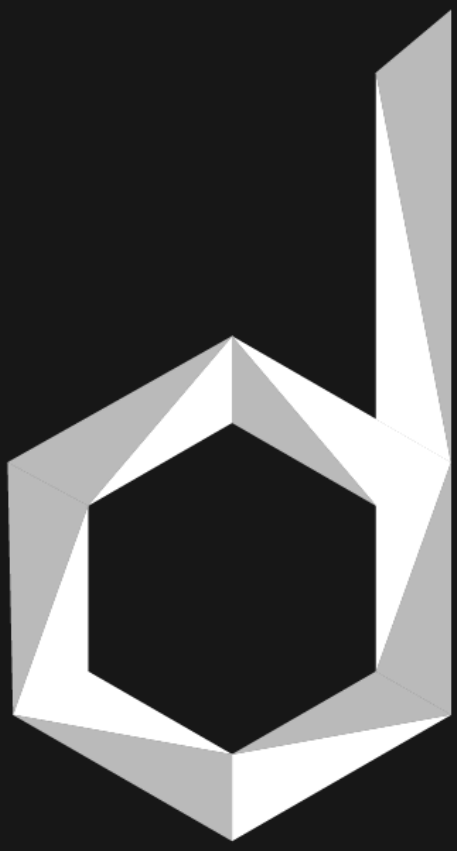
INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

DESARROLLO MÓVIL - ANDROID

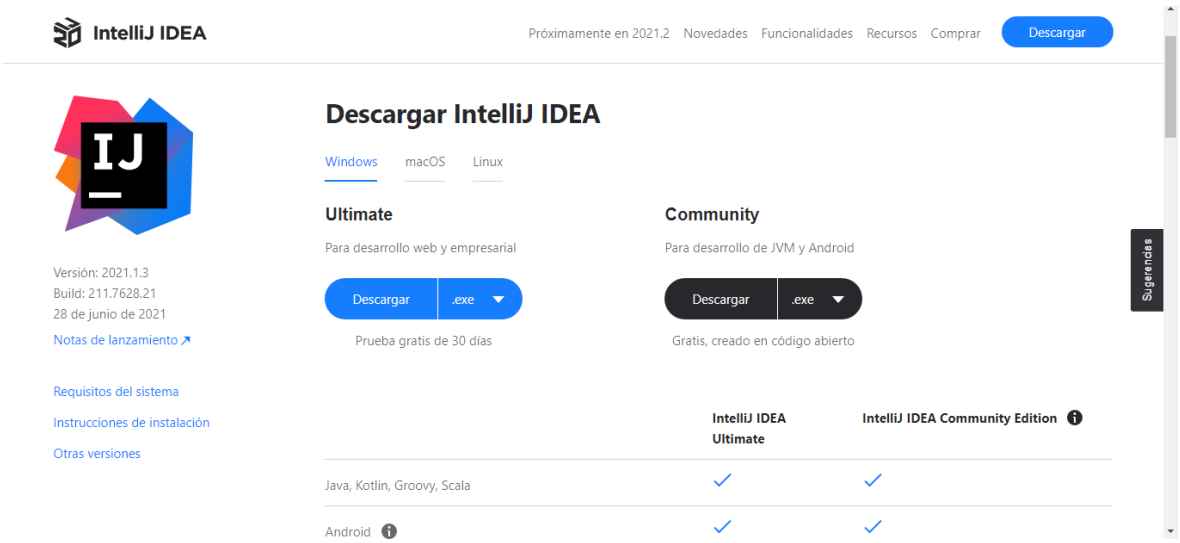
INTELLIJ IDEA

Kotlin: IntelliJ IDEA

Contenido

Introducción a IntelliJ IDEA y Kotlin

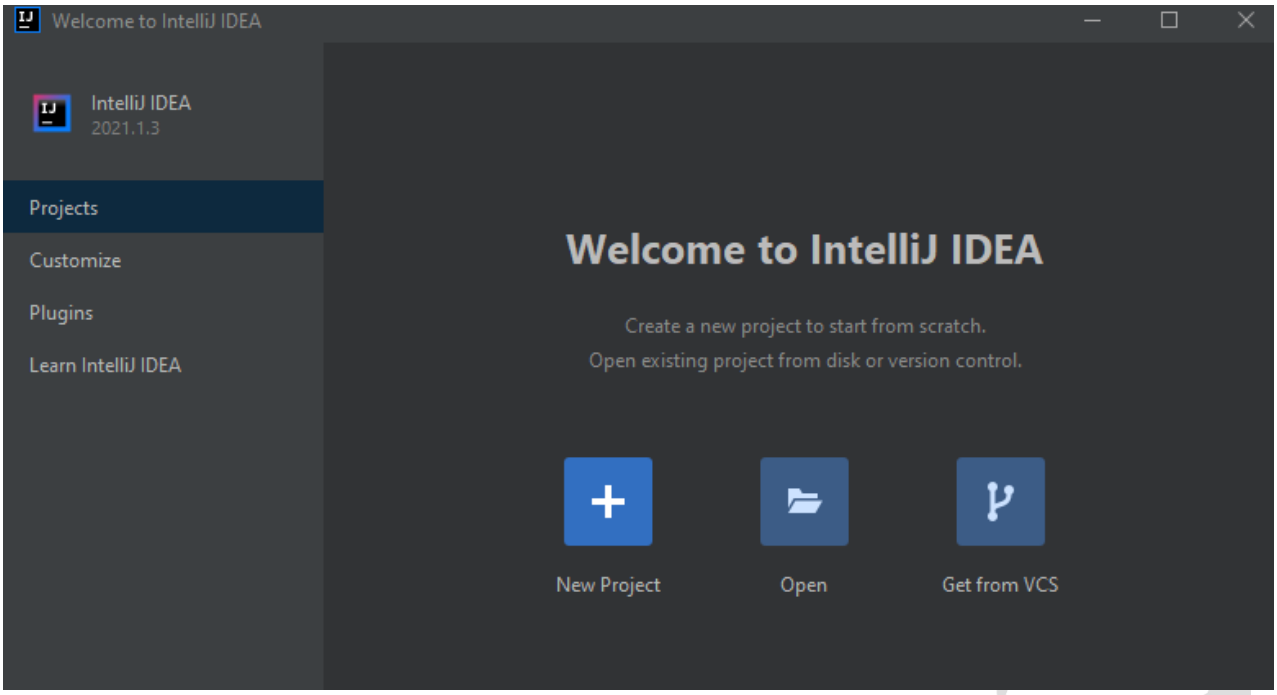
InteliJ IDEA es un entorno de desarrollo como Android Studio, Visual Studio Code, Atom, etc., es el precursor del mismo Android Studio y se usa para programar en Kotlin cuando no se quiere desarrollar una app de Android, como por ejemplo para una aplicación de consola.



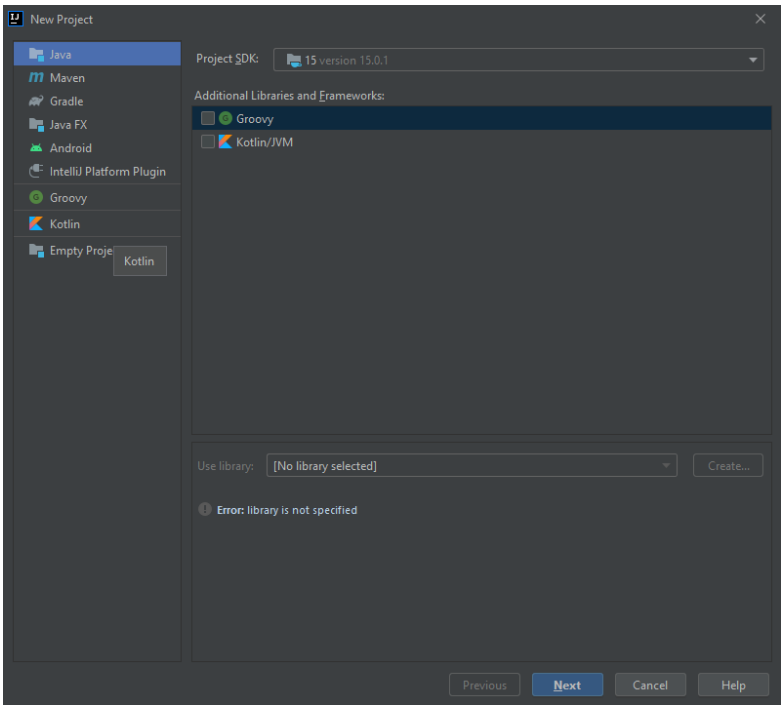
https://www.jetbrains.com/es-es/idea/download/#section=windows

Nuevo Proyecto en IntelliJ IDEA usando Kotlin

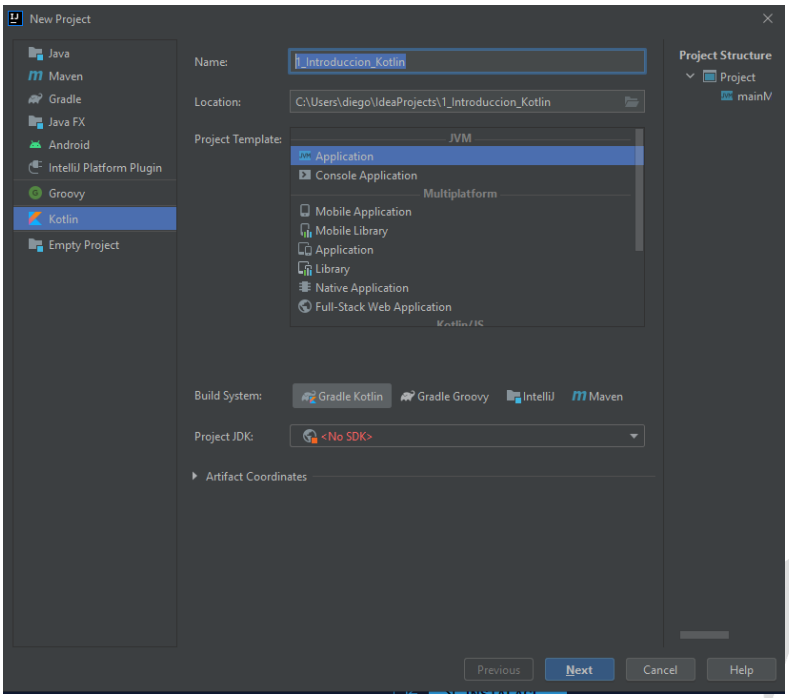
Para crear un nuevo proyecto vamos a dar clic en el botón de New Project.



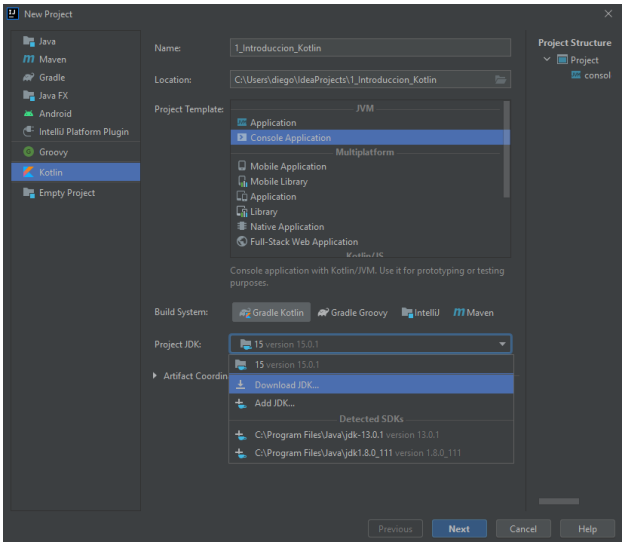
Luego vamos a seleccionar la opción de Kotlin que se encuentra a la izquierda para poder crear un proyecto con el lenguaje de programación Kotlin.

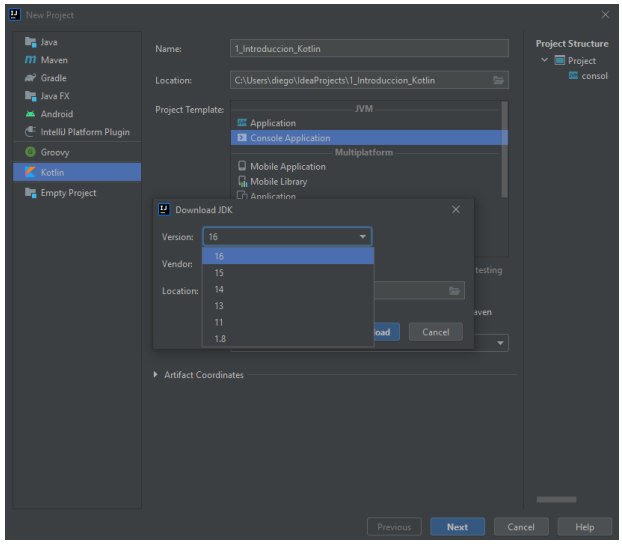


Después le daremos un nombre al proyecto, seleccionaremos Console Application para que el ejemplo sea de un código cuyo resultado solo pueda ser visto en la consola del mismo IDE (entorno de desarrollo), esto no sirve de mucho, pero se hará para conocer las distintas partes del lenguaje de programación Kotlin, para que después este conocimiento se pueda aplicar en la creación de una app en Android Studio (que es un IDE distinto).

