Ejemplo 1:

```
// Paquete que organiza el programa dentro de un espacio de nombres llamado 'my.program'

package my.program


// Función principal que se ejecuta al iniciar el programa

fun main(args: Array<String>) {

  // Imprime el mensaje "Hello, world!" en la consola

  println("Hello, world!")

}
```

Documentación: Este programa básico en Kotlin utiliza la función main como punto de entrada para ejecutar código. En este caso, imprime el mensaje “Hello, ¡world!" en la consola. Es un ejemplo clásico para iniciar el aprendizaje de un lenguaje de programación, demostrando la estructura fundamental de un programa.

Ejemplo 2:

```
// Paquete que organiza el programa dentro de un espacio de nombres llamado 'my.program'

package my.program


// Declaración de un objeto en Kotlin, que es un singleton y reemplaza la necesidad de una clase con una única instancia

object App {

  // Anotación @JvmStatic para que la función se pueda usar como método estático en entornos Java

  @JvmStatic fun main(args: Array<String>) {

    // Imprime "Hello World" en la consola

    println("Hello World")

  }
```

```
}
```

Documentación: Este programa en Kotlin utiliza un objeto (object) llamado App como punto principal de ejecución. Dentro del objeto, la función main es definida con el modificador @JvmStatic, lo que garantiza compatibilidad con Java y permite que la función se ejecute como un método estático. Este programa imprime el mensaje "Hello World" en la consola, mostrando la estructura básica de un programa que utiliza objetos en Kotlin.

Ejemplo 3:

```
// Declaración del paquete que organiza el programa en un espacio de nombres llamado 'my.program'

package my.program


// Clase App que contiene el objeto compañero para definir el punto de entrada del programa

class App {

  companion object { // Objeto compañero que actúa como un singleton dentro de la clase

    // Anotación @JvmStatic para habilitar la compatibilidad con métodos estáticos en Java

    @JvmStatic

    fun main(args: Array<String>) {

      // Imprime "Hello World" en la consola

      println("Hello World")

    }

  }

}
```

Documentación: Este programa utiliza una clase App con un objeto compañero (companion object) para definir el punto principal de ejecución. La anotación @JvmStatic asegura que la función main pueda ser utilizada como un método

estático cuando se ejecuta en entornos compatibles con Java. Este programa, al ejecutarse, imprime "Hello World" en la consola.

Ejemplo 4:

```
// Paquete que organiza el programa dentro de un espacio de nombres llamado 'my.program'

package my.program


// Función principal que sirve como punto de entrada al programa

fun main(vararg args: String) { // 'vararg' permite pasar un número variable de argumentos

    println("Hello, world!") // Imprime el mensaje "Hello, world!" en la consola

}
```

Documentación: Este programa es un ejemplo básico que utiliza la función main en Kotlin, con un parámetro de tipo vararg para aceptar argumentos variables desde la línea de comandos. Su propósito principal es imprimir "Hello, world!" en la consola, funcionando como una introducción simple y efectiva al lenguaje.

Ejemplo 5:

```
fun main(args: Array<String>) {

    // Muestra un mensaje para que el usuario ingrese dos números

    println("Enter Two number")

    // Lee la entrada del usuario como una línea y divide los números usando el espacio como separador

    // El operador "!!" asegura que no se genere una NullPointerException si readLine() retorna nulo

    var (a, b) = readLine()!!.split(' ')


    // Convierte los números ingresados a enteros y llama a la función maxNum para encontrar el máximo

    println("Max number is : ${maxNum(a.toInt(), b.toInt())}")
```

```
}


// Función que determina el mayor de dos números enteros
fun maxNum(a: Int, b: Int): Int {
  // Usa una expresión 'if' para decidir el valor máximo entre los dos números
  var max = if (a > b) {
    println("The value of a is $a") // Imprime el valor de 'a' si es mayor que 'b'
    a // Retorna 'a' como el número mayor
  } else {
    println("The value of b is $b") // Imprime el valor de 'b' si es mayor que 'a'
    b // Retorna 'b' como el número mayor
  }
  return max; // Retorna el valor máximo calculado
}
```

Documentación: Este programa en Kotlin solicita al usuario que ingrese dos números y determina cuál de ellos es el mayor. Utiliza la función maxNum para calcular el valor máximo de los dos números ingresados. También implementa el operador de "aseguración" (!!) para prevenir errores de referencia nula (NullPointerException) durante la lectura de entrada.

Ejemplo 6:

```
// Declara una variable llamada 'text' que almacena el contenido del textField
val text = view.textField?.text?.toString() ?: ""
/*
  - 'view.textField': Accede al componente 'textField' de un objeto 'view'.
  - '?.': El operador de llamada segura verifica si 'textField' es nulo.
      Si no es nulo, continúa evaluando 'textField.text'.
  - 'text?.toString()': Convierte el texto a cadena si 'text' no es nulo.
```

- '?: "": El operador Elvis proporciona un valor alternativo ("") si la expresión es nula.

*/

Documentación: La expresión en Kotlin val text = view.textField?.text?.toString() ?: "" utiliza las características del lenguaje relacionadas con la seguridad ante valores nulos (null safety). Este código asigna el contenido de un campo de texto (textField) a la variable text, garantizando que, en caso de que el campo de texto o su contenido sean nulos, la variable se inicialice con una cadena vacía ("").

Ejemplo 7:

```
// Declaración de la anotación personalizada 'Strippable'
@Target(
    // Lista de posibles objetivos donde se puede aplicar la anotación
    AnnotationTarget.CLASS,         // Puede aplicarse a clases
    AnnotationTarget.FUNCTION,      // Puede aplicarse a funciones
    AnnotationTarget.VALUE_PARAMETER, // Puede aplicarse a parámetros de función
    AnnotationTarget.EXPRESSION     // Puede aplicarse a expresiones
)
annotation class Strippable // Define la anotación personalizada 'Strippable'
```

Documentación: El código compartido define una anotación en Kotlin llamada Strippable y especifica los objetivos en los que se puede aplicar esta anotación utilizando @Target. Las anotaciones son herramientas potentes en Kotlin que permiten asociar metadatos con el código, lo que puede ser utilizado para influir en la lógica de ejecución, análisis de código, o generación de código durante el tiempo de compilación o en tiempo de ejecución.

Ejemplo 8:

```
// Especifica los objetivos donde se puede aplicar esta anotación
@Target(
    AnnotationTarget.CLASS,         // Puede aplicarse a clases
    AnnotationTarget.FUNCTION,      // Puede aplicarse a funciones
```

```
  AnnotationTarget.VALUE_PARAMETER, // Puede aplicarse a parámetros de funciones
  AnnotationTarget.EXPRESSION      // Puede aplicarse a expresiones
)

// Define el nivel de retención de la anotación
@Retention(AnnotationRetention.SOURCE)
/*
  - AnnotationRetention.SOURCE: La anotación solo estará presente en el código fuente.
  - No estará en el bytecode compilado ni será visible en tiempo de ejecución.
*/

// Indica que esta anotación debe ser incluida en la documentación generada
@MustBeDocumented
// Declara la clase de anotación personalizada llamada 'Fancy'
annotation class Fancy
```

Documentación: La anotación Fancy definida en este código sirve como una anotación personalizada en Kotlin. Está configurada para ser aplicada a diferentes elementos de código (como clases y funciones) y, además, posee ciertas características adicionales como @Retention para controlar su visibilidad y @MustBeDocumented para garantizar que sea incluida en la documentación generada.

Ejemplo 9:

```
// Importación necesaria para utilizar el método Arrays.toString para formatear arreglos
import java.util.Arrays

fun main() {
```

```
    // Crear un arreglo de tamaño 5 donde cada elemento es generado dinámicamente
    var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
    /*
       - 'Array<String>': Declara un arreglo de tipo String.
       - 'size = 5': Define el tamaño del arreglo como 5.
       - 'init = { index -> "Item #$index" }': Función lambda que genera cada elemento,
         usando el índice para personalizar el contenido (e.g., "Item #0", "Item #1").
    */

    // Imprimir el contenido del arreglo en un formato legible
    println(Arrays.toString(strings))
    // 'Arrays.toString(strings)' convierte el arreglo en una representación de texto,
    // mostrando todos los elementos entre corchetes.

    // Imprimir el tamaño del arreglo
    println(strings.size)
    // '.size' obtiene el tamaño del arreglo, en este caso 5.
}
```

Documentación: Este programa utiliza la función Array en Kotlin para crear y trabajar con un arreglo de tamaño predefinido. Cada elemento se genera dinámicamente a través de un inicializador (init), y luego se imprimen el contenido del arreglo y su tamaño. Además, se utiliza Arrays.toString para mostrar el arreglo en un formato legible.

Ejemplo 10:

```
fun main() {
    // Declaración de un arreglo de números en punto flotante (Double)
    val doubles = doubleArrayOf(1.5, 3.0)
```

```
    /*
       - 'doubleArrayOf': Crea un arreglo de tipo Double.
       - Los valores iniciales son 1.5 y 3.0.
    */

    // Calcular y mostrar el promedio de los elementos en el arreglo
    print(doubles.average()) // prints: 2.25
    /*
       - 'average()': Método para calcular el promedio de los elementos en el arreglo.
       - En este caso, el promedio de (1.5 + 3.0) / 2 es 2.25.
    */

    // Declaración de un arreglo de números enteros (Int)
    val ints = intArrayOf(1, 4)
    /*
       - 'intArrayOf': Crea un arreglo de tipo Int.
       - Los valores iniciales son 1 y 4.
    */

    // Calcular y mostrar el promedio de los elementos en el arreglo
    println(ints.average()) // prints: 2.5
    /*
       - 'average()': Método para calcular el promedio de los elementos en el arreglo.
       - En este caso, el promedio de (1 + 4) / 2 es 2.5.
    */
}
```

Documentación: Este fragmento de código demuestra cómo trabajar con arreglos numéricos en Kotlin, utilizando las funciones doubleArrayOf y intArrayOf para crear arreglos de tipo Double y Int, respectivamente. También utiliza el método average() para calcular el promedio de los elementos en cada arreglo.

