** al recorrer el listado hasta su tamaño e intentar acceder al valor de un índice que no existe. Para evitar que se produzca un error de ejecución, controlamos el error mediante un control de excepciones:

```
val arrayList: ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)

try {
    for (position: Int in 0 ≤ .. ≤ arrayList.size) {
        Log.d( tag: "TAG", arrayList[position].toString())
    }
} catch (e: IndexOutOfBoundsException) {
    Log.d( tag: "TAG",  msg: "Error: IndexOutOfBoundsException in arrayList")
    e.printStackTrace()
} finally {
    Log.d( tag: "TAG",  msg: "Continúa el flujo de la aplicación")
}
```

- **Ejercicio 10:** Crea un nuevo proyecto con el nombre de **BotSeguridadApp**. En esta aplicación, implementa en Kotlin:

  1. Una función llamada **botSeguridad()** a la que se invocará desde el método onCreate() del MainActivity y en la que implementaremos la funcionalidad del Bot.
  2. Una clase de datos llamada **Persona**.

- La aplicación debe seguir las siguientes reglas :

1. La clase de datos Persona debe incluir tres constantes llamadas **name, age** y **hobbies** (que será un listado de Strings).
2. Se debe crear una variable de tipo Persona con nuestra información (nombre, edad y listado de hobbies).
3. Mediante el uso de estructuras condicionales, nuestro Bot debe realizar lo siguiente:
   - Mostrarnos un mensaje de error en el caso de que el nombre almacenado en la clase Persona no sea el nuestro y no dejarnos acceder (mostrando "Acceso denegado"). En caso contrario, el Bot mostrará un mensaje de éxito y continuará la ejecución de nuestro programa.
   - A continuación, el Bot controlará los siguientes rangos de edad:
     - Si tenemos entre 0 y 17 años, el Bot nos informará de que somos menores de edad y nos mostrará un mensaje de "Acceso denegado".
     - Si tenemos entre 65 y 100 años, el Bot nos denegará igualmente el acceso informándonos de que somos demasiado mayores.
     - En caso de que tengamos entre 18 y 64 años, el Bot nos dejará pasar y nos mostrará un mensaje de éxito. Posteriormente mostrará nuestro listado de hobbies por pantalla.