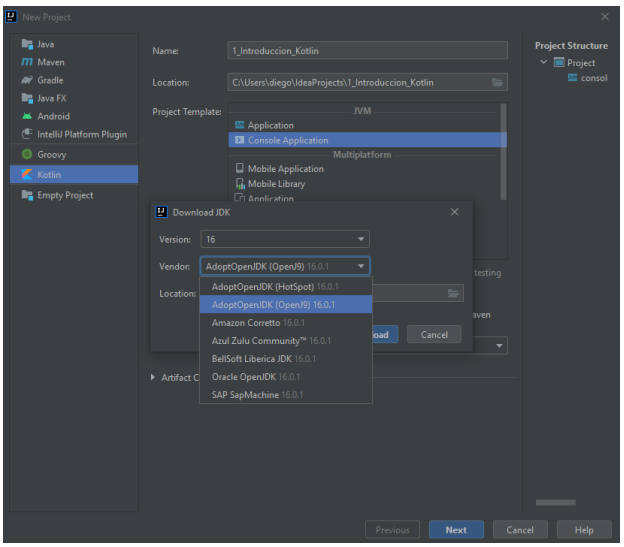


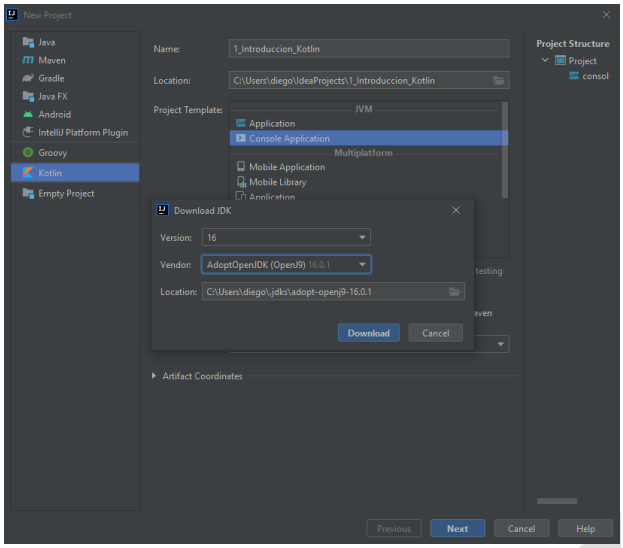
Ahora lo que se hará es seleccionar el tipo de JDK (Java Development Kit), que podrá ser el que tengamos descargado si es que alguna vez hemos hecho una aplicación en Java o se podrá descargar aquí mismo si es que se quiere usar alguna versión alterna de JDK, aunque es recomendable usar la misma que ya esté instalada.



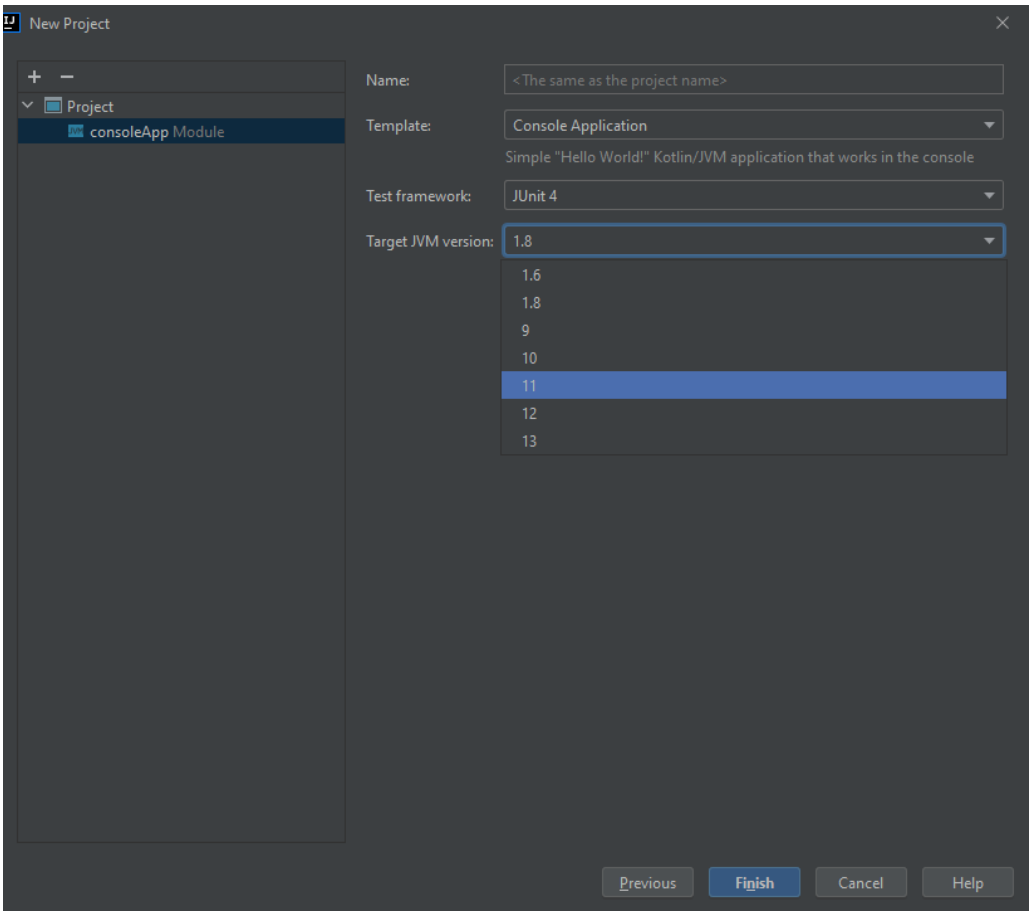


Y como Vendor se puede elegir el que sea, aunque en este caso elegimos el AdoptOpenJDK (OpenJ9) y esa versión se instalará en el directorio indicado hasta abajo.

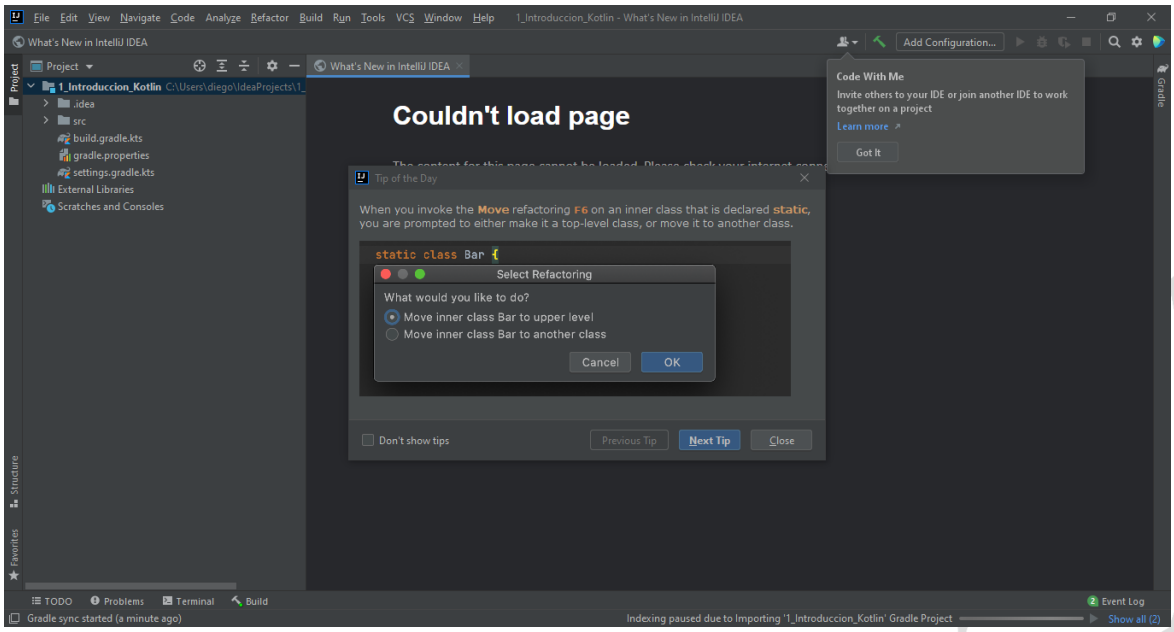




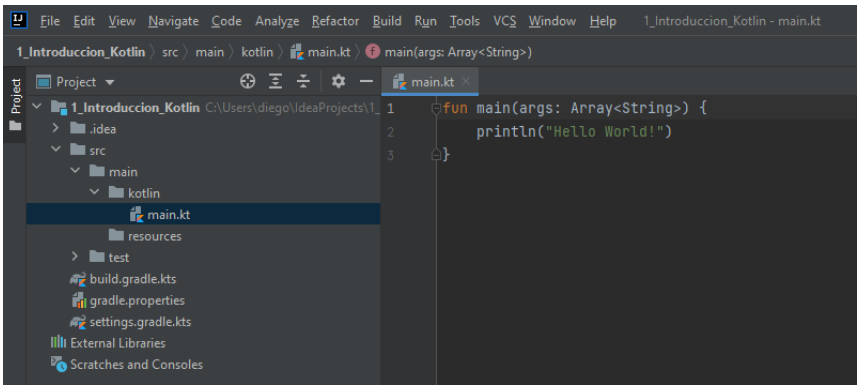
Posteriormente daremos clic en el botón de Next e indicaremos que el nombre del proyecto sea el mismo, que es una aplicación de consola, la versión de JDK y finalmente podremos crear el proyecto.



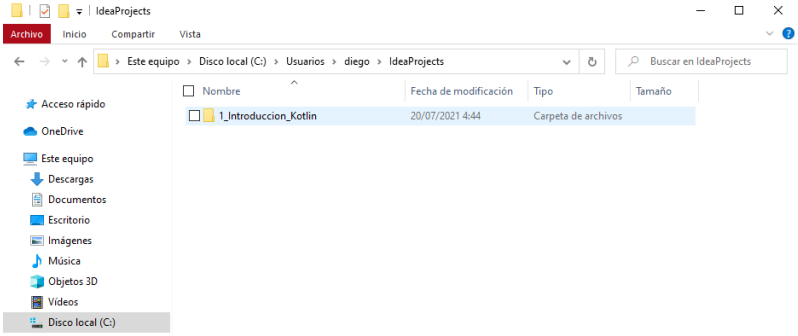
Cuando hayamos creado el proyecto, el IDE creará todas las carpetas del código con todo y su graddle, que es usado para la inyección de dependencias y poder compilar el código para que se muestre en consola, pero todo esto no es de gran utilidad explicarlo ya que el código solo servirá para ejemplificar el uso de variables, clases, etc. en el lenguaje de Kotlin.



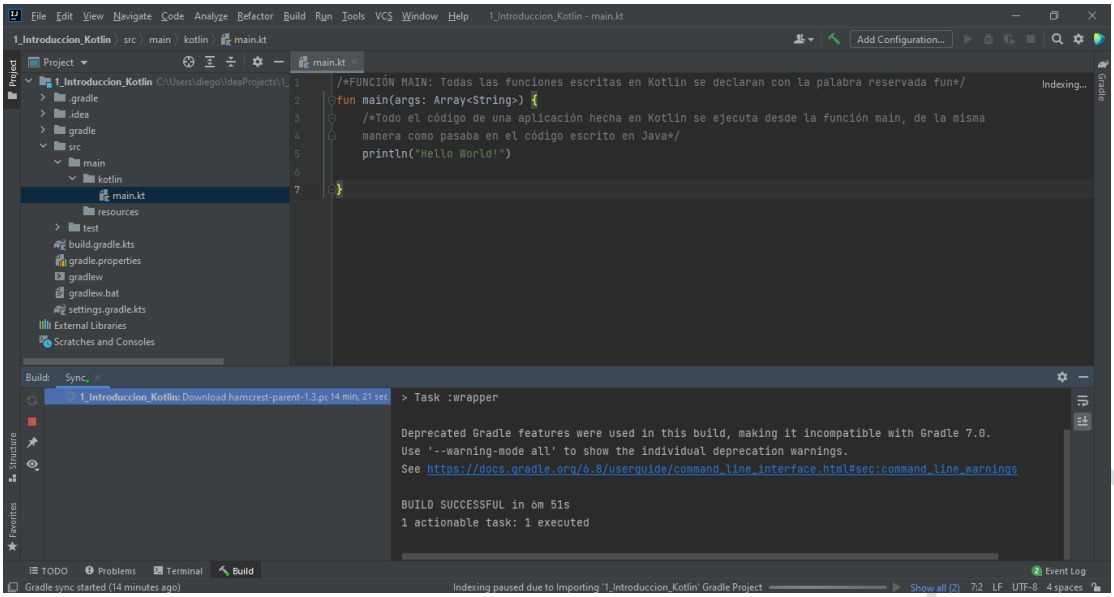
La única carpeta que nos interesa es la de 1_Introduccion_Kotlin/src/main/kotlin/main.kt porque el archivo main es el que correrá todo el código hecho en Kotlin para mostrarlo en consola.