Ejemplo 11:

```
fun main() {
  // Crear un arreglo de 5 elementos donde cada posición contiene el cuadrado del índice como cadena de texto
  val asc = Array(5, { i -> (i * i).toString() })
  /*
    - 'Array(5)': Crea un arreglo de tamaño 5.
    - 'init = { i -> (i * i).toString() }':
      - 'i': Representa el índice actual del arreglo (de 0 a 4).
      - '(i * i)': Calcula el cuadrado del índice.
      - '.toString()': Convierte el resultado a tipo String.
  */

  // Iterar sobre el arreglo y mostrar cada elemento
  for (s: String in asc) {
    println(s) // Imprime el valor actual de 's' en la consola
  }
}
```

Documentación: Este código en Kotlin crea un arreglo de 5 elementos (Array) y lo llena dinámicamente con los cuadrados de los índices (i * i). Luego, itera sobre el arreglo y muestra cada elemento en la consola. Es un ejemplo simple pero efectivo para comprender cómo inicializar y recorrer arreglos en Kotlin.

Ejemplo 12:

```
// Declaración de un arreglo llamado 'a' que contiene tres elementos enteros: 1, 2 y 3
```

```
val a = arrayOf(1, 2, 3)

/*

  - 'arrayOf': Función de Kotlin que crea un arreglo.

  - Los valores iniciales dentro de los paréntesis se asignan directamente como los elementos del arreglo.

  - El tipo de datos del arreglo se infiere automáticamente como 'Array<Int>' debido a que los valores proporcionados son enteros.

*/
```

Documentación: El código val a = arrayOf(1, 2, 3) crea un arreglo en Kotlin de tipo Array con tamaño 3. Los valores iniciales del arreglo son [1, 2, 3]. Es una forma sencilla de inicializar un arreglo con valores específicos y trabajar con colecciones en Kotlin.

Ejemplo 13:

```
// Crear un arreglo de tamaño 3 donde cada elemento se genera dinámicamente

val a = Array(3) { i -> i * 2 }

/*

  - 'Array(3)': Declara un arreglo de tamaño 3.

  - '{ i -> i * 2 }': Función lambda que define el valor de cada elemento.

    - 'i': Representa el índice actual (0, 1, 2).

    - 'i * 2': Calcula el doble del índice y lo asigna como valor del elemento.

  - Resultado: El arreglo 'a' contiene [0, 2, 4].

*/
```

Documentación: Este fragmento de código crea un arreglo en Kotlin utilizando el constructor Array. El tamaño del arreglo es 3, y cada elemento se genera dinámicamente con una función lambda que toma el índice (i) y calcula su doble (i * 2). Como resultado, el arreglo contiene los valores [0, 2, 4].

Ejemplo 14:

```
// Declaración de un arreglo de enteros que puede contener valores nulos

val a = arrayOfNulls<Int>(3)
```

```
/*
  - 'arrayOfNulls<Int>(3)': Crea un arreglo de tamaño 3.
    - 'Int?': Indica que los elementos del arreglo pueden contener valores nulos.
    - Inicialmente, todos los elementos se configuran como 'null'.
*/
```

Documentación: El código val a = arrayOfNulls(3) crea un arreglo en Kotlin de tipo Array con tres elementos inicializados como null. Este enfoque es útil cuando se necesita crear un arreglo para contener valores que se asignarán posteriormente, pero que inicialmente están vacíos.

Ejemplo 15:

```
fun main() {
  // Repetir el bloque de código 10 veces
  repeat(10) { i ->
    /*
      - 'repeat(10)': Ejecuta el bloque de código 10 veces.
      - 'i ->': La variable 'i' representa el índice de la iteración, que comienza en 0.
    */

    // Imprime un mensaje fijo en cada iteración
    println("This line will be printed 10 times")

    // Imprime un mensaje dinámico que incluye el número de iteración (inicio en 1)
    println("We are on the ${i + 1}. loop iteration")
    /*
      - '${i + 1}': Ajusta el índice de la iteración (inicia en 0) para que sea más legible (inicio en 1).
    */
```

```
    }
}
```

Documentación: El código utiliza la función repeat en Kotlin para ejecutar un bloque de código 10 veces. Durante cada iteración, se imprime un mensaje fijo y otro dinámico que indica el número de la iteración actual. Este enfoque es ideal para simplificar tareas repetitivas sin necesidad de configurar explícitamente estructuras como bucles for.

Ejemplo 16:

```
fun main() {
    // Crear una lista con tres elementos de tipo String
    val list = listOf("Hello", "World", "!")
    /*
       - 'listOf': Función que crea una lista inmutable.
       - Los elementos de la lista son "Hello", "World" y "!".
    */

    // Iterar sobre la lista y mostrar cada elemento en la consola
    for (str in list) { // 'str' representa cada elemento de la lista en cada iteración
        print(str) // Imprime el elemento actual sin salto de línea
    }
}
```

Documentación: Este código crea una lista de cadenas utilizando la función listOf en Kotlin y luego utiliza un bucle for para iterar a través de cada elemento de la lista y mostrarlo en la consola. Es una forma sencilla y directa de trabajar con colecciones en Kotlin.

Ejemplo 17:

```
// Bucle 'while' en Kotlin
while(condition) {
```

```
    // Ejecuta este bloque de código mientras la condición sea verdadera
    doSomething()
    /*
        - 'condition': Es una expresión booleana que se evalúa antes de cada iteración.
        - Si la condición es falsa desde el inicio, el cuerpo del bucle no se ejecutará.
        - 'doSomething()': Representa cualquier operación que deseas realizar en cada iteración.
    */
}

// Bucle 'do-while' en Kotlin
do {
    // Ejecuta este bloque de código al menos una vez
    doSomething()
    /*
        - 'doSomething()': Este bloque de código siempre se ejecutará al menos una vez,
                incluso si la condición es falsa desde el inicio.
    */
} while (condition)
    // 'condition': Es una expresión booleana que se evalúa después de cada iteración.
    // Si es verdadera, el bucle continuará ejecutándose.
```

Documentación: El código presentado incluye dos estructuras de bucle en Kotlin: while y do-while. Ambos permiten ejecutar un bloque de código repetidamente, dependiendo de una condición. Mientras que el while evalúa la condición antes de cada iteración, el do-while ejecuta el bloque de código al menos una vez antes de verificar la condición.

Ejemplo 18:

```
while (true) {
```

```
  // Comienza un bucle infinito

  if (condition1) {
    continue
    /*
      - 'continue': Salta inmediatamente al inicio de la próxima iteración del bucle.
      - El resto del cuerpo del bucle no se ejecutará en esta iteración.
      - Esto es útil para omitir ciertas operaciones cuando se cumplen condiciones específicas.
    */
  }

  if (condition2) {
    break
    /*
      - 'break': Termina la ejecución del bucle por completo.
      - Útil para salir del bucle cuando se cumple una condición específica.
    */
  }

  // Cualquier código aquí se ejecutará solo si 'condition1' es falsa
  // porque 'continue' omitiría este bloque si se cumple 'condition1'.
}
```

Documentación: Este código implementa un bucle while infinito (while(true)) y utiliza las instrucciones de control continue y break para gestionar su flujo. Es útil para comprender cómo interrumpir o pasar a la siguiente iteración de un bucle según diferentes condiciones.

Ejemplo 19:

```
// Declaración de un mapa utilizando la función hashMapOf
var map = hashMapOf(1 to "foo", 2 to "bar", 3 to "baz")
/*
  - 'hashMapOf': Crea un mapa mutable (HashMap) con pares clave-valor.
  - '1 to "foo"': Define un par clave-valor (clave: 1, valor: "foo").
  - Resultado: {1=foo, 2=bar, 3=baz}
*/

// Bucle para iterar sobre las entradas del mapa
for ((key, value) in map) {
  /*
    - 'key': Representa la clave actual del mapa.
    - 'value': Representa el valor asociado a la clave actual.
  */

  // Imprime cada clave y su valor correspondiente en el mapa
  println("Map[$key] = $value")
  /*
    - 'Map[$key]': Formatea la clave para que se muestre en un índice estilo mapa.
    - '$value': Muestra el valor asociado a la clave.
  */
}
```

Documentación: Este código utiliza un bucle for para iterar sobre un mapa (Map) en Kotlin, accediendo a cada clave (key) y su correspondiente valor (value). El mapa se inicializa como un HashMap con pares clave-valor, y en cada iteración del bucle se imprime un mensaje que muestra el contenido actual del mapa.

Ejemplo 20:

```
// Función para calcular el factorial de un número utilizando recursión
fun factorial(n: Long): Long =
  if (n == 0) 1 // Caso base: el factorial de 0 es 1
  else n * factorial(n - 1)
  /*
    - Caso recursivo: multiplica 'n' por el factorial de 'n - 1'.
    - La recursión continúa hasta llegar al caso base (n == 0).
  */

// Llamada a la función factorial y mostrar el resultado
println(factorial(10)) // Imprime el factorial de 10
```

Documentación: Este código en Kotlin define una función recursiva llamada factorial que calcula el factorial de un número. Utiliza una expresión if para manejar el caso base y la lógica recursiva. Posteriormente, imprime el factorial del número 10, que es igual a 3628800.