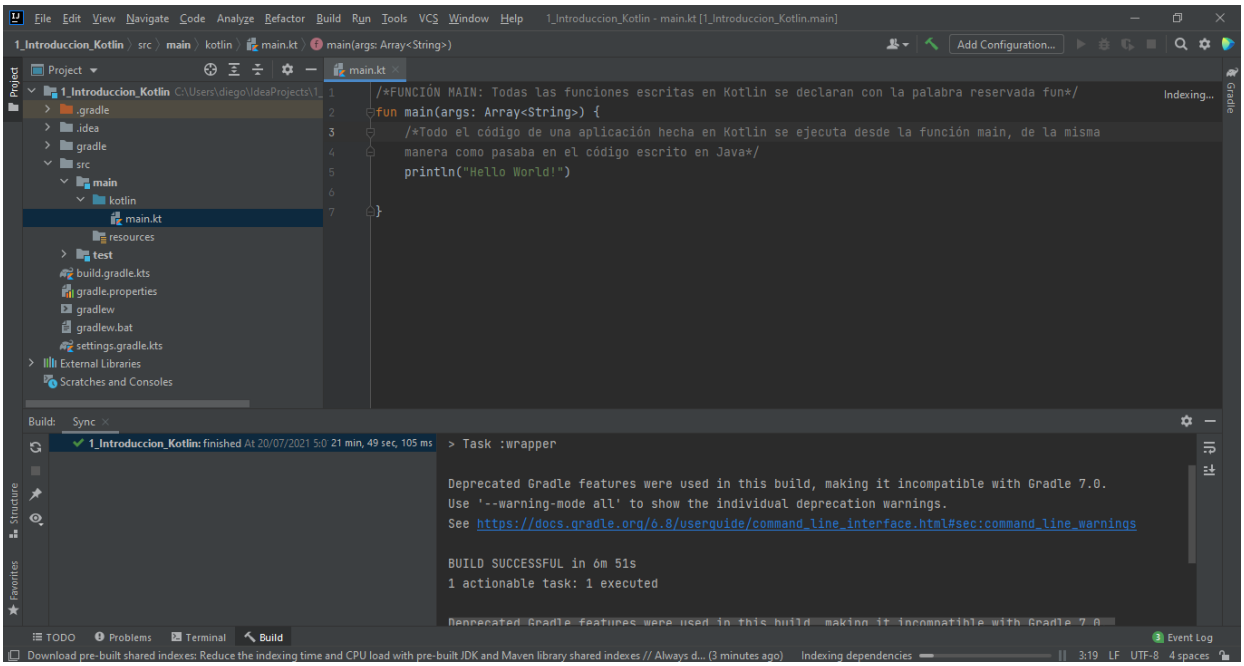


Y la ubicación de las carpetas creadas para el proyecto dentro de la computadora es la misma en donde se encuentra la carpeta para la creación de proyectos hechos con Android Studio, que es la de C:\Users\diego\IdeaProjects.



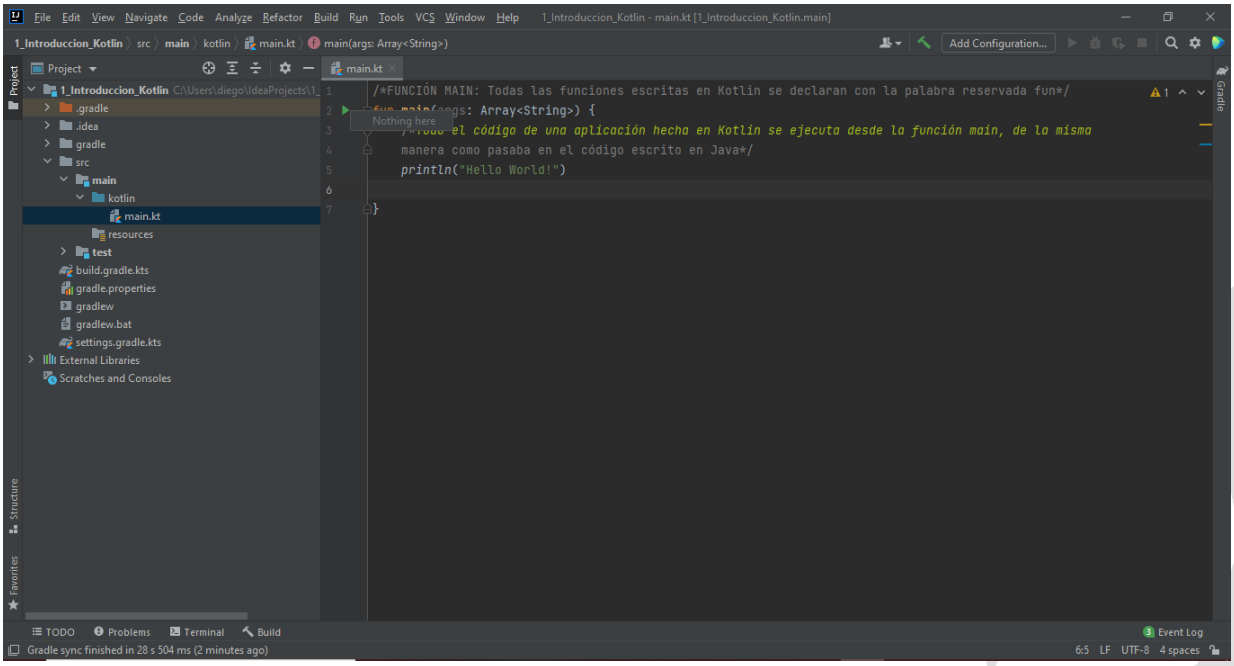
Para correr el código debemos esperar un buen rato a que se terminen de instalar las dependencias del proyecto, que se habrán terminado de instalar cuando en la parte de abajo donde se encuentra un martillo con un punto verde, se muestre todo gris.

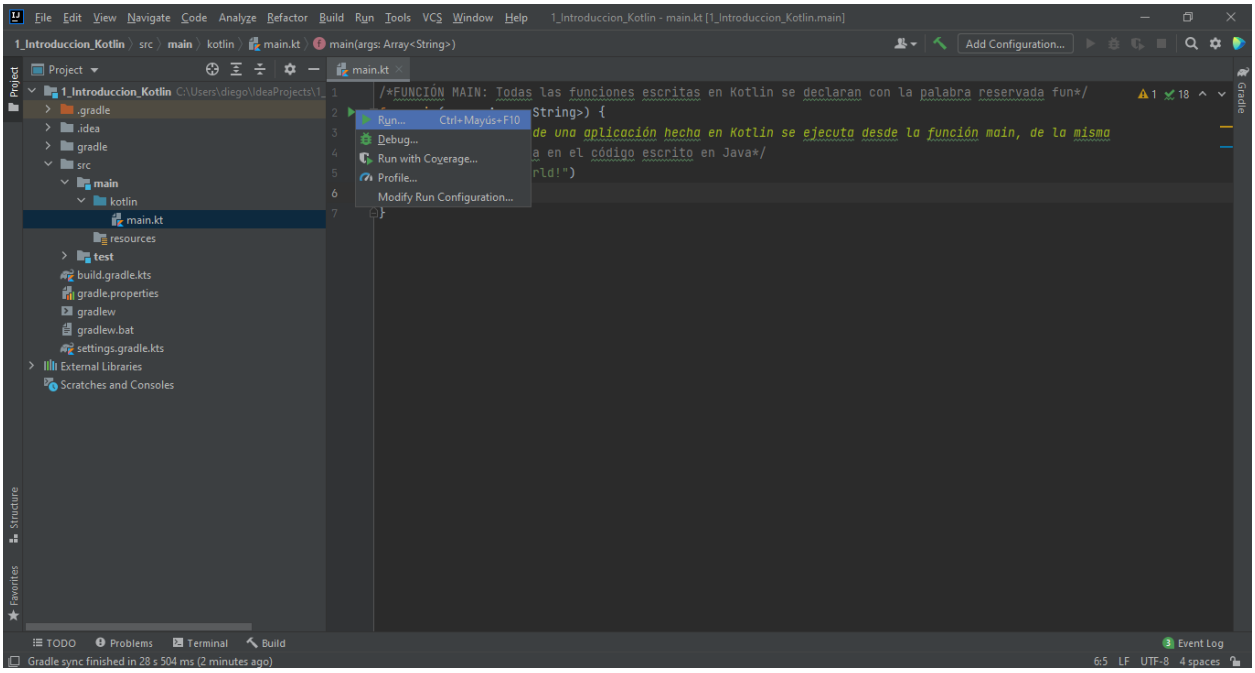




Cuando la barra inferior que dice Indexing dependencies termine de cargar, aparecerá un botón verde de play en la esquina superior izquierda del código.

Cuando se hayan terminado de instalar las dependencias debemos dar clic en el botón verde de Run.




File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
1_Introduccion_Kotlin - main.kt [1_Introduccion_Kotlin.main]
Add Configuration...
Project
1_Introduccion_Kotlin
.gradle
.idea
gradle
src
main
kotlin
main.kt
resources
test
build.gradle.kts
gradle.properties
gradlew
gradlew.bat
settings.gradle.kts
External Libraries
Scratches and Consoles
main.kt
/*FUNCIÓN MAIN: Todas las funciones escritas en Kotlin se declaran con la palabra reservada fun*/
String>) {
de una aplicación hecha en Kotlin se ejecuta desde la función main, de la misma
a en el código escrito en Java*/
rld!")
}
Run... Ctrl+Mayús+F10
Debug...
Run with Coverage...
Profile...
Modify Run Configuration...
Structure
Favorites
Gradle
TODO
Problems
Terminal
Build
Event Log
Gradle sync finished in 28 s 504 ms (2 minutes ago)
6:5 LF UTF-8 4 spaces

Lenguaje de programación Kotlin

En Kotlin todos los tipos primitivos son considerados como objetos, esto para que con ellos se pueda aplicar métodos para realizar operaciones, las operaciones más utilizadas por medio de estos métodos aplicados a los tipos primitivos son las siguientes y su sintaxis se puede observar en la documentación descrita en el siguiente link, considerando que los objetos creados con las variables de los tipos primitivos son llamados kotlin dentro de la documentación, por lo que su sintaxis es **kotlin.nombre_método**:

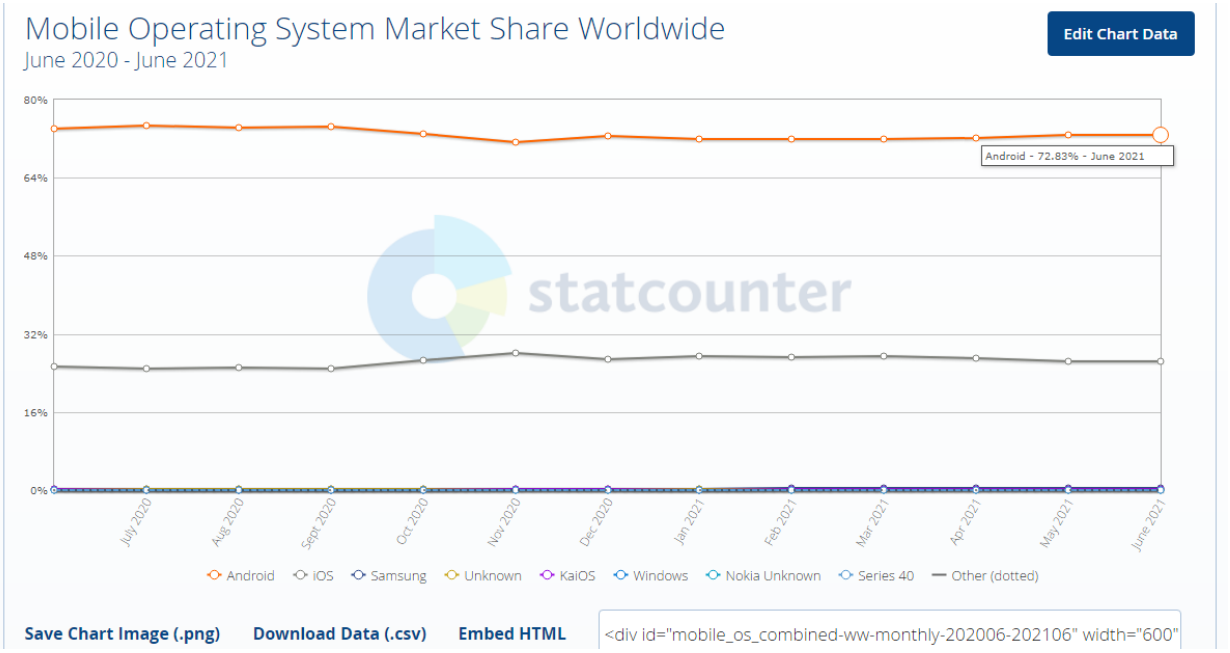
https://kotlinlang.org/api/latest/jvm/stdlib/

A continuación, en la columna que dice Operator Fun se muestra la operación realizada por medio del método aplicado al objeto Kotlin, mientras que en las primeras 2 se describe su equivalente, pero hecho con simbología más simple.

Operaciones más utilizadas

Expresión	Función	Operator Fun
a + b	c = a + b	public operator fun plus(other: Int): Int
a - b	c = a - b	public operator fun minus(other: Int): Int
a * b	c = a * b	public operator fun times(other: Int): Int
a / b	a = a / b	public operator fun div(other: Int): Int
a % b	c = a % b	public operator fun rem(other: Int): Int
a++	c = a++	public operator fun inc(): Int
a–	c = a–	public operator fun dec(): Int
a > b	c = a > b	public override operator fun compareTo(other: Int): Int
a < b	c = a < b	public override operator fun compareTo(other: Int): Int
a >= b	c = a >= b	public override operator fun compareTo(other: Int): Int
a <= b	c = a <= b	public override operator fun compareTo(other: Int): Int
a != b	c = a != b	public open operator fun equals(other: Any?): Boolean

Nota: Es preferible elegir utilizar el desarrollo en Android Studio ya que por estadísticas realizadas por statcounter, el %72.83 del mercado global opta por usar el sistema operativo Android.



https://gs.statcounter.com/os-market-share/mobile/worldwide

Kotlin como Programación funcional

Los paradigmas de programación nos describen la forma en la que se escribe un código, eligiendo una de las dos formas principales, que son el paradigma imperativo y el funcional.

- **Paradigma imperativo:** Se basa en modificar el valor de las variables en un código y se centra más en describir cómo funciona un programa.
- **Paradigma funcional:** Se centra más en qué tiene que hacer un programa y no cómo lo hace.
    - **Nunca mutable, siempre inmutable:** Un elemento es mutable cuando puede cambiar su valor, por lo que Kotlin es mejor dirigirlo a ser inmutable, por lo que utilizando sus variables de lectura que no pueden cambiar su valor una vez declarado, nos aseguramos de que sea inmutable el código.
    - **Las funciones pueden funcionar como objetos:** Las funciones pueden almacenarse en variables, pasarse como parámetros y tratarse como cualquier objeto.
    - **Usa funciones puras:** Esto implica que las funciones declaradas solo dependen de las variables que reciban como parámetro y no otras, evitando así que puedan ser afectadas por su entorno exterior.

Código de Ejemplo Kotlin y sus Características

```
/*TEMAS: VARIABLES DE LECTURA Y ESCRITURA, CONSTANTES, IMPRIMIR EN CONSOLA,
CONCATENACIÓN, TIPOS DE DATOS PRIMITIVOS, OPERACIONES MATEMÁTICAS, ARRAYS Y
LISTAS, CONDICIONALES (IF, ELSE IF y WHEN) Y BUCLES (WHILE, DO WHILE, FOR Y
FOR EACH), FUNCIONES NORMALES, ANÓNIMAS, DE EXTENSIÓN Y DE ORDEN SUPERIORE,
FUNCIONES ANÓNIMAS .MAP Y .FILTER, EXCEPCIONES, TRY CATCH, ESTRUCTURA DE
DATOS: MAPS, SET, MAPS Y SCOPE FUNCTIONS*/
```

Constantes y Variables

```
/*CONSTANTES: Estas deben ir definidas fuera de la función y antes de ella, se declara con la palabra
reservada const val y su valor no debe cambiar nunca, su sintaxis es la siguiente:
const val nombre_constante: tipo_de_dato = valor*/
const val PI = 3.1416

/*FUNCIÓN MAIN: Todas las funciones escritas en Kotlin se declaran con la palabra reservada fun*/
fun main(args: Array<String>) {
    /*El código de una aplicación hecha en Kotlin se ejecuta desde la función main, de la misma manera como
    pasa en el código escrito en Java*/

    /*IMPRIMIR EN CONSOLA: Se hace a través del método println() y dentro de éste se puede imprimir
    el contendio de una variable o un String declarado con las comillas de siempre*/
    println("Hello World!")
    /*Nota: Presionando ALT+Enter con el mouse sobre una línea de código podemos visualizar los consejos
    del entorno de desarrollo Intellij IDEA*/




    //VARIABLES:
    //VARIABLES DE LECTURA Y ESCRITURA
    /*var: Es la palabra reservada usada para declarar variables, que son datos almacenados en la memoria
    RAM y que podrán cambiar su valor a lo largo del código, su sintaxis es la siguiente:
    var nombre_variable: tipo_de_dato = valor*/
    var dinero: Int = 10
    println(dinero)
    dinero = 5
    println(dinero)

    //VARIABLES DE SOLO LECTURA
    /*val: La palabra reservada val nos permite crear variables de solo lectura, en estas solo podrán ser
    asignado su valor una vez y luego de eso el valor no podrá cambiar, su sintaxis es la siguiente:
    val nombre_variable: tipo_de_dato = valor*/
    val nombre = "María"
    /*Si dejo esta línea de código, el programa me dará un error
    nombre = "Pedro"*/
    println(nombre)

    /*CONSTANTES: Estas deben ir definidas fuera de la función y antes de ella, se declara con la palabra
    reservada const, en este caso la constante se declaró fuera de la función main*/
    println(PI)
```

Tipos de Datos

```
    /*TIPOS DE DATOS: En Kotlin cuando se declara una variable o constante, dependiendo del valor que tenga
    esta, puede ser que poner explícitamente el tipo de dato sea redundante, esto lo indica IntelliJ IDEA
    poniendo el nombre de la variable o constante en color gris y al dar clic en ALT+Enter sobre el nombre de
    la variable nos mostrará que podemos quitar el tipo de dato sin problema y el programa seguirá funcionando.
    Los distintos tipos de datos que existen en Kotlin para declarar variables son considerados como objetos y
    son los siguientes:
    - Boolean: Es un tipo de dato que puede valer true o false. Aunque en Kotlin existe un tercer valor que
    puede ser null, que significa nada, esto puede pasar cuando a alguna variable no se le ha asignado ningún
    valor, por lo que tiene valor null.
    - Int: Es un tipo de dato numérico entero que abarca desde el número -2,147,483,648 hasta el número
    2,147,483,647.
    - Long: Es un tipo de dato numérico entero que soporta muchísimas cifras, se pone su valor seguido de
    la letra L, si ponemos la letra L no es necesario indicar que es de tipo Long y abarca desde el número
    -9,223,372,036,854,775,808 hasta el número 9,223,372,036,854,775,807.
    - Float: Es un tipo de dato numérico decimal que permite 6 a 7 cifras decimales en el número, se pone su
    valor seguido de la letra f.
```

```
- Double: Es un tipo de dato numérico decimal que permite mostrar 15 cifras decimales en el número.
- String: Es una cadena de caracteres y su valor se declara entre comillas dobles "", en Kotlin los
objetos String no se pueden declarar entre comillas simples*/
val boolean = true
val numeroLargo = 3L
val double = 2.71972
val float = 1.1f

/*TIPO DE DATO NULL: Es un tipo de dato que aparta un lugar en la RAM para que almacene un valor de
cierto tipo, pero considerando que esa variable pueda o no tener un valor durante la ejecución del código,
por lo que considera el valor null, que significa nada y se relaciona con los valores booleanos.
- Boolean: Es un tipo de dato que puede valer true o false. Aunque en Kotlin existe un tercer valor que
puede ser null, que significa nada, esto puede pasar cuando a alguna variable no se le ha asignado ningún
valor, por lo que tiene valor null. Más abajo en el código se realiza un ejemplo con este tipo de dato.*/
```

Operaciones Matemáticas

```
/*OPERACIONES MATEMÁTICAS: Ya que hayamos declarado variables o constantes, con ellas podremos aplicar
operaciones matemáticas entre ellas, esto lo podemos hacer porque en el lenguaje de programación Kotlin
todas las variables o constantes se consideran como un objeto kotlin, los distintos métodos matemáticos
que se pueden aplicar con los objetos kotlin se pueden observar en la documentación de kotlin, donde dice
kotlin.math, que se encuentra en el siguiente link
https://kotlinlang.org/api/latest/jvm/stdlib*/
val primerValor = 20
val segundoValor = 10
/*Métodos matemáticos:
- kotlin.minus(): Sirve para restar el valor del objeto Kotlin al que se le está aplicando el método menos
el valor que se le pasa como parámetro al método minus, aunque esto se puede realizar también con el signo -*/
val tercerValor1 = primerValor.minus(segundoValor)
val tercerValor2 = primerValor - segundoValor
/*Nota: Si presiono CTRL+B sobre algún método matemático se me mostrará el archivo Primitives.kt donde se
muestran todos los métodos de este tipo*/
```

Concatenación

```
/*CONCATENACIÓN: Sirve para sumar dos cadenas de texto o una cadena de texto y un número cualquiera, aunque
al hacer esto lo que ocurrirá es que el número se convertirá a una cadena de texto, por lo que ya no se
podrán hacer operaciones matemáticas con él, esto se puede hacer de la forma normal como se hace con otros
lenguajes de programación o por medio de la siguiente sintaxis:
variable_o_constante = "texto normal $nombre_variable_a_contatenar"
variable_o_constante = "texto normal ${nombre_variable_a_contatenar.método_aplicado_a_variable}"*/
val nombrecito = "Pedro"
val frase = nombrecito + " " + tercerValor2
println(frase)
val frase2 = "la primera frase fue $frase"
println(frase2)
```

Tipos de Datos Agrupados: Listas y Arrays

```
/*LISTAS: Son aquellos grupos de elementos guardados en una misma variable que se crean por medio del
método listOf(), existen dos tipos de listas, las listas mutables y las listas inmutables, en ellas se
puede declarar explícitamente el tipo primitivo de los elementos que almacenan o no.*/

/*- Listas inmutables: Son aquellas a las que no se les puede modificar ni borrar el valor de sus elementos,
pero si se puede obtener el valor de ellos, se crean por medio del método listOf() y la palabra reservada
val, siguiendo la sintaxis mostrada a continuación:
val nombre_lista_inmutable = listOf<Tipo_de_dato_primitivo>("Elemento1", "Elemento2", ..., "Elemento_n")*/
val listaInmutable = listOf<String>("Juan", "Enrique", "Camila")
/*Imprimir el valor de la lista hará que se muestren entre corchetes en consola de la siguiente manera:
[Juan, Enrique, Camila]
Esta lista como es inmutable no podrá ser modificada por ningún método aplicado a los objetos Lista.*/
println(listaInmutable)
```

```
    /*- Listas mutables: Son aquellas listas a las que se les puede eliminar o modificar sus valores, se crean
    por medio del método mutableListOf() y la palabra reservada val, siguiendo la sintaxis mostrada a
    continuación:
    val nombre_lista_mutable = mutableListOf<Tipo_de_dato_primitivo>("Elemento1", "Elemento2", ...,
"Elemento_n")*/
    /*Cuando se crea una lista vacía si se debe indicar el tipo de dato que espera recibir.*/
    val listaVaciaMutable = mutableListOf<String>()
    /*Imprimir el valor de una lista vacía hará que se muestre entre corchetes en consola de la siguiente manera:
    []
    Esta lista como es mutable podrá ser modificada por ningún método aplicado a los objetos Lista.*/
    println(listaVaciaMutable)
    /*Los métodos para modificar listas mutables son los siguientes:
    AGREGAR ELEMENTOS A UNA LISTA
    - listaMutable.add: Sirve para añadir elementos a una lista mutable.

    ELIMINAR ELEMENTOS DE UNA LISTA
    - listaMutable.removeAt(): Sirve para eliminar el elemento de una lista mutable, indicado por el índice
    puesto en su paréntesis.
    - listaMutable.removeIf{}: Sirve para eliminar los elementos de la lista siempre y cuando estos cumplan la
    condición indicada en las llaves del condicional if, para ello se crea una variable intermedia que
    representará todos y cada uno de los elementos incluidos en la lista y a la cual se le aplicará un método
    para analizar la condición de cada uno de ellos para ver si los elimina o no.

    OBTENER VALORES DE UNA LISTA
    - listaMutable o listaInmutable.get: Sirve para obtener elementos de una lista mutable o inmutable y
    guardarlos en una variable o constante, indicando el índice del elemento que se quiere extraer.
    - listaMutable o listaInmutable[]: Se utiliza en vez del método .get para obtener elementos de una lista
    mutable o inmutable y guardarlos en una variable o constante, indicando el índice del elemento que se
    quiere extraer.
    - listaMutable o listaInmutable.first(): Se utiliza en vez del método .get o el operador [] para obtener
    solo el primer elemento de una lista mutable o inmutable para guardarlo en una variable o constante.
    - listaMutable o listaInmutable.firstOrNull(): Obtiene solo el primer elemento de una lista mutable o
    inmutable como lo hace la función anónima first() para guardarlo en una variable o constante, pero
    considera la excepción de que ese valor sea null, si es null, en consola se mostrará la palabra null pero
    no ocurrirá un error en el programa.

    ORDENAR LOS ELEMENTOS DE UNA LISTA
    - listaMutable o listaInmutable.sorted(): Ordena los elementos numéricos de una lista de menor a mayor.
    - listaMutable o listaInmutable.sortedDescending(): Ordena los elementos numéricos de una lista de mayor
    a menor.
    - listaMutable o listaInmutable.sortedBy{}: Ordena los elementos numéricos de una lista dependiendo de una
    condición indicada en las llaves del condicional if, para ello se crea una variable intermedia que
    representará todos y cada uno de los elementos incluidos en la lista y a la cual se le aplicará un método
    matemático o lógico para analizar el valor numérico de cada uno de ellos, los elementos numéricos que
    cumplan la condición se acomodarán al inicio de la lista yendo de menor a mayor y los que no la cumplan se
    acomodarán al final de la lista yendo de menor a mayor también.
    - listaMutable o listaInmutable.shuffled(): Ordena los elementos de una lista en forma aleatoria, cada
    vez que el programa sea ejecutado, los elementos de la lista serán ordenados de una manera distinta.
    - listaMutable o listaInmutable.reversed(): Ordena los elementos de una lista en forma inversa a como
    se encuentra actualmente.*/
    //AGREGAR ELEMENTOS A UNA LISTA
    listaVaciaMutable.add("Pedrito")
    //OBTENER ELEMENTOS DE UNA LISTA
    val valorDeListaUsandoGet = listaVaciaMutable.get(0)
    println(valorDeListaUsandoGet)
    val valorDeListaUsandoOperador = listaVaciaMutable[0]
    println(valorDeListaUsandoOperador)
    //OBTENER EL PRIMER ELEMENTO DE UNA LISTA
    val primerElementoListaInmutable: String = listaInmutable.first()
    println(primerElementoListaInmutable)
    listaVaciaMutable.removeAt(0)
    //OBTENER EL PRIMER ELEMENTO DE UNA LISTA NULEABLE
    val primerElementoListaMutable: String? = listaVaciaMutable.firstOrNull()
    println(primerElementoListaMutable)
    listaVaciaMutable.add("Pedrito Vuelve, la vengaza de pedrito")
    println(listaVaciaMutable)
    //ELIMINAR ELEMENTOS DE UNA LISTA QUE CUMPLAN UNA CONDICIÓN
    /*Solo se eliminan los elementos de la listaVaciaMutable que tengan más de 3 caracteres*/
    listaVaciaMutable.removeIf { caracteres -> caracteres.length > 3 }
    println(listaVaciaMutable)
    //ORDENAR ELEMENTOS NUMÉRICOS DE UNA LISTA
    val numerosDeLoteria = listOf(66, 56, 22, 43, 11, 78)
    val menorAMayor = numerosDeLoteria.sorted()
    println(menorAMayor)
    val mayorAMenor = numerosDeLoteria.sortedDescending()
    println(mayorAMenor)
    /*ORDENAR ELEMENTOS DEPENDIENDO DE UNA CONDICIÓN, los que la cumplan se ponen al inicio de la lista y los
    que no se ponen al final, ambos yendo del número menor al mayor, esto normalmente se usa para
    comparaciones.*/
    val ordenarPorMultiplos = numerosDeLoteria.sortedBy { variable_intermedia -> variable_intermedia < 50 }
    println(ordenarPorMultiplos)
```

```
    //ORDENAR ELEMENTOS DE UNA FORMA ALEATORIA
    val ordenAleatorio = numerosDeLoteria.shuffled()
    println(ordenAleatorio)
    //ORDENAR LOS ELEMENTOS DE UNA LISTA DE FORMA INVERSA A COMO SE ENCUENTRAN
    val ordenInverso = numerosDeLoteria.reversed()
    println(ordenInverso)

    /*ARRAYS: Son aquellos grupos de elementos almacenados en una misma variable, pero que NO se crean por
    medio del método listOf() sino por medio del método arrayOf(), este tipo de dato en Kotlin no es tan
    poderoso porque no cuenta con tantos métodos como las listas*/
    val miArray = arrayOf(1, 2, 3, 4, 5)
    println("Nuestro array es $miArray")
    /*LISTAS VS. ARRAYS: Los array no se pueden imprimir en consola tan fácilmente como las listas y no cuentan
    con tantos métodos de manejo de datos debido a que retornan sus datos en tipo Bytecode, por lo que existe
    un método llamado toList() que permite convertir un array en una lista:
    CONVERTIR UN ARRAY EN UNA LISTA
    - array.toList(): Es un método que sirve para convertir un array en una lista*/
    println("Nuestro array es ${miArray.toList()}")
```