Ejemplo 21:

```
// Crear una lista de números enteros
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
/*
  - 'listOf': Función que crea una lista inmutable.
  - Los elementos iniciales son números enteros: 1, 2, 3, ..., 0.
  - Resultado: List<Int> con contenido [1, 2, 3, 4, 5, 6, 7, 8, 9, 0].
*/

// Transformar cada número de la lista en una cadena con formato "Number X"
val numberStrings = numbers.map { "Number $it" }
```

```
/*
   - 'map': Función que aplica una operación sobre cada elemento de la lista original.
   - 'it': Representa el elemento actual de la lista en la iteración.
   - Resultado: List<String> con contenido ["Number 1", "Number 2", ..., "Number 0"].
*/
```

Documentación: Este fragmento de código en Kotlin crea una lista de números enteros (numbers) y luego transforma cada número de la lista en una cadena con el formato "Number X" mediante la función map. La lista resultante (numberStrings) contiene las cadenas generadas.

Ejemplo 22:

```
// Crear una lista inmutable de cadenas (String)
val list = listOf("Item 1", "Item 2", "Item 3")
/*
   - 'listOf': Crea una lista inmutable con los elementos especificados.
   - Cada elemento es de tipo String, en este caso: "Item 1", "Item 2", y "Item 3".
   - La lista generada es de tipo List<String>.
*/

// Imprimir el contenido de la lista en la consola
println(list)
/*
   - 'println': Función que imprime datos en la consola.
   - Al imprimir la lista, se muestra su representación en formato [Item 1, Item 2, Item 3].
*/
```

Documentación: El código crea una lista inmutable (List) en Kotlin con tres elementos de tipo String, y luego la imprime en la consola. Dado que la lista es inmutable, no se pueden agregar, eliminar o modificar elementos después de su creación.

Ejemplo 23:

```
// Crear un mapa inmutable con pares clave-valor (Integer -> String)
val map = mapOf(
   Pair(1, "Item 1"), // Par clave-valor (clave: 1, valor: "Item 1")
   Pair(2, "Item 2"), // Par clave-valor (clave: 2, valor: "Item 2")
   Pair(3, "Item 3")  // Par clave-valor (clave: 3, valor: "Item 3")
)
/*
   - 'mapOf': Función estándar que crea un mapa inmutable.
   - 'Pair': Clase utilizada para definir pares clave-valor.
   - El mapa resultante tiene los pares: {1=Item 1, 2=Item 2, 3=Item 3}.
*/

// Imprimir el contenido del mapa
println(map)
/*
   - 'println': Función que imprime datos en la consola.
   - Al imprimir el mapa, se muestra su representación en formato "{1=Item 1, 2=Item 2, 3=Item 3}".
*/
```

Documentación: Este código crea un mapa inmutable (Map) en Kotlin utilizando la función mapOf. El mapa contiene pares clave-valor, donde las claves son de tipo Integer y los valores son de tipo String. Posteriormente, se imprime el contenido del mapa en la consola.

Ejemplo 24:

```
// Crear un conjunto inmutable de números enteros
val set = setOf(1, 3, 5)
```

```
/*
   - 'setOf': Función que crea un conjunto inmutable.
   - Los elementos iniciales son: 1, 3 y 5.
   - No se permiten duplicados dentro del conjunto.
   - El conjunto resultante tiene tipo Set<Int>, ya que los elementos son enteros.
*/


// Imprimir el contenido del conjunto
println(set)
/*
   - 'println': Función que imprime datos en la consola.
   - La representación del conjunto aparece como "[1, 3, 5]".
   - Nota: Aunque el orden aparece estable aquí, un Set no garantiza que los elementos mantengan el mismo orden.
*/
```

Documentación: Este código crea un conjunto inmutable (Set) en Kotlin utilizando la función setOf. Un conjunto almacena elementos únicos y no garantiza un orden específico. Luego, imprime el contenido del conjunto utilizando println.

Ejemplo 25:

```
buildscript {
  // Define una variable para la versión de Kotlin
  ext.kotlin_version = '1.0.3'
  /*
     - 'ext.kotlin_version': Declara una variable con la versión específica de Kotlin.
     - Esta versión será utilizada para gestionar la dependencia del complemento de Kotlin.
  */
```

```
  // Configura los repositorios donde Gradle buscará las dependencias
  repositories {
    mavenCentral()
    /*
       - 'mavenCentral': Define el repositorio Maven Central como fuente para las dependencias.
       - Maven Central es un repositorio público ampliamente utilizado para bibliotecas Java y Kotlin.
    */
  }
}

dependencies {
  // Agrega la dependencia para el complemento de Kotlin
  classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
  /*
     - 'classpath': Indica que esta dependencia se necesita en el tiempo de configuración de Gradle.
     - 'org.jetbrains.kotlin:kotlin-gradle-plugin': Es el complemento necesario para utilizar Kotlin con Gradle.
     - '$kotlin_version': Usa la versión de Kotlin definida anteriormente.
    */
}
```

Documentación: Este código configura un archivo Gradle para un proyecto que utiliza Kotlin como lenguaje principal. En particular, define el script de construcción (buildscript) y las dependencias necesarias para integrar el complemento de Kotlin (kotlin-gradle-plugin) en el proyecto. Gradle es una herramienta de automatización de compilación utilizada para gestionar dependencias y tareas en proyectos.

Ejemplo 26:

```
import kotlinx.coroutines.* // Importación necesaria para trabajar con corrutinas


fun main(args: Array<String>) {
  // Lanzar una nueva corrutina en el pool de hilos común
  launch(CommonPool) {
    // Retraso no bloqueante de 1 segundo
    delay(1000L)
    /*
      - 'delay(1000L)': Pausa la ejecución de la corrutina durante 1 segundo sin bloquear el hilo principal.
      - '1000L': Especifica el tiempo en milisegundos.
    */

    // Imprime "World!" después del retraso
    println("World!")
  }

  // Imprime "Hello," mientras la corrutina está en pausa
  println("Hello,")

  // Bloquea el hilo principal durante 2 segundos para evitar que el programa termine prematuramente
  Thread.sleep(2000L)
  /*
    - 'Thread.sleep(2000L)': Hace que el hilo principal se detenga durante 2 segundos.
```

- Esto asegura que la corrutina tenga tiempo suficiente para completar su ejecución.

*/

}

Documentación: Este código muestra un ejemplo de cómo trabajar con corrutinas en Kotlin para ejecutar tareas de manera concurrente y no bloqueante. Utiliza la función launch para crear una nueva corrutina en el CommonPool. Mientras la corrutina realiza una operación con retraso (delay), el resto del programa continúa ejecutándose de manera independiente.

Ejemplo 27:

```
// Declarar una variable de tipo String e inicializarla con el valor "Hello!"
val str = "Hello!"


// Comprobar la longitud de la cadena y realizar acciones basadas en esa longitud
if (str.length == 0) {
    // Caso: La longitud de la cadena es 0
    print("The string is empty!")
    /*
        - 'str.length': Obtiene la longitud de la cadena.
        - Si la longitud es 0, se imprime: "The string is empty!".
    */
} else if (str.length > 5) {
    // Caso: La longitud de la cadena es mayor que 5
    print("The string is short!")
    /*
        - Si la longitud es mayor que 5, se imprime: "The string is short!".
    */
} else {
```

```
    // Caso: La longitud de la cadena es menor o igual a 5
    print("The string is long!")
    /*
       - En cualquier otro caso, se imprime: "The string is long!".
    */
}
```

Documentación: Este código verifica la longitud de una cadena (str) y realiza diferentes acciones dependiendo de su longitud. Utiliza una estructura if-else para implementar la lógica condicional.

Ejemplo 28:

```
// Declarar una variable llamada 'str' y asignarle un valor basado en una condición
val str = if (condition) "Condition met!" else "Condition not met!"
/*
   - 'condition': Representa una expresión booleana (true/false) que se evalúa.
   - Si 'condition' es verdadera:
      - Se asigna el valor "Condition met!" a la variable 'str'.
   - Si 'condition' es falsa:
      - Se asigna el valor "Condition not met!" a la variable 'str'.
*/
```

Documentación: Este fragmento de código utiliza una expresión if-else en Kotlin para asignar un valor a la variable str dependiendo de la condición especificada (condition). Es una forma concisa de tomar decisiones en el código utilizando expresiones en lugar de estructuras tradicionales.

Ejemplo 29:

```
// Evaluación de condiciones utilizando 'when'
when {
   str.length == 0 -> print("The string is empty!")
   /*
```

```
    - Verifica si la longitud de 'str' es igual a 0.
    - Si es verdadera, imprime: "The string is empty!".
  */

  str.length > 5  -> print("The string is short!")
  /*
    - Verifica si la longitud de 'str' es mayor que 5.
    - Si es verdadera, imprime: "The string is short!".
  */

  else          -> print("The string is long!")
  /*
    - Se ejecuta si ninguna de las condiciones anteriores es verdadera.
    - Imprime: "The string is long!".
  */
}
```

Documentación: Este código utiliza la expresión when en Kotlin para realizar una evaluación condicional basada en la longitud de una cadena (str). when funciona como una alternativa más legible y concisa a múltiples bloques if-else.