Estructuras de Control: Condicionales y Bucles

```
    /*ESTRUCTURAS DE CONTROL: Estas se componen de condicionales y bucles, que afectan el flujo de ejecución
    del código*/
    /*Los condicionales if o else if pueden ser declarados usando las llaves de apertura y cierre o no*/
    val nombre_chango = "El chango marango"

    //CONDICIONALES IF Y WHEN, SUS DISTINTAS REPRESENTACIONES EN KOTLIN
    //IF CON LLAVES DE APERTURA Y CIERRE
    /*Método kotlin.isNotEmpty(): Este método ayuda a determinar si la variable kotlin esta vacía o no, osea
    si tiene algún valor o no, si el resultado es true es porque la variable no esta vacía y si es false es
    porque la variable no tiene valor*/
    if(nombre_chango.isNotEmpty()){
        /*Si el resultado del paréntesis del if es true se ejecuta lo siguiente*/
        /*Método kotlin.length(): Este método se aplica solo a objetos String y retorna un número que indica el
        número de letras de la cadena de caracteres*/
        println("El largo de la variable $nombre_chango es de ${nombre_chango.length}")
    }else{
        /*Si el resultado del paréntesis del if es false se ejecuta lo siguiente*/
        println("Error, la variable $nombre_chango está vacía")
    }

    //IF SIN LLAVES DE APERTURA Y CIERRE
    var mensaje: String
    if (nombre_chango.length > 4) mensaje = "Tu nombre es largo!" else mensaje = "Tienes un nombre corto"
    println(mensaje)

    //IF QUE SOLO AFECTA UNA VARIABLE EN SU EJECUCIÓN Y SU VARIABLE SE DECLARA APARTE
    var el_tigre: String
    el_tigre = if(nombre_chango != "El tigre"){
        "Tu nombre no es el Tigre"
    }else{
        "Tu nombre es el Tigre"
    }
    println(el_tigre)

    /*IF QUE SOLO AFECTA UNA VARIABLE EN SU EJECUCIÓN Y SU VARIABLE SE DECLARA APLICANDO LA INMUTABILIDAD:
    Inmutabilidad significa que un valor pueda ser asignado a una variable solo una vez, esto para evitar que
    el valor de una variable pueda ser reescrito indeseablemente*/
    val el_chango_marango: String = if(nombre_chango != "El chango marango"){
        "Tu nombre no es El chango marango"
    }else{
        "Tu nombre es El chango marango"
    }
    println(el_chango_marango)

    /*CONDICIONAL ELSE IF: Este sirve para comparar varias condiciones ligadas entre sí, se puede escribir
    en la misma línea de código*/
    val mensaje2: String = if (nombre_chango.isEmpty()) "No tienes nombre!" else if (nombre_chango.length > 2) "Tienes
un nombre medio largo :S" else "Tienes un nombre chistoto XD"
    println(mensaje2)

    /*CONDICIONAL WHEN: El condicional when sirve para cuando tenemos varias condiciones relacionadas finitas*/
    val nombreColor = "amazul"
    when (nombreColor){
        /*El contenido del when dependiendo de que resultado sea el proveniente de su condición puede ser
```

```
    descrito en una línea o en varias contenido dentro de llaves, además para que una misma cosa pase
    con distintos resultados estos se ponen uno tras otro separado por una coma de la siguiente manera:
    condición -> resultado_de_una_línea
    condición -> {
        resultado_de_varias_líneas
        resultado_de_varias_líneas
        resultado_de_varias_líneas
    }*/
    "Amarillo", "amarillo" -> {
        println("El amarillo es el color mas brillante")
        println("Y de los tigres daltónicos")
    }
    "Rojo" -> println("El rojo es el color mas vergas")
    "verde", "Amazul", "amazul" -> println("El verde o coloquialmente llamado Amazul es el color de la vida")
    else -> println("Error, no tengo información del color feo que pusiste o lo escribiste mal por wey")
}

/*WHEN CON RANGOS NUMÉRICOS: El condicional when se puede utilizar para dar un resultado dependiendo de que
lo que haya en su paréntesis se encuentre en un rango numérico descrito después de la palabra reservada in,
el número inicial, dos puntos y el número final de la siguiente manera:
in número_inicial .. número_final -> resultado*/
val code = 400
when(code){
    in 200..299 -> println("Todo ha ido bien")
    in 400..500 -> println("Código fuera de rango")
    else -> println("Código desconocido, algo ha fallado")
}

//WHEN QUE SOLO AFECTA UNA VARIABLE EN SU EJECUCIÓN Y SU VARIABLE SE DECLARA APARTE
val tallaDeZapatos = 40
val mensaje3 = when(tallaDeZapatos){
    41, 43 -> "Si hay ;D"
    else -> "No hay :( sorry, not sorry"
}
println(mensaje3)

//BUCLES WHILE, DO WHILE, FOR Y FOR EACH, SUS DISTINTAS REPRESENTACIONES EN KOTLIN
/*BUCLE WHILE: Este bucle se ejecuta hasta que la condición deje de ser sea verdadera, si nunca es verdadero
el resultado de la condición, nunca se ejecuta, no sabemos exactamente cuantas veces se ejecutará*/
var contador = 10
while (contador >= 0){
    println("El valor del contador es $contador")
    contador--
}

//BUCLE DO WHILE: Este bucle se ejecuta una vez aún cuando la condición no sea verdadera
/*En este caso se aplica el bucle do while imprimiendo números aleatorios hasta que este sea menor a 50,
pero aunque a la primera el número aleatorio sea menor a 50, se imprimirá mínimo una vez antes de que sea
menor a 50, por lo que se verá el número que sea menor a 50*/
do {
    println("Generando número aleatorio...")
    /*Los rangos numéricos en Kotlin se describen poniendo un número inicial, dos puntos y el número final,
    a este rango se le debe aplicar el método kotlin.random() para que se cree un número aleatorio en el
    rango al que se le aplicó el método*/
    val numeroAleatorio = (0..100).random()
    println("El número generado es $numeroAleatorio")
}while (numeroAleatorio > 50)

/*BUCLE FOR: En este bucle se sabe exactamente cuantas veces se va a ejecutar, para ello se crea una
variable local que solo se aplique para el for y luego se indica a qué variable tipo array o tipo Lista
creada con el método listOf() se va a aplicar*/
val listaDeFrutas = listOf("Manzana", "Pera", "Frambuesa", "Durazno")
for (fruta in listaDeFrutas){
    println("Hoy voy a comerme una fruta llamada $fruta")
}

/*BUCLE FOR EN UNA MISMA LÍNEA: Para usar más de una línea de código en el for se puede usar el ;*/
val superheroes = listOf("Ironman", "Spiderman", "Thor", "Deadpool")
var numeroLista = 1
for (heroe in superheroes) println("$numeroLista.- $heroe"); numeroLista++

/*BUCLE FOR EACH: Este tipo de bucles solo se utiliza para leer una lista creada con el método ListOf o
un array, y su diferencia contra el bucle for es que este se aplica como método al objeto tipo lista o
array y se elige la segunda opción que muestra el asistente de IntelliJ o Android Studio, este método es
de tipo función anónima. Al usar la función anónima forEach también se debe crear una variable intermedia
que usaremos para que esta almacene y se sobreescriba con todos los valores del array. Su utilidad radica
en que al ser una función anónima, puede ser mezclada con otras funciones anónimas.

FUNCIÓN ANÓNIMA:
El bucle for each es una función anónima, una función anónima es un pedazo de código que se ejecutará n
cantidad de veces mientras que exista un elemento dentro de algún índice o posición del objeto lista o
```

```
    array, esta usa la sintaxis de flecha para crear y utilizar la variable intermedia de bucle forEach
    utilizando la siguiente sintaxis. Si sobre el nombre de la variable que almacena todos los elementos del
    objeto lista damos clic en ALT+ENTER -> Specify type explicitly, podremos ver de que tipo son los elementos
    que hay dentro del array.

    variable_tipo_lista.forEach {variable_intermedia -> uso de la variable_intermedia aplicado a todos los elementos
de la lista}*/
    val changos: List<String> = listOf("Pelon", "Marango", "Mandril")
    changos.forEach { tiposChangos -> println("Los tipos de changos son: $tiposChangos")}
```

Funciones Anónimas

```
    //FUNCIONES ANÓNIMAS MAP Y FILTER
    /*FUNCIÓN ANÓNIMA MAP: Esta es una de las funciones anónimas más utiles en Kotlin, ya que debido a la
    importancia de la inmutabilidad en el código (que el valor de las variables no pueda cambiar), para no
    tener que estar creando un bucle for y luego agregar elementos constantemente a una nueva lista, podemos
    hacer que una misma lista se adapte y convertirla a otro tipo de dato, para que por ejemplo una lista de
    Strings pueda convertirse en una de números enteros y contenga el largo de caracteres de cada uno de los
    objetos String que exista en la lista, esta función anónima se aplica al objeto tipo Lista que guarda los
    elementos originales que queremos convertir a otro tipo de dato en este nuevo objeto tipo lista, se
    indica el tipo de dato al que se quiere convertir el tipo de dato original creando una nueva variable
    intermedia y aplicándole un método que convierta ese tipo de dato en otro usando la siguiente sitaxis:

    val nombre_variable_con_tipo_de_dato_nuevo = variable_tipo_lista_original.map {variable_intermedia ->
variable_intermedia.método_para_cambiar_tipo_de_dato}.*/
    val caracteresDeChangos: List<Int> = changos.map { numeroLetrasTiposChangos -> numeroLetrasTiposChangos.length
}
    /*Si después de haber aplicado la función anónima .map damos clic en ALT+ENTER sobre el nombre de la nueva
    lista podemos hacer que se vea explícitamente el nuevo tipo de dato al que se convirtió la lista original
    en esta nueva lista creada con la opción de Specify type explicitly, al hacerlo aparecerá la lista con la
    siguiente sintaxis que indica que la variable es de tipo lista con valores primitivos enteros en este caso,
    comprobando así que los elementos se cambiaron de se tipo String a ser tipo Int.

    val nombre_variable_con_tipo_de_dato_nuevo: List<tipo_de_dato_nuevo> = variable_tipo_lista_original.map
{...*/
    println(caracteresDeChangos)

    /*FUNCIÓN ANÓNIMA FILTER: Esta función anónima sirve para filtrar los elementos que cumplan con una
    característica, en esta usualmente se utilizan los elementos creados con la función anónima .map para
    obtener las características que deseamos filtrar de los elementos de una lista original, utilizándola
    para aplicar la función anónima .filter*/
    val listaFiltrada = caracteresDeChangos.filter { changosFiltrados -> changosFiltrados > 5 }
    println(listaFiltrada)

    /*DATO ALMACENADO POR LA FUNCIÓN ANÓNIMA .MAP EN UNA VARIABLE CUALQUIERA: Así como en el primer ejemplo se
    convirtió con la función anónima kotlin.map un tipo de dato String a uno numérico, se puede hacer lo inverso
    y convertir un tipo de dato numérico a un String que esté relacionado a cada una de las posiciones numéricas
    de la lista original, pero es importante mencionar que dentro de la función .map se podrán ejecutar varias
    líneas de código, pero lo que se almacenará en la variable que use la función será el resultado de la última
    línea de código escrita, por lo que si por ejemplo se quiere imprimir en consola los valores de la lista
    original, se podrá hacer pero no deberá ser la última línea de código ya que el programa no podrá almacenar
    eso en la variable que utilice la función .map.*/
    //FUNCIÓN .MAP PARA CONVERTIR UNA LISTA NUMÉRICA EN UNA LISTA DE STRINGS
    val numeritos = listOf(98, 435, 34, 2156, 78, 1)
    val mensajesNumericos = numeritos.map { la_variable_intermedia ->
        println(la_variable_intermedia)
        //La última línea de código es la que se asigna a la variable mensajesNumericos
        "Tu numerito es de valor $la_variable_intermedia"
    }
    println(mensajesNumericos)
    /*COMBINACIÓN DE LAS FUNCIONES .MAP Y .FILTER: En este caso se utiliza para filtrar una lista y luego
    con los elementos filtrados crear una nueva lista de tipo String que se almacene en una variable.*/
    val numerosFiltrados = numeritos.filter { var_intermedia -> var_intermedia > 50 }.map { num -> "numero: $num"
}
    println(numerosFiltrados)
```

Excepciones

```
    //EXCEPCIONES: Se le llama excepción a cuando en un códgo ocurre un error.
    /*EXCEPCIÓN NULL POINTER: Es la excepción (o error que es lo mismo) causada cuando el programa espera
```

```
que una variable tenga un valor de cualquier tipo primitivo y resulta que este tiene valor null, por lo
que se crea un error.
Por ejemplo pensemos que queremos aplicar el método kotlin.length que solo se puede aplicar a objetos
tipo String, si este tiene un valor null, en consola se verá el error NullPointerException.*/

/*TIPO DE DATO NULL: Es un tipo de dato que aparta un lugar en la RAM para que almacene un valor de
cierto tipo, pero considerando que esa variable pueda o no tener un valor durante la ejecución del código,
por lo que considera el valor null, que significa nada y se relaciona con los valores booleanos.
- Boolean: Es un tipo de dato que puede valer true o false. Aunque en Kotlin existe un tercer valor que
puede ser null, que significa nada, esto puede pasar cuando a alguna variable no se le ha asignado ningún
valor, por lo que tiene valor null. Más abajo en el código se realiza un ejemplo con este tipo de dato.*/
/*NULL SAFETY: Kotlin nos ayuda a manejar el tipo de dato null, ya que nos da herramientas para que el
programa sepa que hacer cuando una variable tenga valor null, indicando en algunas variables que estas
pueden llegar a ser null, a este tipo de dato al que le decimos que puede valer null o cualquier otro
tipo de dato primitivo se llama NULABLE.*/
/*NULABLE: Para declarar un tipo de dato nulable (que puede valer null o cualquier otro tipo de dato
primitivo) se utiliza la siguiente sintaxis, poniendo al final del tipo de dato de la variable un signo
de interrogación, esto normalmente se aplica a variables de lectura y escritura (var) y no es bueno
aplicarlo en todas las variables, solo en las que sea necesario, para evitar errores:
var nombre_variable_nulable: tipo_de_dato? = valor*/
var variable_nulable : String? = null

/*SAFE CALLS: Las safe calls son las formas que tiene Kotlin para declarar que un código se ejecute solo
cuando el valor de la variable no sea null, pero esto solo se puede aplicar a variables NULABLES y se hace
utilizando la siguiente sitaxis:
nombre_variable_nulable?.método_aplicado_a_la_variable_nulable*/
/*En el siguiente ejemplo si no se usa una safe call el método dará error, pero si usamos el safe call, en
pantalla se imprimirá solamente el valor null, pero el programa no mostrará una excepción, esto pasa porque
al darse cuenta el programa de que la variable tiene valor null, se detiene en el signo de interrogación
y no ejecuta el método .length*/
println(variable_nulable?.length)

/*DOUBLE BANG: Esto se utiliza para cuando estamos seguros que una variable no puede tener valor null y se
indica por medio de dos signos de exclamación !!, esto hay que utilizarlo lo menos posible ya que es
considerado como malas prácticas y para hacerlo se utiliza la siguiente sintaxis:
nombre_variable_nulable!!.método_aplicado_a_una_variable_cuyo_valor_NO_puede_ser_null*/
//En el siguiente ejemplo si se usa el double bang, el método dará el error NullPointerException.
//println(variable_nulable!!.length)
```

Gestión de Excepciones

```
//GESTIÓN DE EXCEPCIONES
/*TRY CATCH: Es una forma de decirle al código que debe hacer cuando ocurra un error, para ello se
puede utilizar de manera correcta el double bang dentro de las llaves del try y en el paréntesis del catch
se debe crear un objeto Exception para indicar el tipo de error que está manejando este try catch, como en
este caso sabemos que puede ser una excepción de tipo NullPointerException, declaramos que el objeto
excepción es de tipo NullPointerException para ser más específicos en el tipo de error que estamos
gestionando, aunque ya de por sí el objeto Exception incluye dentro de sí el error NullPointerException,
si lo ponemos explícitamente es mejor*/
try {
    /*El código de dentro de las llaves del try{} se va a ejecutar y en caso de que dé una excepción, se
    brincará a la parte del catch(){} donde capturará el error de tipo NullPointerException*/
    /*DOUBLE BANG: Se utiliza dentro del catch para asegurar que dé un error y se capture el error en el
    catch*/
    variable_nulable!!.length
    /*throw: Si quiero, lo que puedo hacer es que en indicar que tipo de error se está creando manualmente,
    para así decidir si el error será manejado por el catch(){} o por el finally{}, ya que estos dependen
    del tipo de excepción que se haya creado, cuando use el throw debo indicar un mensaje que se
    despliegará en consola cuando se cree este tipo de error*/
    throw NullPointerException("Demonios! ha ocurrido un error y creamos un NullPointerException")
}catch (objetoExcepcion : NullPointerException){
    /*Capturar la excepción significa que en el caso de que el código del try dé una excepción, se
    ejecutará el código que hay dentro de las llaves del catch, pero solo cuando el error sea del tipo
    específicado entre sus paréntesis, de esa forma evitamos que se imprima en pantalla un null sin
    contexto*/
    println("La hemos regado y ha ocurrido un NullPointerException, ups!")
}finally {
    /*Si no ocurrió el tipo de excepción indicado en el paréntesis del catch, pero aún así ocurrió un
    error, el programa entrará en el finally{} y ejecutará el código que se encuentre dentro de sus
    llaves*/
    println("Como la regamos, se va a autodestruir la aplicación en 1 segundo... BOOM!!")
}

/*TRY CATCH PARA ASIGNAR UN VALOR A UNA VARIABLE: Esto se utiliza cuando estemos usando el código de
alguien más o una librería de dudosa procedencia, donde podría ocurrir un error, de esta manera cuando
ocurra ese error, asignaremos cierto valor a la variable, no podemos hacer varias cosas dentro del
```

```
    try catch, porque ese valor será asignado a la variable, por lo que se debe dar valores concretos.
    En este caso vamos a hacer un ejemplo con una operación matemática de un número divido entre cero, lo
    que dará un error.*/
    val a = 10
    val b = 0
    val resultado : Int = try {
        /*Primero lo que hace el código es intentar hacer la operación que puede tener un error*/
        a/b
    }catch (errorMatematico : Exception){
        /*Si eso no funciona y recibe un error, que es un objeto de la clase Exception, asigna el valor
        indicado entre las comillas del catch(){} a la variable resultado*/
        0
    }
    /*Después se puede agregar un condicional para analizar si el resultado fue erróneo o no, por el valor
    de la variable resultado*/
    if (resultado == 0){
        /*En este caso */
        println("El resultado de $a / $b es erróneo, por lo que a la variable resultado se le asignó " +
                "el valor de $resultado")
    }else{
        println("El resultado de $a / $b es de $resultado")
    }
    /*Es bueno contar con la herramienta try catch, pero no es bueno abusar de ella y utilizarla en todos
    lados, solamente cuando sea muy necesario sino no nos daremos cuenta de los errores del código y su
    lógica puede estar siendo manejada erróneamente ya que la clase Exception tiene varios tipos de errores,
    todos causados por motivos distintos.*/
```

Operador Elvis: Tipo de Dato Null

```
    /*OPERADOR ELVIS: Este sirve para que se pueda obtener un resultado cuando un método se aplica a una
    variable cuando su valor es null, convirtiendo así una variable nulable en una variable de tipo NO
    nulable, asignando un valor por defecto a la variable en caso de que el resultado del código ejecutado
    sea null*/
    //VARIABLE DE LECTURA Y ESCRITURA NULEABLE
    var variable_nulable_2: String? = null
    /*VARIABLE DE ESCRITURA NULEABLE: Aunque se esté utilizando un safe call, la línea de código da error
    porque se está intentando guardar un dato tipo null en una variable de tipo entero que lea el número de
    caracteres de la variable_nulable_2, por lo que se debe utilizar el OPERADOR ELVIS para resolver este
    error, este se aplica después de una SAFE CALL utilizando el signo de interrogación, seguido de dos
    puntos, luego de asignar el valor que se le debe dar a la variable en caso de que esta sea nula, usando
    la siguiente sintaxis:
    val nombre_variable : Tipo_de_dato_primitivo = variable_nuleable?.método_safe_call ?:
valor_asignado_si_el_resultado_del_safe_call_es_null*/
    val caracteres_de_la_variable_nulable : Int = variable_nulable_2?.length ?: 0
    println("El resultado del operador elvis ?: que da los caracteres de una variable nulable " +
            "es $caracteres_de_la_variable_nulable")
```

Estructura de Datos Clave: Función Anónima Maps

```
    //ESTRUCTURA DE DATOS DE CLAVE - VALOR POR MEDIO DE MAPS
    /*MAPS: Es una forma en la que a cada valor de una lista se le puede asignar una clave, como en
    inventarios donde a cada producto se le asigna un número de serie o como en los JSON de JavaScript.
    Para cada clave existe un valor, un valor puede pertenecer a varias claves pero no pueden existir dos
    claves iguales en un mismo mapa, los mapas tienen una serie de pares, a los cuales primero se les asigna
    un key o clave y luego un valor, esto crea un par, que a diferencia de las listas asocia dos cosas entre
    sí para crear un elemento dentro del mapa, a los mapas no se accede por medio de un índice específico como
    se hace en las listas, sino que se accede a sus valores por medio de las claves o key, por eso las keys no
    se pueden repetir dentro de un mismo maps.
    Existen tipos de maps mutables e inmutables.
    - Los maps inmutables se crean por medio del método mapOf() y dentro del paréntesis del método se crea
    la clave (k de key), la palabra reservada to y el valor (v de value) de cada elemento, los elementos se
    saparán entre sí por comas.
    En los mapas inmutables no se puede eliminar ni modificar los elementos, pero si se puede acceder a ellos.*/
    val edadSuperHeroesInmutable = mapOf(
        //KEY to VALUE
        "Ironman" to 35,
        "Spiderman" to 23,
        "Capitan America" to 99
    )
    /*Cuando se imprima un mapa en pantalla se mostrará su key y valor de la siguiente manera:
```

```
{Ironman=35, Spiderman=23, Capitan America=99}*/
println(edadSuperHeroesInmutable)

/*- Los maps mutables se crean por medio del método mutableMapOf() y dentro del paréntesis del método se
crea la clave (k de key), la palabra reservada to y el valor (v de value) de cada elemento.
En los mapas mutables se puede eliminar o modificar los elementos, además de poder acceder a ellos.*/
val edadSuperHeroesMutable = mutableMapOf(
    //KEY to VALUE
    "Hulk" to 40,
    "Thor" to 1500,
    "Ant-man" to 50
)
/*Los mapas mutables e inmutables se ven igual cuando se imprimen en pantalla*/
println(edadSuperHeroesMutable)
/*Los métodos para modificar los mapas mutables son los siguientes:
AGREGAR ELEMENTOS A UN MAPS
- mapsMutable.put(Key, Value): Sirve para añadir elementos a una lista mutable y al usarlo se debe indicar
la clave y valor del nuevo elemento del mapa.
- mapsMutable[Key] = Value: Se utiliza en vez del método .put() para agregar elementos a un mapa mutable,
indicando la clave y valor del nuevo elemento del mapa.

ELIMINAR ELEMENTOS DE UN MAPS
- mapsMutable.remove(): Sirve para eliminar el elemento de un mapa mutable, incluyendo su key y value,
indicado por la clave del elemento puesto en su paréntesis.

OBTENER VALORES ESPECÍFICOS DE UN MAPS
- mapsMutable o mapsInmutable[]: Se utiliza para obtener el valor del elemento de un mapa mutable o
inmutable y guardarlo en una variable o constante, indicando la clave (o key) del elemento que se
quiere extraer.

OBTENER TODAS LAS CLAVES DE UN MAPS
- mapsMutable o mapsInmutable.keys: Se utiliza para poder ver todas las claves de un maps en forma de
lista.

OBTENER TODOS LOS VALORES DE UN MAPS
- mapsMutable o mapsInmutable.values: Se utiliza para poder ver todos los valores de un maps en forma de
lista.

ORDENAR LOS ELEMENTOS DE UN MAPS
- mapsMutable o mapsInmutable.sorted(): Ordena los elementos numéricos de una lista de menor a mayor.
- mapsMutable o mapsInmutable.sortedDescending(): Ordena los elementos numéricos de una lista de mayor
a menor.
- mapsMutable o mapsInmutable.sortedBy{}: Ordena los elementos numéricos de una lista dependiendo de una
condición indicada en las llaves del condicional if, para ello se crea una variable intermedia que
representará todos y cada uno de los elementos incluidos en la lista y a la cual se le aplicará un método
matemático o lógico para analizar el valor numérico de cada uno de ellos, los elementos numéricos que
cumplan la condición se acomodarán al inicio de la lista yendo de menor a mayor y los que no la cumplan se
acomodarán al final de la lista yendo de menor a mayor también.
- mapsMutable o mapsInmutable.shuffled(): Ordena los elementos de una lista en forma aleatoria, cada
vez que el programa sea ejecutado, los elementos de la lista serán ordenados de una manera distinta.
- mapsMutable o mapsInmutable.reversed(): Ordena los elementos de una lista en forma inversa a como
se encuentra actualmente.*/
//AGREGAR ELEMENTOS A UN MAPA
edadSuperHeroesMutable.put("Wolverine", 45)
edadSuperHeroesMutable["Storm"] = 30
println(edadSuperHeroesMutable)
//ACCEDER AL VALOR DE UN ELEMENTO DEL MAPA
val edadIronman = edadSuperHeroesInmutable["Ironman"]
println(edadIronman)
//ELIMINAR UN ELEMENTO DE UN MAPA
edadSuperHeroesMutable.remove("Wolverine")
println(edadSuperHeroesMutable)
//OBTENER TODAS LAS CLAVES DE UN MAPA EN FORMA DE LISTA
println(edadSuperHeroesMutable.keys)
println(edadSuperHeroesInmutable.keys)
//OBTENER TODOS LOS VALORES DE UN MAPA EN FORMA DE LISTA
println(edadSuperHeroesMutable.values)
println(edadSuperHeroesInmutable.values)
```