Ejemplo 30:

```
// Evaluar la variable 'x' utilizando la expresión 'when'
when (x) {
  "English" -> print("How are you?")
  /*
    - Si 'x' es igual a "English", imprime: "How are you?".
    */
```

```
    "German" -> print("Wie geht es dir?")
    /*
       - Si 'x' es igual a "German", imprime: "Wie geht es dir?".
       */

    else -> print("I don't know that language yet :(")
    /*
       - El caso 'else' se ejecuta si 'x' no coincide con "English" ni "German".
       - Imprime: "I don't know that language yet :(".
       */
}
```

Documentación: Este código utiliza la expresión when en Kotlin para ejecutar diferentes bloques de código en función del valor de la variable x. En este caso, x representa un idioma, y el programa imprime un mensaje de saludo correspondiente a ese idioma. Si el idioma no está contemplado, se ejecuta un caso por defecto (else).

Ejemplo 31:

```
// Evaluar la variable 'x' utilizando 'when' y asignar un saludo a la variable 'greeting'
val greeting = when (x) {
    "English" -> "How are you?"
    /*
       - Si 'x' es igual a "English", se asigna el valor "How are you?" a 'greeting'.
    */

    "German" -> "Wie geht es dir?"
    /*
       - Si 'x' es igual a "German", se asigna el valor "Wie geht es dir?" a 'greeting'.
    */
```

```
    else -> "I don't know that language yet :("
    /*
       - Si 'x' no coincide con "English" ni con "German",
         se asigna el valor "I don't know that language yet :(" a 'greeting'.
    */
}

// Imprimir el saludo almacenado en 'greeting'
print(greeting)
/*
    - 'print': Función para mostrar datos en la consola.
    - Imprime el valor de la variable 'greeting'.
*/
```

Documentación: Este código utiliza una expresión when en Kotlin para asignar un valor a la variable greeting basado en el valor de la variable x. Posteriormente, imprime el saludo almacenado en greeting.

Ejemplo 32:

```
// Definir una enumeración para los días de la semana
enum class Day {
    Sunday,    // Representa el día domingo
    Monday,    // Representa el día lunes
    Tuesday,   // Representa el día martes
    Wednesday, // Representa el día miércoles
    Thursday,  // Representa el día jueves
    Friday,    // Representa el día viernes
    Saturday   // Representa el día sábado
```

```
}
/*
  - 'enum class': Define una enumeración que permite listar valores únicos y constantes.
  - Cada valor dentro de 'Day' es una instancia de la enumeración.
*/

// Función para realizar acciones basadas en el día
fun doOnDay(day: Day) {
  when(day) {
    Day.Sunday ->    // Ejecutar acción para el domingo
    /*
      - Este bloque se ejecutará si 'day' es igual a 'Day.Sunday'.
      - Especifica las acciones que deben realizarse el domingo.
    */

    Day.Monday, Day.Tuesday ->    // Ejecutar acción para lunes y martes
    /*
      - Este bloque se ejecutará si 'day' es igual a 'Day.Monday' o 'Day.Tuesday'.
      - Permite agrupar días que comparten la misma lógica.
    */

    Day.Wednesday ->  // Ejecutar acción para el miércoles
    Day.Thursday ->   // Ejecutar acción para el jueves
    Day.Friday ->    // Ejecutar acción para el viernes
    Day.Saturday ->   // Ejecutar acción para el sábado
```

```
    /*
      - Cada uno de estos bloques define las acciones para los días correspondientes.
      - Se pueden completar con lógica específica según las necesidades del programa.
    */
  }
}
```

Documentación: Este código define una enumeración llamada Day que representa los días de la semana, y una función doOnDay que utiliza una expresión when para ejecutar acciones específicas dependiendo del día proporcionado como argumento.

Ejemplo 33:

```
// Declarar una interfaz llamada 'Foo' con una función abstracta
interface Foo {
  fun example()
  /*
    - 'fun example()': Declaración de una función abstracta.
    - Las clases que implementan esta interfaz deben definir esta función.
  */
}

// Declarar una clase llamada 'Bar' que contiene una función 'example'
class Bar {
  fun example() {
    println("Hello, world!")
    /*
      - 'example': Imprime "Hello, world!" cuando se invoca.
```

```
    */
  }
}

// Declarar una clase llamada 'Baz' que implementa 'Foo' mediante delegación
class Baz(b: Bar) : Foo by b
/*
  - 'Foo by b': La implementación de la interfaz 'Foo' se delega a la instancia de la clase 'Bar' proporcionada.
  - La clase 'Baz' no necesita redefinir la función 'example', ya que la delega a 'Bar'.
*/

// Crear una instancia de 'Baz' pasando una instancia de 'Bar', y llamar a la función 'example'
Baz(Bar()).example()
/*
  - Se crea un objeto de la clase 'Baz', que delega la función 'example' al objeto de 'Bar'.
  - Cuando se invoca 'Baz(Bar()).example()', se ejecuta la implementación de 'example' en la clase 'Bar'.
  */
```

Documentación: Este código demuestra la delegación en Kotlin utilizando la palabra clave by. Implementa una interfaz (Foo) y delega su implementación a otra clase (Bar) a través de una clase delegada (Baz). Como resultado, la función de la clase delegada se invoca sin necesidad de definirla explícitamente en la clase delegada.

Ejemplo 34:

```
// Declarar una función de extensión infix para comparar valores
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

/*

- 'infix': Permite que la función sea llamada sin paréntesis ni punto, mejorando la legibilidad.

- '<T>': Declara que la función es genérica y puede trabajar con cualquier tipo 'T'.

- 'T?': Especifica que el tipo genérico 'T' puede ser nullable.

- 'expected: T?': Parámetro que representa el valor esperado en la comparación.

- 'this': Referencia al valor actual en el contexto donde se llama la función.

- 'assertEquals(expected, this)': Verifica que el valor actual ('this') es igual al valor esperado ('expected').

*/

Documentación: Este código define una función de extensión en Kotlin utilizando la palabra clave infix. La función llamada shouldBe es genérica y puede ser utilizada con cualquier tipo (T). Su propósito es comparar dos valores (el valor actual y el valor esperado) y verificar que son iguales, utilizando la función assertEquals.

Ejemplo 35:

```
// Declarar una clase llamada 'MyExample'

class MyExample(val i: Int) {

  /*

    - 'i': Es una propiedad inmutable (de tipo Int) que se inicializa al crear una instancia de la clase.

  */


  // Sobrecargar el operador 'invoke' para ejecutar bloques de código

  operator fun <R> invoke(block: MyExample.() -> R) = block()

  /*

    - 'operator': Permite sobrecargar operadores para casos personalizados.

    - '<R>': Define que el método es genérico y puede retornar cualquier tipo 'R'.

    - 'block: MyExample.() -> R':
```

- 'block' es un bloque de código que opera dentro del contexto de la instancia actual ('MyExample').

- Devuelve un resultado de tipo 'R'.

- 'block()': Ejecuta el bloque de código pasado como argumento.

*/

// Función de extensión para comparar un entero con la propiedad 'i'

fun Int.bigger() = this > i

/*

- 'fun Int.bigger()': Extiende la funcionalidad del tipo 'Int'.

- 'this': Hace referencia al valor de la instancia actual del tipo 'Int'.

- 'this > i': Devuelve true si el valor de 'this' es mayor que el valor de 'i' en la instancia actual de 'MyExample'.

*/

}

Documentación: La clase MyExample en Kotlin ilustra el uso de operadores personalizados (invoke), funciones genéricas, y extensiones para tipos específicos como Int. Este diseño permite crear instancias flexibles y ejecutar bloques de código con una sintaxis limpia y fluida.

Ejemplo 36:

// Crear una instancia de Random con una semilla fija

val r = Random(233)

/*

- 'Random(233)': Inicializa un generador de números pseudoaleatorios con una semilla.

- La semilla fija garantiza resultados reproducibles en las pruebas.

*/

```
// Declarar una función infix, inline, y operadora que extiende 'Int'
infix inline operator fun Int.rem(block: () -> Unit) {
  /*
    - 'infix': Permite usar la función como una operación entre el número entero y el bloque.
    - 'inline': Mejora el rendimiento al insertar el código del bloque en lugar de crear una llamada de función.
    - 'operator': Sobrecarga el operador `%` para un uso personalizado.
    */

  if (r.nextInt(100) < this) block()
  /*
    - 'r.nextInt(100)': Genera un número entero aleatorio entre 0 (incluido) y 100 (excluido).
    - 'this': Hace referencia al valor entero (contexto de la operación).
    - 'this < r.nextInt(100)': Evalúa si el valor actual (entero) es mayor que el número aleatorio.
    - Si la condición es verdadera, se ejecuta el bloque de código proporcionado.
    */
}
```

Documentación: El código presentado define una operación personalizada utilizando la sobrecarga del operador rem (%) para ejecutar un bloque de código condicionado por una probabilidad. El comportamiento se basa en un valor aleatorio generado con una semilla específica (233) para mantener resultados reproducibles.

Ejemplo 37:

```
// Sobrecargar el operador 'invoke' para la clase String
operator fun <R> String.invoke(block: () -> R) = {
  /*
```

- 'operator': Palabra clave que indica que el operador 'invoke' está siendo sobrecargado.

- '<R>': La función es genérica y puede devolver un resultado de cualquier tipo 'R'.

- 'block: () -> R': Define un bloque de código que retorna un valor de tipo 'R'.

*/

```
  try {
    block.invoke()
    /*
      - 'block.invoke()': Ejecuta el bloque de código proporcionado.
      - Si no ocurre ninguna excepción, el bloque se ejecuta normalmente.
    */
  } catch (e: AssertException) {
    System.err.println("$this\n${e.message}")
    /*
      - 'catch (e: AssertException)': Captura excepciones del tipo 'AssertException'.
      - 'System.err.println("$this\n${e.message}")':
        - Muestra un mensaje de error en la salida de error estándar (stderr).
        - Incluye la representación en cadena del objeto 'String' actual ('$this') y el mensaje de la excepción ('$e.message').
    */
  }
}
```

Documentación: Este código define una sobrecarga del operador invoke para la clase String en Kotlin. La implementación permite que un objeto de tipo String actúe como una función que recibe un bloque de código (block) y lo ejecuta dentro de un entorno con manejo de excepciones.

Ejemplo 38:

```
// Definir una enumeración llamada 'Color'
enum class Color(val rgb: Int) {
  /*
    - 'enum class': Define una enumeración en Kotlin.
    - Cada constante de la enumeración puede incluir propiedades y métodos personalizados.
    */

  RED(0xFF0000), // Rojo, con valor RGB hexadecimal: 0xFF0000
  /*
    - '0xFF0000': Representa el color rojo en formato RGB, donde:
      - FF (rojo)
      - 00 (verde)
      - 00 (azul)
  */

  GREEN(0x00FF00), // Verde, con valor RGB hexadecimal: 0x00FF00
  /*
    - '0x00FF00': Representa el color verde en formato RGB, donde:
      - 00 (rojo)
      - FF (verde)
      - 00 (azul)
  */

  BLUE(0x0000FF) // Azul, con valor RGB hexadecimal: 0x0000FF
  /*
```

- '0x0000FF': Representa el color azul en formato RGB, donde:

- 00 (rojo)

- 00 (verde)

- FF (azul)

*/

}

Documentación: Este código define una enumeración llamada Color en Kotlin que representa colores básicos utilizando valores RGB (Red-Green-Blue) codificados en hexadecimal. La enumeración incluye tres constantes: RED, GREEN y BLUE, cada una asociada a un valor RGB.

Ejemplo 39:

```
// Declarar una enumeración llamada 'Color'

enum class Color {

  RED {

    override val rgb: Int = 0xFF0000 // Rojo, con valor RGB: 0xFF0000

  },

  /*

    - 'override val rgb': Sobrescribe la propiedad abstracta 'rgb' y define el valor único para el color rojo.

  */


  GREEN {

    override val rgb: Int = 0x00FF00 // Verde, con valor RGB: 0x00FF00

  },

  /*

    - Define el valor único RGB para el color verde.

  */
```

```
    BLUE {
        override val rgb: Int = 0x0000FF // Azul, con valor RGB: 0x0000FF
    }
    /*
        - Define el valor único RGB para el color azul.
    */

    ; // Separador requerido entre las constantes y los miembros de la enumeración

    abstract val rgb: Int
    /*
        - Propiedad abstracta que debe ser sobrescrita por cada constante en la enumeración.
        - Representa el valor RGB asociado con cada color.
    */

    fun colorString() = "#%06X".format(0xFFFFFF and rgb)
    /*
        - 'colorString()': Método que genera una representación en formato hexadecimal del color.
        - 'format': Formatea el número RGB en una cadena hexadecimal con seis dígitos.
        - '0xFFFFFF and rgb': Asegura que solo se utilicen los componentes RGB válidos (ignora partes excedentes).
    */
}
```

Documentación: Esta implementación de la enumeración Color en Kotlin no solo define constantes relacionadas con colores básicos (RED, GREEN, BLUE), sino que también utiliza propiedades y métodos abstractos, con lógica personalizada para cada constante. Además, incorpora un método adicional (colorString) para generar una representación hexadecimal del color.

Ejemplo 40:

```
// Declarar una enumeración llamada 'Color'

enum class Color {

  RED,   // Representa el color rojo

  GREEN, // Representa el color verde

  BLUE   // Representa el color azul

}

/*

  - 'enum class': Palabra clave utilizada para definir una enumeración en Kotlin.

  - Cada constante dentro de la enumeración es única e inmutable.

  - 'RED', 'GREEN', 'BLUE': Valores que pertenecen a la enumeración 'Color'.

*/
```

Documentación: El código define una enumeración (enum class) llamada Color en Kotlin, utilizada para representar un conjunto fijo de valores: RED, GREEN y BLUE. Una enumeración es ideal para representar constantes relacionadas de manera clara y estructurada.

Ejemplo 41:

```
// Definir una enumeración llamada 'Planet' con una propiedad modificable 'population'

enum class Planet(var population: Int = 0) {

  EARTH(7 * 100000000), // Define el planeta Tierra con una población inicial

  /*

    - 'EARTH(7 * 100000000)': La población se inicializa como 700 millones (valor predeterminado).
```

```
    */


  MARS(); // Define el planeta Marte con población inicial predeterminada (0)

  /*

    - 'MARS()': Marte no tiene población inicial especificada, por lo que utiliza el valor predeterminado (0).

    */


  override fun toString() = "$name[population=$population]"

  /*

    - Sobrescribe el método 'toString' para que devuelva una representación personalizada del planeta.

    - '$name': Referencia al nombre del planeta (como 'EARTH' o 'MARS').

    - '[population=$population]': Muestra la población del planeta.

    */

}
```

Documentación: Este código muestra cómo usar enumeraciones (enum class) en Kotlin para representar planetas con propiedades modificables, como la población. Además, sobrescribe el método toString para mostrar información personalizada de cada planeta.

Ejemplo 42:

```
// Transformar una colección de objetos 'people' en una lista de nombres

val list = people.map { it.name }

/*

  - 'people': Representa una colección (lista, conjunto, etc.) de objetos que tienen una propiedad 'name'.

  - 'map': Función de transformación que aplica el bloque de código a cada elemento de la colección.
```

- 'it': Referencia al elemento actual en la iteración.

- 'it.name': Extrae la propiedad 'name' de cada elemento.

- Devuelve una nueva lista que contiene solo los nombres.

*/

Documentación: Este código en Kotlin utiliza la función de extensión map para transformar una colección (people) en una lista (list) que contiene únicamente los nombres (name) de los elementos de la colección original. No se necesita llamar a toList() porque map ya devuelve una nueva lista.

Ejemplo 43:

```
// Unir los elementos de la colección 'things' en una cadena de texto

val joined = things.joinToString()

/*

- 'things': Representa una colección (lista, conjunto, etc.) cuyos elementos serán combinados.

- 'joinToString()':

    - Combina los elementos de la colección en una única cadena.

    - Usa la coma y un espacio (", ") como separador predeterminado entre elementos.

    - Retorna una cadena de texto.

*/
```

Documentación: En este código de Kotlin, se utiliza la función joinToString para combinar los elementos de una colección (things) en una sola cadena de texto. Si no se especifica un separador, la función usa una coma y un espacio (", ") como separador predeterminado.

Ejemplo 44:

```
// Calcular la suma de los salarios de todos los empleados

val total = employees.sumBy { it.salary }

/*
```

- 'employees': Representa una colección de objetos (lista, conjunto, etc.) donde cada objeto tiene una propiedad 'salary'.

- 'sumBy': Función de extensión que realiza una suma basada en un criterio definido.

- 'it.salary': Obtiene el valor de la propiedad 'salary' de cada objeto en la colección.

- 'total': Contiene la suma total de los valores de 'salary' en la colección.

*/

Documentación: Este fragmento de código en Kotlin utiliza la función de extensión sumBy para calcular la suma de valores derivados de los elementos de una colección. En este caso, employees es una colección que contiene objetos con una propiedad salary, y la suma total de todos los salarios se almacena en la variable total.

Ejemplo 45:

```
// Agrupar empleados por su departamento

val byDept = employees.groupBy { it.department }

/*

- 'employees': Representa una colección de objetos (como lista o conjunto).

- 'groupBy': Función de extensión que organiza los elementos de una colección en un mapa.

- Cada elemento de la colección se agrupa según el valor retornado por la función lambda.

- 'it.department': Utiliza la propiedad 'department' de cada empleado como clave para la agrupación.

- 'byDept': Contendrá un mapa donde:

- Las claves son los nombres de los departamentos.

- Los valores son listas de empleados asociados a cada departamento.

*/
```

Documentación: Este fragmento de código utiliza la función de extensión groupBy en Kotlin para agrupar una colección de empleados (employees) según un atributo común: el departamento (department). El resultado es un mapa en el que las claves son los nombres de los departamentos y los valores son listas de empleados pertenecientes a cada departamento.

Ejemplo 46:

```
// Agrupar empleados por departamento y calcular el salario total por departamento

val totalByDept = employees

  .groupBy { it.dept }

  /*

    - 'groupBy { it.dept }': Agrupa los empleados en un mapa donde:

      - Las claves son los nombres de los departamentos (it.dept).

      - Los valores son listas de empleados asociados a cada departamento.

  */


  .mapValues { it.value.sumBy { it.salary } }

  /*

    - 'mapValues { ... }': Transforma los valores del mapa resultante del groupBy.

      - 'it.value': Se refiere a la lista de empleados en un departamento específico.

      - 'sumBy { it.salary }': Calcula la suma de los salarios de todos los empleados en esa lista.