Tipos de Datos Agrupados: Sets

```
/*SET: Los set son muy parecidos a las listas, pero a diferencias de las listas, estos no pueden tener
elementos repetidos, los elementos repetidos serán eliminados, solo se tomará en cuenta el primero.
Además, como los elementos dentro del set se eliminan cuando se repiten, para acceder a un set no se
puede utilizar un índice para decir la posición que se quiere accesar.
```

```
Así como con las listas y los maps, los set pueden ser inmutables (no se puede modificar ni borrar sus
valores) o mutables (se puede modificar o borrar sus valores), en ambos casos sus elementos pueden ser
accedidos.
- Los sets inmutables se crean por medio del método setOf() y dentro del paréntesis del método se indica
el valor de cada elemento como se hace con las listas, pero en este caso por ser un set, los elementos
repetidos serán borrados y solo se tomará en cuenta el primero.
En los mapas inmutables no se puede eliminar ni modificar los elementos, pero si se puede acceder a
ellos.*/
val vocales = setOf("a", "e", "i", "o", "u", "a", "e", "i", "o", "u")
/*Al imprimir el set podremos ver que los elementos repetidos son borrados*/
println(vocales)

/*- Los maps mutables se crean por medio del método mutableSetOf() y dentro del paréntesis del método se
indica el valor de cada elemento, los elementos repetidos serán borrados y solo se tomará en cuenta el
primero.
En los mapas mutables se puede eliminar o modificar los elementos, además de poder acceder a ellos.*/
val numerosFavoritos = mutableSetOf(1,2,3,4)
/*Los métodos utilizados para el manejo de datos de SETS son los mismos a los que se utilizan con las
listas.
AGREGAR ELEMENTOS A UNA LISTA O SET
- listaMutable.add: Sirve para añadir elementos a una lista mutable.

ELIMINAR ELEMENTOS DE UN SET: Como los set no cuentan con índices, se debe utilizar distinto algunos métodos
- listaMutable.removeAt(): Sirve para eliminar el elemento de una lista mutable, indicado por el valor
específico que se quiere borrar puesto en su paréntesis.
- listaMutable.removeIf{}: Sirve para eliminar los elementos de la lista siempre y cuando estos cumplan la
condición indicada en las llaves del condicional if, para ello se crea una variable intermedia que
representará todos y cada uno de los elementos incluidos en la lista y a la cual se le aplicará un método
para analizar la condición de cada uno de ellos para ver si los elimina o no.

OBTENER VALORES DE UN SET: Como los set no cuentan con índices, se debe utilizar distinto algunos métodos
- listaMutable o listaInmutable.get: Sirve para obtener elementos de una lista mutable o inmutable y
guardarlos en una variable o constante, indicando el índice del elemento que se quiere extraer.
- listaMutable o listaInmutable[]: Se utiliza en vez del método .get para obtener elementos de una lista
mutable o inmutable y guardarlos en una variable o constante, indicando el índice del elemento que se
quiere extraer.
- listaMutable o listaInmutable.first(): Se utiliza en vez del método .get o el operador [] para obtener
solo el primer elemento de una lista mutable o inmutable para guardarlo en una variable o constante.
- listaMutable o listaInmutable.firstOrNull(): Obtiene solo el primer elemento de una lista mutable o
inmutable como lo hace la función anónima first() para guardarlo en una variable o constante, pero
considera la excepción de que ese valor sea null, si es null, en consola se mostrará la palabra null pero
no ocurrirá un error en el programa.
- listaMutable o listaInmutable.first{}: Se utiliza mucho en los set para obtener el primer elemento de una
lista o set mutable o inmutable que cumpla la condición indicada en las llaves, para ello se crea una
variable intermedia que representará todos y cada uno de los elementos incluidos en la listao set y a la
cual se le aplicará un método u operación matemática o lógica para analizar la condición de cada uno de
ellos y guardar el primero que encuentre en una variable o constante.
- listaMutable o listaInmutable.firstOrNull{}: Hace lo mismo que el método kotlin.fisrt{} pero considera
que la lista puede tener valores null dentro de ella o no.

ORDENAR LOS ELEMENTOS DE UNA LISTA
- listaMutable o listaInmutable.sorted(): Ordena los elementos numéricos de una lista de menor a mayor.
- listaMutable o listaInmutable.sortedDescending(): Ordena los elementos numéricos de una lista de mayor
a menor.
- listaMutable o listaInmutable.sortedBy{}: Ordena los elementos numéricos de una lista dependiendo de una
condición indicada en las llaves del condicional if, para ello se crea una variable intermedia que
representará todos y cada uno de los elementos incluidos en la lista y a la cual se le aplicará un método
matemático o lógico para analizar el valor numérico de cada uno de ellos, los elementos numéricos que
cumplan la condición se acomodarán al inicio de la lista yendo de menor a mayor y los que no la cumplan se
acomodarán al final de la lista yendo de menor a mayor también.
- listaMutable o listaInmutable.shuffled(): Ordena los elementos de una lista en forma aleatoria, cada
vez que el programa sea ejecutado, los elementos de la lista serán ordenados de una manera distinta.
- listaMutable o listaInmutable.reversed(): Ordena los elementos de una lista en forma inversa a como
se encuentra actualmente.*/
println(numerosFavoritos)
numerosFavoritos.add(5)
println(numerosFavoritos)
numerosFavoritos.add(5)
println(numerosFavoritos)
numerosFavoritos.remove(4)
println(numerosFavoritos)
var valorDeSet: Int? = numerosFavoritos.firstOrNull { numero -> numero > 2 }
println(valorDeSet)
```

Funciones

```
/*FUNCIONES: Las funciones se utilizan (mandan a llamar) dentro de la función main pero deben ser
declaradas fuera de ella.*/
val fraseAleatoria = "Holi crayoli"
imprimirFrase(mayusculasRandom(fraseAleatoria))

/*FUNCIÓN DE EXTENSIÓN: Este tipo de función se aplica directamente al objeto Kotlin (a la variable)
para ejecutar su función y no recibe nada como parámetro.*/
val fraseAleatoria2 = "Holi crayoli, la vengaza del holi crayoli".funcion_de_extension()
imprimirFrase(mayusculasRandom(fraseAleatoria2))

/*PARÁMETROS NOMBRADOS: Se le llama parámetro nombrado a poner el nombre del parámetro declarado
en la función seguido de un signo de igual y su valor cuando es llamada la función para usarse dentro de
la función main, esto es buenas prácticas para que cualquiera entienda que valores recibe la función.*/
imprimirNombre(nombre = "Diego", apellido = "Cervantes")
/*PARPÁMETRO POR DEFECTO: Se le da un valor por default a alguno de los parámetros, pero esto se hace
desde la declaración de la función, en este caso el valor default se asignó al segundo nombre es el de
comillas abiertas "" y se hace uso de los parámetros nombrados para indicar el valor de cada parámetro.*/
imprimirNombreCompleto(nombre = "Diego", apellido = "Cervantes")

/*FUNCIONES ANÓNIMAS O LAMBDAS: Es un tipo de función que no tiene nombre, por eso se les dice anónimas
y además una característica importante de las Lambdas es que estas son consideradas como objetos que
pueden ser asignadas directamente a una variable, por lo mismo su declaración no se hace fuera de la
función main como sucedía con las funciones normales y funciones de extensión, su sintaxis es la siguiente:
val nombre_variable : (parámetro1: tipo_de_dato,..., parámetro_n: tipo_de_dato) -> tipo_de_dato_retornado = {
    it: Hace referencia al parámetro que entra a la función anónima, funciona parecido a lo que hace this
    en las funciones de extensión. Pero en las funciones anónimas no es necesario poner it, con declarar una
    variable intermedia, el contenido del parámetro se transfiere a esta para que la podamos usar, aunque
    también puede ser utilizado it.

    variable intermedia: Cuando se usa una variable intermedia primero se debe declarar su nombre, luego
    se coloca una flechita y se utiliza la variable, usando la siguiente sintaxis:
    variable_intermedia -> variable_intermedia.método()

    código de la función...
}*/
//DECLARACIÓN DE FUNCIÓN ANÓNIMA: Se declara con la sintaxis descrita arriba
val funcion_lambda_o_anonima : (String) -> Int = {
    //Se puede utilizar el it para utilizar el parámetro de la función anónima o Lambda.
    //it.length

    /*O se puede crear una variable intermedia para hacer lo mismo, es mejor crear una variable
    itermedia por si se tiene una función lambdas anidada dentro de otra.*/
    variable_intermedia -> variable_intermedia.length
}
//USO DE FUNCIÓN ANÓNIMA: Se utiliza como cualquier otra función
println(funcion_lambda_o_anonima("Holi crayoli 3, la venganza del holi"))

//FUNCIÓN ANÓNIMA USADA COMO PARÁMETRO DE OTRA FUNCIÓN
/*El caso mas utilizado en Kotlin es que cuando se ha creado una función anónima, esta sea pasada al
parámetro de la función anónima .map, para que la función .map convierta el tipo de dato y ejecute la
función de la función anónima que recibió como parámetro además de eso, esto es muy útil cuando se
quiere realizar una serie de operaciones para varias listas.*/
//En este caso el ejemplo convertirá de tipo String a numérico y calculará el numero de caracteres.
val saludos = listOf("Hello", "Hola", "Oli crayoli", "Ciao", "Hallo")
val longitudSaludos = saludos.map(funcion_lambda_o_anonima)
println(longitudSaludos)

/*FUNCIONES DE ALTO ORDEN: Crea una función que recibe como parámetro una función Lamnda, por eso
es que la acción que realiza la función lambda se debe describir dentro de la función main justo
cuando se quiera utilizar la función de alto orden. El resultado de la función de alto orden se
debe almacenar dentro de una variable. Por estandarización de Kotlin, en la función de alto orden
se le llama block al parámetro que sea de tipo función lambda.

El flujo de ejecución de esta función de alto orden en específico es que recibe dos parámetros, uno
es un String y el otro es una función anónima, dentro de la función de alto orden declarada fuera
de la función main se asigna el primer parámetro a la función lambda y se declara que la función debe
retornar un tipo de dato entero (Int), en este punto se indica que es lo que hará la función anónima
con el primer parámetro que le llega a la función de alto orden, que luego será retornado a la función
de alto orden y asignado a la variable.*/
val largoValorInicial = funcionAltoOrden(valorInicial = "Hola!", block = { valor ->
    valor.length
})
println(largoValorInicial)

/*USO DE UNA FUNCIÓN DE ORDEN SUPERIOR QUE RETORNA UNA FUNCIÓN LAMBDA: La función de orden superior fue
declarada fuera de la función main.*/
val lambdaInception = funcionInception(nombre = "Enrique")
/*Si persionamos ALT+ENTER -> Specify type explicitly, aparecerá que el tipo de la variable es de función
```

```
anónima que no recibe ningún parámetro y retorna un String:
val lambdaInception: () -> String = funcionInception(nombre = "Enrique")
Pero ahora para que se pueda ver el resultado de la función anónima, tiene que ser invocada, sin pasarle
ningún parámetro porque no los pide y si damos clic en ALT+ENTER veremos que es de tipo String.*/
val valorLambdaInception: String = lambdaInception()
println(valorLambdaInception)
```