  */
```

Documentación: Este código de Kotlin combina las funciones de extensión groupBy y mapValues para agrupar una colección de empleados (employees) por su departamento (dept) y calcular la suma total de los salarios (salary) dentro de cada grupo. El resultado es un mapa donde las claves son los nombres de los departamentos y los valores son las sumas totales de los salarios en cada departamento.

Ejemplo 47:

```
// Dividir a los estudiantes en dos listas: aprobados y reprobados
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
/*
  - 'students': Representa una colección (lista, conjunto, etc.) de estudiantes.
  - 'partition { it.grade >= PASS_THRESHOLD }':
    - Función de extensión que divide la colección en dos listas.
    - 'it.grade': Accede a la propiedad 'grade' de cada estudiante.
    - 'PASS_THRESHOLD': Representa el umbral mínimo para aprobar.
    - Los estudiantes que cumplen la condición ('grade >= PASS_THRESHOLD') se incluyen en la primera lista.
    - Los estudiantes que no cumplen la condición se incluyen en la segunda lista.
  - 'passingFailing': Contendrá un par (Pair) donde:
    - `first`: Lista de estudiantes que aprobaron.
    - `second`: Lista de estudiantes que reprobaron.
*/
```

Documentación: El código presentado utiliza la función de extensión partition de Kotlin para dividir una colección (students) en dos listas basadas en una condición. En este caso, los estudiantes se dividen en dos grupos: los que aprobaron y los que reprobaron, según el umbral de aprobación (PASS_THRESHOLD).

Ejemplo 48:

```
// Crear una lista con los nombres de los miembros masculinos del roster
val namesOfMaleMembers = roster
  .filter { it.gender == Person.Sex.MALE }
  /*
    - 'filter': Filtra los elementos de la colección según la condición especificada.
    - 'it.gender == Person.Sex.MALE': Selecciona solo aquellos elementos cuyo género sea 'MALE'.
  */
```

```
    .map { it.name }
    /*
        - 'map': Transforma cada elemento restante en un nuevo valor.
        - 'it.name': Extrae el valor de la propiedad 'name' de cada elemento.
        - Devuelve una nueva lista que contiene solo los nombres de los miembros masculinos.
    */
```

Documentación: Este código utiliza dos funciones de extensión, filter y map, para procesar una colección llamada roster y crear una lista con los nombres de los miembros masculinos. Estas funciones son parte de la biblioteca estándar de Kotlin y se usan comúnmente en el procesamiento de colecciones.

Ejemplo 49:

```
// Agrupar los nombres por género
val namesByGender = roster
    .groupBy { it.gender }
    /*
        - 'groupBy { it.gender }': Agrupa los elementos de la colección según la propiedad 'gender'.
        - Cada clave del mapa será un valor único de 'gender'.
        - Cada valor del mapa será una lista de objetos que tienen ese género.
    */


    .mapValues { it.value.map { it.name } }
    /*
        - 'mapValues { it.value.map { it.name } }':
          - Transforma las listas asociadas a cada género (los valores del mapa generado por groupBy).
```

- 'it.value': Accede a la lista de objetos en el grupo actual.

- '.map { it.name }': Extrae los nombres de los objetos en esa lista.

- Devuelve un mapa donde las claves son géneros y los valores son listas de nombres.

*/

Documentación: Este código utiliza las funciones de extensión groupBy y mapValues en Kotlin para agrupar una colección de objetos (roster) por género (gender) y luego transformar los grupos para obtener listas de nombres (name) dentro de cada grupo. El resultado es un mapa (namesByGender) con claves de tipo gender y valores que son listas de cadenas.

Ejemplo 50:

```
// Filtrar los elementos de la colección 'items' que comienzan con 'o'

val filtered = items.filter { item.startsWith('o') }

/*

  - 'items': Representa una colección (lista, conjunto, etc.) de cadenas u objetos similares.

  - 'filter': Función de extensión que recorre la colección y selecciona los elementos que cumplen con la condición especificada.

  - '{ item.startsWith('o') }': Bloque lambda que define la condición.

    - 'item': Referencia al elemento actual de la colección durante la iteración.

    - 'startsWith('o')': Verifica si el elemento comienza con la letra 'o'.

  - 'filtered': Contendrá una nueva colección con los elementos que cumplen la condición.

*/
```

Documentación: Este fragmento de código utiliza la función de extensión filter en Kotlin para crear una nueva colección (filtered) que contiene solo los elementos de items que cumplen con una condición específica: los que comienzan con la letra 'o'.

Ejemplo 51:

```
// Encontrar el elemento más corto en la colección 'items'
```

```
val shortest = items.minBy { it.length }

/*

  - 'items': Representa una colección (lista, conjunto, etc.) de cadenas u objetos similares.

  - 'minBy { it.length }': Busca el elemento con el valor mínimo basado en el criterio proporcionado.

     - 'it.length': Evalúa la longitud de cada elemento en la colección.

  - 'shortest': Contendrá el elemento más corto de la colección.

*/
```

Documentación: Este código de Kotlin utiliza la función de extensión minBy (deprecated en versiones recientes, ahora reemplazada por minByOrNull) para encontrar el elemento más corto en una colección (items) basado en una propiedad: la longitud del elemento (it.length).

Ejemplo 52:

```
// Crear una secuencia de cadenas

sequenceOf("a1", "a2", "a3")

/*

  - 'sequenceOf("a1", "a2", "a3")': Crea una secuencia de cadenas con los valores "a1", "a2", "a3".

  - Una secuencia es una colección lazily evaluada, lo que significa que los elementos se procesan a medida que se necesitan.

*/


// Obtener el primer elemento de la secuencia, si existe

.firstOrNull()

/*

  - 'firstOrNull()': Retorna el primer elemento de la secuencia.

  - Si la secuencia está vacía, retorna 'null' en lugar de lanzar una excepción.
```

*/

// Aplicar una acción al elemento encontrado, si no es nulo

?.apply(::println)

/*

- '?.apply { ... }': Ejecuta la acción proporcionada solo si el elemento no es nulo.

- '::println': Referencia a la función 'println', que imprime el elemento en la salida estándar.

*/

Documentación: Este código en Kotlin utiliza una combinación de funciones de extensión para trabajar con una secuencia de elementos (sequenceOf) y aplicar una acción (println) al primer elemento de la secuencia que no sea nulo.

Ejemplo 53:

(1..3).forEach(::println)

/*

- '(1..3)': Crea un rango que incluye los números del 1 al 3 (inclusive).

- '.forEach': Itera sobre cada elemento del rango y aplica la acción proporcionada.

- '::println': Es una referencia a la función 'println', que imprime cada elemento en la consola.

*/

Documentación: Este código en Kotlin utiliza el rango (1..3) junto con la función de extensión forEach para iterar sobre todos los números en el rango e imprimir cada uno de ellos. La sintaxis ::println es una referencia a la función println, lo que hace que cada número se imprima directamente.

Ejemplo 54:

// Crear un array con los elementos iniciales

arrayOf(1, 2, 3)

/*

- 'arrayOf(1, 2, 3)': Crea un arreglo que contiene los valores 1, 2 y 3.
- El arreglo puede ser usado para operaciones de transformación y cálculo.

*/

```
// Transformar cada elemento del array utilizando 'map'
.map { 2 * it + 1 }
```

/*

- 'map { 2 * it + 1 }': Aplica una transformación a cada elemento del array.
  - 'it': Referencia al elemento actual del array.
  - '2 * it + 1': Multiplica el elemento por 2 y suma 1.
  - Devuelve una nueva colección con los valores transformados.
  - Para el arreglo inicial [1, 2, 3], los resultados son [3, 5, 7].

*/

```
// Calcular el promedio de los valores resultantes
.average()
```

/*

- 'average()': Calcula el promedio de los valores transformados.
- Para [3, 5, 7], el promedio es (3 + 5 + 7) / 3 = 5.0.

*/

```
// Aplicar una acción al promedio utilizando 'apply'
.apply(::println)
```

/*

- '.apply { ... }': Aplica una acción al valor resultante (promedio).
- '::println': Referencia a la función 'println', que imprime el valor.

- En este caso, imprime '5.0' en la consola.

*/

Documentación: Este código utiliza una combinación de funciones de extensión de Kotlin (map, average, y apply) para transformar los elementos de un array, calcular el promedio de los valores resultantes, y aplicar una acción (println) para imprimir el resultado.

Ejemplo 55:

sequenceOf("a1", "a2", "a3")

/*

- 'sequenceOf("a1", "a2", "a3")': Crea una secuencia de cadenas con los valores "a1", "a2", "a3".

- Las secuencias se evalúan de forma lazy (diferida), lo que las hace eficientes para grandes colecciones.

*/

.map { it.substring(1) }

/*

- 'map { it.substring(1) }': Aplica una transformación a cada elemento de la secuencia.

  - 'it.substring(1)': Extrae la subcadena desde el índice 1 de cada elemento.

  - Para "a1", "a2", "a3", los resultados son "1", "2", "3".

*/

.map(String::toInt)

/*

- 'map(String::toInt)': Convierte cada cadena resultante ("1", "2", "3") en un entero.

  - 'String::toInt': Referencia directa al método 'toInt' de la clase String.

  - Los resultados ahora son 1, 2, 3.