Scope Functions: Let, With, Run y Also

```
//SCOPE FUNCTIONS: LET, WITH, RUN, APPLY, ALSO
/*LET: La función let combinado con el operador de SAFE CALLS nos permite ejecutar un código cuando la
variable nuleable no tenga valor null.*/
//VARIABLE NULA DE ESCRITURA Y LECTURA
var variable_nula : String? = null
//FUNCIÓN .let con el OPERADOR SAFE CALLS
variable_nula?.let {
    mensajeVariableNoNula -> println("La variable no es nula, es $variable_nula")
}
variable_nula = "No nuloooo!!"
variable_nula.let {
    variableIntermedia -> println("La variable no es nula, es $variable_nula")
}

/*WITH: Permite acceder directamente a las propiedades de una variable que le pasamos como
parámetro a la función with, de esta manera podemos evitar llamar a la variable varias veces
en el código y poder usar mejor el operador this o sus propiedades. Esto es más útil cuando
se aplica a listas. With no acepta valores nulables, si algún valor es null dará error.*/
//LISTA DE SOLO LECTURA
val colores = listOf("Azul", "Amarillo", "Rojo")
//FUNCIÓN with
with(colores){
    //this hace referencia a la misma lista.
    println("Nuestros colores son $this")
    /*size hace referencia al tamaño de la lista sin necesidad de usar this.size porque with accede a sus
    propiedades.*/
    println("Esta lista tiene una cantiad de colores de $size")
}

/*RUN: Ejecuta una serie de operaciones luego de haber declarado una variable, los cambios hechos a la
variable se almacenan en la misma variable usando el operador this, esto se hace para modificar de manera
sencilla una lista que se haya extraido de algún lado, ya sea un servidor, una API, etc.*/
//PRIMERO SE DECLARA UNA VARIABLE MUTABLE
val moviles = mutableListOf("Google Pixel 2XL", "Google Pixel 4A", "Huawei Redmi 9")
//Luego ya se puede utilizar la función .run
    .run{
        removeIf {
            /*El método kotlin.contains() devuelve un true o false dependiendo de si algún elemento
            del objeto lista tiene la palabra o número indicado en su paréntesis y removeIf
            elimina los elementos que cumplan la condición descrita entre sus llaves.*/
            variable_intermedia -> variable_intermedia.contains("Google")
        }
        /*La palabra resevrada this hace alusión a la variable que tiene encima la función run*/
        this
    }
println(moviles)


/*APPLY: No permite ejecutar una serie de operaciones luego de haber declarado una variable y luego
devolver el valor por si es que se quiere almacenar en una variable nueva o no se quiere utilizar
el operador this, además de que como apply es una función de extensión, esta se aplica directamente
a la variable. Apply acepta valores nuleables.*/
//PRIMERO SE DECLARA UNA VARIABLE MUTABLE
val moviles2 = mutableListOf("Motorolla", "Samsung A30", "Samsung Galaxy").apply {
    removeIf {
        /*El método kotlin.contains() devuelve un true o false dependiendo de si algún elemento
        del objeto lista tiene la palabra o número indicado en su paréntesis y removeIf
        elimina los elementos que cumplan la condición descrita entre sus llaves.*/
        variable_intermedia -> variable_intermedia.contains("Samsung")
    }
}
println(moviles2)
//EJEMPLO CON UNA LISTA NULEABLE
val colores2 : MutableList<String>? = null
//OPERADOR APPLY CON SAFE CALL, esto mismo con el operador with hubiera dado error.
colores2?.apply {
    println("Nuestros colores son $this")
```

```
        println("La cantidad de colores es $size")
    }

    /*ALSO: No permite obtener una variable, luego ejecutar un código con esa variable y devolverla como
    parámetro para que pueda volver a ser utilizada por una función más adelante, además de que como also
    es una función de extensión, esta se aplica directamente a la variable. Also acepta valores nuleables.*/
    //PRIMERO SE DECLARA UNA VARIABLE MUTABLE
    val moviles3 = mutableListOf("Iphone 4", "Iphone 8", "Samsung Note").also {
        lista -> println("El valor original de la lista es $lista")
    /*Al terminar el método .asReversed() pone la lista modificada al revés.*/
    }.asReversed()
    println(moviles3)
}

/*FUNCIÓN: Una función es un código que puede ser reutilizable y al pasarle distintos parámetros en
su paréntesis puede obtener distintos resultados y ser utilizada en cualquier otra parte del código
cuando es llamada. Las funciones se crean fuera de la función main por medio de la siguiente sintaxis:

fun nombre_funcion (parámetro1: tipo_de_dato,..., parámetro_n: tipo_de_dato) : tipo_de_dato_retornado{
    código de la función...

    return: Después de esta instrucción se indica qué es lo que va a retornar la función como resultado, solo
    puede existir un return por cada función y debe ir hasta el final porque la instrucción return hará que
    el programa salga de la función, esto ocurre excepto en los condicionales o bucles, donde dependiendo de
    las condiciones se puede devolver un resultado u otro. El tipo de dato que retorna la función se indica en
    la parte de arriba donde se encuentra el nombre.

    unit (es lo mismo que void): En Kotlin se utiliza la palabra reservada unit para indicar que la función
    no retorna nada, esto se puede poner explícitamente en el tipo_de_dato_retornado o no, con que no haya un
    return en la función el código o no haya un tipo de dato específico que retorna la función (ni los dos
    puntos), Kotlin entiende que no se va a devolver nada.
}*/

//FUNCIÓN CON RETURN
/*Como ejemplo se creará una función que convierta una frase a mayúscula o minúscula dependiendo del valor
de un número aleatorio creado dentro de la misma función*/
fun mayusculasRandom(frase: String) : String{
    //Creación de número aleatorio con un rango de 0 a 99
    val rango =0..99                    //Rango numérico
    val numAleatorio = rango.random()  //Número aleatorio con el método random()

    /*La operación numérica que se va a ejecutar es que el número aleatorio se va a dividir entre 2 y si el
    residuo es igual a cero, todas las letras del parámetro serán convertidas a mayúsculas, sino se quedarán
    en minúsculas, para dividir entre un número y obtener el residuo como resultado se utiliza el método
    kotlin.rem(divisor_de_la_división)*/
    /*El método return se puede aplicar a los posibles resultados que debe devolver la función o ser aplicado
    al condicional if en general.
    if(numAleatorio.rem(2) == 0){
        /*El método kotlin.toUpperCase() pone en mayúsculas todas las letras de un texto.*/
        return frase.toUpperCase()
    }else{
        /*El método kotlin.toLowerCase() pone en minúsculas todas las letras de un texto.*/
        return frase.toLowerCase()
    }*/
    return if(numAleatorio.rem(2) == 0){
        /*El método kotlin.toUpperCase() pone en mayúsculas todas las letras de un texto.*/
        frase.toUpperCase()
    }else{
        /*El método kotlin.toLowerCase() pone en minúsculas todas las letras de un texto.*/
        frase.toLowerCase()
    }
}

//FUNCIÓN CON UNIT (que es lo mismo que void en los demás lenguajes de programación)
/*En las funciones Unit que no retornan nada no es necesario poner explícitamente la palabra Unit, se
puede dejar sin nada y el programa entiende que la función no devuelve nada.*/
fun imprimirFrase(frase: String) : Unit{
    println("Tu frase es $frase")
}
```

Función de Extensión

```
/*FUNCIÓN DE EXTENSIÓN: Este tipo de función se utiliza directo en la variable por medio de la nomenclatura
del punto, así como se aplican los métodos .toUpperCase(), este tipo de función no recibe parámetros porque
se aplica directo a un objeto Kotlin (que es cualquier variable o cosa del código) y para utilizar lo que
sea que esté entrando a la función se hace uso de la palabra reservada this:
```

```
fun tipo_de_dato_que_maneja_la_funcion.nombre_funcion_de_extensión () : tipo_de_dato_retornado{
    código de la función...

    this: Esto se utiliza en las funciones de extensión para reemplazar la variable declarada en el parámtro
    de una función cualquiera para poder manipular el valor que entre a la función y realizar operaciones con
    él.

    return: Después de esta instrucción se indica qué es lo que va a retornar la función como resultado, solo
    puede existir un return por cada función y debe ir hasta el final porque la instrucción return hará que
    el programa salga de la función, esto ocurre excepto en los condicionales o bucles, donde dependiendo de
    las condiciones se puede devolver un resultado u otro. El tipo de dato que retorna la función se indica en
    la parte de arriba donde se encuentra el nombre.

    unit (es lo mismo que void): En Kotlin se utiliza la palabra reservada unit para indicar que la función
    no retorna nada, esto se puede poner explícitamente en el tipo_de_dato_retornado o no, con que no haya un
    return en la función el código o no haya un tipo de dato específico que retorna la función, Kotlin
    entiende que no se va a devolver nada.
}*/
fun String.funcion_de_extension() : String{
    val rango =0..99                    //Rango numérico
    val numAleatorio = rango.random()   //Número aleatorio con el método random()

    return if(numAleatorio.rem(2) == 0){
        /*La palabra reservada this reemplaza el uso de la variable declarada en los paréntesis de la
        función que representa al parámetro y hace referencia a lo que sea a lo que se le esté
        aplicando la función de extensión*/
        this.toUpperCase()             //Convierte a letras mayúsculas
    }else{
        this.toLowerCase()             //Convierte a letras minúsculas
    }
}
```

Tipos de Parámetros en las Funciones

```
/*TIPOS DE PARÁMETROS: Existen los parámetros nombrados o los parámetros por defecto.
PARÁMETROS NOMBRADOS: Se le llama parámetro nombrado a poner el nombre del parámetro declarado
en la función seguido de su valor cuando es llamada la función para usarse dentro de la función main.
PARÁMETROS POR DEFECTO: Se utiliza cuando el valor de alguno de los parámetros debería tener un valor
por default, si el usuario pone otro valor este se sobreescribe pero sino se queda así.*/
//PARPÁMETRO NOMBRADO: Esto se nota cuando se manda a llamar la función dentro de l función main
fun imprimirNombre(nombre: String, apellido: String) {
    println("Mi nombre es $nombre $apellido")
}
//PARPÁMETRO POR DEFECTO: Se le da un valor por default a alguno de los parámetros
fun imprimirNombreCompleto(nombre: String, segundoNombre:String = "", apellido: String) {
    println("Mi nombre es $nombre $segundoNombre $apellido")
}
```

Funciones Anónimas o Lambdas

```
/*FUNCIONES ANÓNIMAS O LAMBDAS: Es un tipo de función que pueden ser asignadas directamente a una variable,
por lo que se declaran dentro de la función main.*/

/*FUNCIONES DE ALTO ORDEN: La característica primordial de este tipo de funciones es que reciben como
parámetro otras funciones, en específico funciones anónimas (o lambdas que es lo mismo) ya que estas
representan valores con un tipo de dato primitivo, devolviendo así el resultado de otra función.
Es un estándar de Kotlin que cuando un parámetro vaya a ser una función lambda se le llame block, la
sintaxis se muestra a continuación:

fun nombre_funcion (parámetro1: tipo_de_dato,..., block: (parámetro_lambda_1: tipo_de_dato,...) ->
tipo_de_dato_retornado_lambda) : tipo_de_dato_retornado{
    código de la función...

    return: En las funciones de alto orden se utiliza el return para conectar ambas funciones, pero lo que
    hará la función anónima se declara dentro de la función main, justo cuando se quiera utilizar la función
    de alto orden, por lo que una misma función podría hacer varias cosas pero se tendría que declarar que va
    a hacer cada una.
 }*/
fun funcionAltoOrden(valorInicial : String, block : (String) -> Int) : Int{
```

```
    /*Dentro de la misma función se le pasa como parámetro a la función lambda el parámetro del valorInicial
    de esta misma función.*/
    return block(valorInicial)
}
//El código sigue dentro de la función main.

/*FUNCIONES DE ALTO ORDEN QUE RETORNAN UNA FUNCIÓN LAMBDA COMO RESULTADO: Cuando queramos que la función de
alto orden regrese una función lambda como resultado en vez de un dato primitivo se usa la siguiente
sintaxis:

fun nombre_funcion (parámetro1: tipo_de_dato,...) : (parámetro_lambda_1: tipo_de_dato,...) ->
tipo_de_dato_retornado_lambda {
    código de la función...

    return {
        En las funciones de alto orden que retornan una lambda se utiliza el return para devolver una lambda con
        el tipo de dato indicado arriba que retorna la lambda.
    }
 }*/
fun funcionInception(nombre: String) : () -> String {
    return {
        "Hola desde la función anónima Inception $nombre"
    }
}
```

Referencias:

Platzi, Giuseppe Vetri, “Curso de Kotlin desde Cero”, [Online], Available: https://platzi.com/clases/2245-kotlin/36584-introduccion-al-curso-de-kotlin/