```
*/


.max()

/*
  - 'max()': Encuentra el valor máximo dentro de la secuencia transformada (1, 2, 3).
    - En este caso, devuelve el valor 3.
*/


.apply(::println)

/*
  - '.apply { ... }': Aplica una acción al resultado del paso anterior, si no es nulo.
  - '::println': Es una referencia directa a la función 'println', que imprime el valor en la consola.
  - Imprime 3 en este ejemplo.
*/
```

Documentación: Este código utiliza una secuencia en Kotlin (sequenceOf) y varias funciones de extensión encadenadas (map, max, y apply) para transformar los elementos de la secuencia, encontrar el valor máximo y luego imprimirlo utilizando println.

Ejemplo 56:

```
(1..3)
  .map { "a$it" }
  /*
    - '(1..3)': Crea un rango de números del 1 al 3 (inclusive).
    - 'map { "a$it" }': Aplica una transformación a cada número del rango.
      - 'it': Referencia al número actual en el rango durante la iteración.
      - '"a$it"': Construye una cadena utilizando el número, con el formato "aX".
```

- Resultado: ["a1", "a2", "a3"].

*/

.forEach(::println)

/*

- '.forEach { ... }': Aplica una acción a cada elemento de la colección transformada.

- '::println': Referencia directa a la función 'println', que imprime cada elemento en la consola.

- Imprime las cadenas transformadas una por una.

*/

Documentación: Este código utiliza Kotlin para crear un rango de números (1..3), transformar cada número en una cadena personalizada con el formato "aX" (map), y luego imprimir cada cadena resultante (forEach) utilizando una referencia a la función println. Este enfoque combina transformación y acción en un flujo funcional.

Ejemplo 57:

sequenceOf(1.0, 2.0, 3.0)

/*

- 'sequenceOf(1.0, 2.0, 3.0)': Crea una secuencia con los valores 1.0, 2.0 y 3.0.

- Las secuencias son colecciones evaluadas de forma diferida (lazy), útiles para grandes conjuntos de datos.

*/

.map(Double::toInt)

/*

- 'map(Double::toInt)': Convierte cada valor de tipo 'Double' en un valor de tipo 'Int'.

- 'Double::toInt': Referencia a la función 'toInt' de la clase 'Double', que trunca el decimal.

- Para 1.0, 2.0, 3.0, el resultado es 1, 2, 3.

```
*/


.map { "a$it" }

/*

   - 'map { "a$it" }': Transforma cada entero en una cadena personalizada.

   - '"a$it"': Combina la letra 'a' con el número actual.

   - Resultado: ["a1", "a2", "a3"].

*/


.forEach(::println)

/*

   - '.forEach { ... }': Recorre la secuencia transformada y aplica la acción proporcionada.

   - '::println': Es una referencia directa a la función 'println', que imprime cada elemento.

   - Imprime cada cadena en la consola.

*/
```

Documentación: Este código utiliza una secuencia de valores (sequenceOf) y aplica varias transformaciones encadenadas mediante funciones de extensión como map y forEach. Cada elemento en la secuencia se convierte de un número decimal (Double) a un entero (Int), se transforma en una cadena personalizada, y finalmente, se imprime en la consola.

Ejemplo 58:

```
// Contar los elementos de 'items' que comienzan con la letra 't'

val count = items.filter { it.startsWith('t') }.size

/*

   - 'items': Representa una colección (lista, conjunto, etc.) de cadenas u objetos similares.
```

- 'filter { it.startsWith('t') }':

  - Filtra los elementos de la colección según una condición.

  - 'it.startsWith('t')': Verifica si el elemento actual comienza con la letra 't'.

  - Devuelve una nueva colección con los elementos que cumplen la condición.

- '.size': Obtiene el número de elementos en la colección filtrada.

- 'count': Almacena el número total de elementos que cumplen la condición.

*/

Documentación: Este fragmento de código utiliza las funciones de extensión filter y size de Kotlin para contar los elementos en la colección items que comienzan con la letra 't'.

Ejemplo 59:

```
// Crear una lista inicial de cadenas

val list = listOf("a1", "a2", "b1", "c2", "c1")

/*

- 'listOf("a1", "a2", "b1", "c2", "c1")': Inicializa una lista con elementos específicos.

- Contiene cadenas con un formato "letra + número".

*/


// Filtrar los elementos que comienzan con 'c'

list.filter { it.startsWith('c') }

/*

- 'filter { it.startsWith('c') }': Selecciona los elementos que cumplen con la condición.

  - 'it.startsWith('c')': Verifica si el elemento comienza con la letra 'c'.

- Resultado: ["c2", "c1"].

*/
```

```
// Transformar los elementos filtrados a mayúsculas
.map(String::toUpperCase)
/*
  - 'map(String::toUpperCase)': Convierte cada cadena de la lista filtrada a mayúsculas.
     - 'String::toUpperCase': Referencia directa al método 'toUpperCase'.
  - Resultado: ["C2", "C1"].
*/

// Ordenar los elementos transformados
.sorted()
/*
  - 'sorted()': Ordena los elementos en orden alfabético.
     - Resultado: ["C1", "C2"].
*/

// Imprimir los elementos de la lista ordenada
.forEach(::println)
/*
  - 'forEach(::println)': Itera sobre los elementos y aplica la función 'println' a cada uno.
  - Imprime cada cadena en una línea de la consola.
*/
```

Documentación: Este fragmento de código en Kotlin trabaja con una lista inicial (list) para realizar una serie de operaciones encadenadas: filtrar, transformar, ordenar, e iterar sobre los elementos. Al final, imprime los resultados en la consola.

Ejemplo 60:

listOf("a1", "a2", "a3")

/*

- 'listOf("a1", "a2", "a3")': Crea una lista inmutable que contiene las cadenas "a1", "a2" y "a3".

*/

// Obtener el primer elemento de la lista o `null` si está vacía

.firstOrNull()

/*

- 'firstOrNull()': Devuelve el primer elemento de la lista.

- Si la lista está vacía, retorna `null` en lugar de lanzar una excepción.

*/

// Aplicar una acción al elemento encontrado, si no es `null`

?.apply(::println)

/*

- '?.apply { ... }': Ejecuta el bloque proporcionado solo si el elemento no es `null`.

- '::println': Es una referencia directa a la función 'println', que imprime el valor en la consola.

*/

Documentación: Este código utiliza una lista en Kotlin (listOf) y una serie de funciones de extensión (firstOrNull y apply) para encontrar el primer elemento de la lista y aplicar una acción a ese elemento si no es null.

Ejemplo 61:

val phrase = persons

.filter { it.age >= 18 }

/*

- 'filter { it.age >= 18 }': Filtra la lista original (`persons`) para incluir solo aquellos elementos donde la edad (`age`) es mayor o igual a 18.

- 'it': Hace referencia al elemento actual durante la iteración.

*/

.map { it.name }

/*

- 'map { it.name }': Transforma la lista filtrada en una lista de nombres (`name`) de las personas que cumplen con la condición.

*/

.joinToString(" and ", "In Germany ", " are of legal age.")

/*

- 'joinToString(" and ", "In Germany ", " are of legal age.")':

- Concatena los nombres resultantes en una única cadena, separados por " and ".

- Agrega un prefijo (`"In Germany "`) al inicio de la cadena.

- Agrega un sufijo (`" are of legal age."`) al final de la cadena.

*/

Documentación: Este fragmento de código en Kotlin filtra una lista de personas según su edad, transforma los datos en una lista de nombres, y luego construye una frase concatenada con un formato específico. Finalmente, imprime el resultado en la consola.

Ejemplo 62:

val map6 = persons

.groupBy { it.age }

/*

- 'groupBy { it.age }': Agrupa a las personas en un mapa según su edad ('age').

- Las claves del mapa son las edades únicas en la colección.

- Los valores son listas de personas que tienen esa edad.

*/

.mapValues { it.value.joinToString(";") { it.name } }

/*

- 'mapValues { ... }': Transforma los valores del mapa (listas de personas) en cadenas concatenadas.

- 'it.value': Accede a la lista de personas con una edad específica.

- 'joinToString(";") { it.name }': Combina los nombres de las personas en esa lista en una sola cadena.

- Los nombres están separados por un punto y coma (';').

*/

Documentación: Este fragmento de código en Kotlin agrupa una colección de personas (persons) por edad (age) y, para cada grupo, concatena los nombres de las personas separándolos con un punto y coma (;). Finalmente, se imprime el mapa resultante en la consola.

Ejemplo 63:

val names = persons

.map { it.name.toUpperCase() }

/*

- 'map { it.name.toUpperCase() }': Transforma cada elemento de la colección.

- 'it.name': Accede al nombre (`name`) de cada objeto en `persons`.

- 'toUpperCase()': Convierte el nombre a mayúsculas.

- Resultado: Una lista de nombres en mayúsculas.

*/

.joinToString(" | ")

```
  /*
     - 'joinToString(" | ")': Combina los nombres transformados en una sola cadena.
      - '" | "': Define el separador entre los nombres concatenados.
  */
```

Documentación: Este código utiliza las funciones de extensión map y joinToString de Kotlin para crear una cadena que combina los nombres en mayúsculas (toUpperCase) de una colección de objetos (persons), separados por " | ".

Ejemplo 64:

```
inline fun Collection<Int>.summarizingInt(): SummaryStatisticsInt =
  this.fold(SummaryStatisticsInt()) { stats, num -> stats.accumulate(num) }
/*
  - 'inline': Indica que la función será inyectada directamente en los lugares donde se llame, optimizando la ejecución.
  - 'Collection<Int>': Función de extensión para colecciones de enteros.
  - 'summarizingInt()': Calcula estadísticas resumen de los enteros en la colección.
  - 'fold': Realiza una reducción acumulativa sobre la colección.
     - 'SummaryStatisticsInt()': Inicializa un objeto para almacenar estadísticas.
     - 'stats.accumulate(num)': Acumula el valor actual en las estadísticas.
*/


inline fun <T: Any> Collection<T>.summarizingInt(transform: (T) -> Int): SummaryStatisticsInt =
  this.fold(SummaryStatisticsInt()) { stats, item -> stats.accumulate(transform(item)) }
/*
  - 'inline': Optimiza la ejecución al evitar la creación de funciones de alto orden en tiempo de ejecución.
  - '<T: Any>': Genera una función genérica para colecciones de cualquier tipo (`T`).
```

- 'summarizingInt(transform: (T) -> Int)': Calcula estadísticas resumen transformando cada elemento de la colección en un entero.

- 'transform: (T) -> Int': Recibe una función lambda para transformar cada elemento en un entero.

- 'fold': Reduce la colección en un objeto acumulador.

- 'SummaryStatisticsInt()': Inicializa las estadísticas.

- 'stats.accumulate(transform(item))': Aplica la transformación y acumula el resultado en las estadísticas.

*/

Documentación: Este fragmento de código define dos funciones de extensión en Kotlin que calculan estadísticas resumen de una colección, específicamente números enteros (Int). Estas funciones utilizan una clase llamada SummaryStatisticsInt, que se supone contiene métodos y propiedades para acumular valores y generar estadísticas.

Ejemplo 65:

```
try {
  doSomething() // Intentar ejecutar la operación principal
  /*
    - 'doSomething()': Representa una operación que podría arrojar una excepción.
    - Es el bloque principal donde ocurre la lógica que se espera completar exitosamente.
  */
}
catch (e: MyException) {
  handle(e) // Manejar la excepción de tipo específico
  /*
    - 'catch (e: MyException)': Captura excepciones del tipo 'MyException'.
    - 'handle(e)': Define cómo tratar la excepción capturada, por ejemplo, mostrando un mensaje o registrando un error.
```

```
  */
}
finally {
  cleanup() // Ejecutar tareas de limpieza
  /*
    - 'finally': Este bloque se ejecuta siempre, independientemente de si ocurrió una excepción o no.
    - 'cleanup()': Representa las operaciones de limpieza, como liberar recursos o cerrar conexiones.
  */
}
```

Documentación: Este fragmento de código en Kotlin utiliza un bloque try-catch-finally para manejar excepciones, asegurando que se ejecuten las operaciones necesarias, incluso si ocurre un error. Este enfoque es ideal para manejar errores y garantizar que el código sea robusto y limpio.

Ejemplo 66:

```
buildscript {
  ...
  /*
    - El bloque 'buildscript' define configuraciones específicas necesarias para el proyecto.
    - Incluye la definición de repositorios y dependencias necesarias para ejecutar los complementos del proyecto.
    - Por ejemplo, podría incluir configuraciones como:
      repositories { google(); mavenCentral() }
      dependencies { classpath 'com.android.tools.build:gradle:X.X.X' }
  */
}
```

```
apply plugin: "com.android.application"

/*
   - 'apply plugin: "com.android.application"':
      - Aplica el complemento que configura el proyecto como una aplicación de Android.
      - Este complemento habilita características específicas para crear y empaquetar una aplicación móvil.
*/

apply plugin: "kotlin-android"

/*
   - 'apply plugin: "kotlin-android"':
      - Aplica el complemento para integrar Kotlin en el proyecto Android.
      - Permite usar Kotlin como lenguaje de programación principal.
*/

apply plugin: "kotlin-android-extensions"

/*
   - 'apply plugin: "kotlin-android-extensions"':
      - Habilita características adicionales en Kotlin, como el acceso directo a vistas usando las extensiones de sintaxis de Android.
      - Nota: Este complemento está deprecado desde Kotlin 1.4.20, y se recomienda usar ViewBinding o Jetpack Compose.
*/
```

Documentación: Este fragmento de código pertenece a un archivo de configuración en Gradle, comúnmente usado en proyectos de Android. Combina configuraciones

del bloque buildscript y la aplicación de varios complementos (plugins) necesarios para configurar y compilar un proyecto en Kotlin para Android.

Ejemplo 67:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

  android:layout_width="match_parent"

  android:layout_height="match_parent">

  /*

    - '<?xml version="1.0" encoding="utf-8"?>': Define la versión XML y el tipo de codificación utilizado.

    - '<LinearLayout>': Contenedor principal que organiza los elementos en una disposición lineal (vertical u horizontal).

    - 'xmlns:android="..."': Define el espacio de nombres para los atributos de Android.

    - 'android:layout_width="match_parent"': Hace que el contenedor ocupe todo el ancho de la pantalla.

    - 'android:layout_height="match_parent"': Hace que el contenedor ocupe todo el alto de la pantalla.

  */


  <Button

    android:id="@+id/my_button"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="My button"/>

  /*

    - '<Button>': Define un botón interactivo en la interfaz.

    - 'android:id="@+id/my_button"': Asigna un identificador único al botón, usado para referenciarlo en el código.
```

- 'android:layout_width="wrap_content"': Ajusta el ancho del botón según su contenido.

- 'android:layout_height="wrap_content"': Ajusta la altura del botón según su contenido.

- 'android:text="My button"': Define el texto que se muestra en el botón.

*/

</LinearLayout>

Documentación: Este código XML define la interfaz de usuario de un diseño simple en Android utilizando un LinearLayout que contiene un Button. El diseño está configurado para ocupar toda la pantalla y tiene un botón con propiedades básicas.

Ejemplo 68:

```
android {
  productFlavors {
    paid {
      ...
      /*
        - 'paid': Define una variante del producto llamada "paid" (versión de pago).
        - Dentro de este bloque, puedes especificar configuraciones específicas para la variante.
        - Ejemplos: diferente `applicationId`, recursos exclusivos, o ajustes específicos.
      */
    }
    free {
      ...
      /*
        - 'free': Define una variante del producto llamada "free" (versión gratuita).
        - Similar al bloque 'paid', permite personalizar esta variante.
```

```
            - Ejemplo: activar anuncios, recursos distintos, o un identificador único.
        */
    }
  }
}
```

Documentación: Este código forma parte de la configuración de Gradle en un proyecto de Android. Utiliza el bloque productFlavors dentro de android para definir variantes de producto (product flavors), lo que permite crear diferentes versiones de una misma aplicación a partir del mismo código base. En este caso, se definen dos variantes: paid y free.

Ejemplo 69:

```
mView.afterMeasured {
  // Bloque de código que se ejecuta cuando la vista está completamente medida
  // 'it': Hace referencia a la vista actual (mView)
  // 'it.height': Obtiene la altura final de la vista
  // 'it.width': Obtiene el ancho final de la vista

  // Aquí puedes realizar operaciones que dependan de las dimensiones finales
}
```

Documentación: Este fragmento utiliza la función afterMeasured en Kotlin, que parece ser un método de extensión para una vista (mView) en Android. Permite ejecutar un bloque de código después de que la vista haya sido completamente medida y dibujada. Esto es útil para operaciones que requieren conocer las dimensiones finales de la vista, como su altura (it.height) y ancho (it.width).

Ejemplo 70:

```
fun twice(x: () -> Any?) {
  x() // Primera ejecución de la lambda pasada como argumento
  x() // Segunda ejecución de la misma lambda
}
```

```
/*
  - 'twice': Función que toma como argumento una lambda que no tiene parámetros y devuelve cualquier tipo (`Any?`).
  - 'x()': Invoca la lambda proporcionada. Esta llamada ocurre dos veces dentro de la función.
  - Uso: Permite ejecutar cualquier operación definida en la lambda dos veces.
*/

fun main() {
  twice {
    println("Foo") // La lambda imprime "Foo"
  }
  /*
    - 'twice { println("Foo") }': Llama a la función 'twice' y le pasa una lambda que imprime "Foo".
    - Resultado: La función 'println("Foo")' se ejecuta dos veces, generando dos líneas con "Foo".
  */
}
```

Documentación: Este fragmento en Kotlin muestra cómo se utiliza una función llamada twice para ejecutar un bloque de código dos veces. La función twice toma una lambda como parámetro (x: () -> Any?) y la invoca dos veces. En el programa principal (main), se pasa una lambda que imprime "Foo", lo que genera el resultado esperado.

Ejemplo 71:

```
{ name: String ->
  "Your name is $name"
}
```

Documentación: El fragmento presentado define una **lambda** en Kotlin. Este tipo de función anónima toma un argumento llamado name (de tipo String) y retorna una cadena que incluye el valor del argumento interpolado dentro del texto "Your name is $name".

Ejemplo 72:

```
fun addTwo(x: Int) = x + 2

/*
   - 'addTwo(x: Int)': Define una función que toma un entero como entrada.
   - 'x + 2': Devuelve el resultado de sumar 2 al número proporcionado.
   - Es una función simple y efectiva para realizar cálculos repetidos.
*/


listOf(1, 2, 3, 4).map(::addTwo)

/*
   - 'listOf(1, 2, 3, 4)': Crea una lista con los elementos 1, 2, 3, y 4.
   - '.map(::addTwo)': Aplica la función 'addTwo' a cada elemento de la lista.
      - '::addTwo': Referencia directa a la función 'addTwo'.
      - 'map': Itera sobre cada elemento de la lista, aplica la función y genera una nueva lista con los valores transformados.
   */
```

Documentación: Este fragmento de código en Kotlin define una función llamada addTwo que incrementa un número entero en 2 y luego utiliza la función de extensión map para aplicar dicha función a cada elemento de una lista.

Ejemplo 73:

Documentación:

Ejemplo 74:

Documentación:

Ejemplo 75:

Documentación:

Ejemplo 76:

Documentación:

Ejemplo 77:

Documentación:

Ejemplo 78:

Documentación:

Ejemplo 79:

Documentación:

Ejemplo 80:

Documentación:

Ejemplo 81:

Documentación:

Ejemplo 82:

Documentación:

Ejemplo 83:

Documentación:

Ejemplo 84:

Documentación:

Ejemplo 85:

Documentación:

Ejemplo 86:

Documentación:

Ejemplo 87:

Documentación:

Ejemplo 88:

Documentación:

Ejemplo 89:

Documentación:

Ejemplo 90:

Documentación: