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:

```
fun sayMyName(name: String): String {
   /*
      - 'fun sayMyName(name: String)': Define una función llamada 'sayMyName' que toma un parámetro:
         - 'name: String': Parámetro de tipo String que representa el nombre que se procesará.
   */
   return "Your name is $name"
   /*
      - 'return "Your name is $name"': Devuelve una cadena interpolada que incluye el valor del parámetro 'name'.
      - La interpolación de cadenas permite insertar valores directamente dentro del texto.
   */
}
```

Documentación: Esta función en Kotlin recibe un nombre como parámetro de tipo String y retorna un mensaje formateado que incluye dicho nombre. Es útil para personalizar mensajes.

Ejemplo 74:

```
fun sayMyName(name: String): String = "Your name is $name"
/*
   - 'fun sayMyName(name: String)': Define una función llamada 'sayMyName' que toma un parámetro:
      - 'name: String': Parámetro de tipo String que representa el nombre proporcionado.
   - 'String = "Your name is $name"':
      - Retorna una cadena interpolada que incluye el valor del parámetro 'name'.
      - La sintaxis compacta utiliza '=' para simplificar el cuerpo de la función.
*/
```

Documentación: Esta función compacta en Kotlin devuelve un mensaje personalizado utilizando interpolación de cadenas. Utiliza la sintaxis de una expresión única para simplificar el código.

Ejemplo 75:

```
inline fun sayMyName(name: String) = "Your name is $name"

/*
   - 'inline': Indica que la función se reemplazará directamente en el lugar donde se llame, evitando la sobrecarga de una llamada de función.
      - Esto es útil para funciones pequeñas y frecuentemente invocadas.
   - 'fun sayMyName(name: String)': Define una función llamada 'sayMyName' que toma:
      - 'name: String': Un parámetro de tipo String que representa el nombre proporcionado.
   - 'String = "Your name is $name"':
      - Retorna una cadena interpolada que incluye el valor del parámetro 'name'.
      - La sintaxis compacta utiliza '=' para simplificar el cuerpo de la función.
*/
```

Documentación: Esta función de una sola línea en Kotlin utiliza la palabra clave inline para optimizar llamadas repetidas, combinada con la sintaxis compacta para devolver una cadena interpolada que incluye un parámetro name.

Ejemplo 76:

```
data class IntListWrapper(val wrapped: List<Int>) {
   /*
      - 'data class': Declara una clase de datos, que automáticamente genera métodos como 'toString', 'equals', y 'hashCode'.
      - 'val wrapped: List<Int>': Propiedad inmutable que contiene una lista de enteros.
   */

   operator fun get(position: Int): Int = wrapped[position]
   /*
      - 'operator fun get(position: Int)': Sobrecarga el operador de indexación ('[]').
```

- 'position: Int': Parámetro que representa el índice del elemento que se desea acceder.

- 'wrapped[position]': Devuelve el elemento en la posición especificada dentro de la lista envuelta.

*/

}

```
val a = IntListWrapper(listOf(1, 2, 3))

/*

- 'IntListWrapper(listOf(1, 2, 3))': Crea una instancia de 'IntListWrapper', envolviendo la lista de enteros [1, 2, 3].

- 'a': Variable que almacena la instancia de 'IntListWrapper'.

*/


a[1] // == 2

/*

- 'a[1]': Utiliza la sobrecarga del operador 'get' para acceder al segundo elemento de la lista (índice 1).

- Devuelve el valor 2.

*/
```

Documentación: Este código crea una clase de datos en Kotlin que envuelve una lista de enteros (Int) y sobrecarga el operador de indexación ([]) para acceder a los elementos de la lista contenida.

Ejemplo 77:

```
fun printHello(name: String?): Unit {

/*

- 'fun printHello(name: String?): Unit': Declara una función llamada 'printHello' que toma:

- 'name: String?': Un parámetro que puede ser nulo (`nullable`).

- 'Unit': Retorno de tipo vacío (opcional en Kotlin; `Unit` puede omitirse).
```

```
    */

    if (name != null)
        println("Hello ${name}")
    /*
        - 'if (name != null)': Verifica que el parámetro 'name' no sea nulo antes de usarlo.
        - 'println("Hello ${name}")': Imprime un saludo que incluye el valor del parámetro 'name'.
          - Utiliza interpolación de cadenas para insertar el valor directamente en el texto.
    */
}

fun printHello(name: String?) {
    ...
    /*
        - Segunda función con el mismo nombre 'printHello(name: String?)'.
        - Este es un lugar reservado para una implementación futura o una sobrecarga de la función.
        - Actualmente no realiza ninguna acción.
    */
}
```

Documentación: Este fragmento define dos funciones llamadas printHello en Kotlin. La primera está completa y muestra cómo manejar parámetros opcionales (nullable) para imprimir un saludo. La segunda parece estar incompleta o destinada a ser implementada más adelante.

Ejemplo 78:

```
for (i in 1..4) print(i)
/*
    - 'for (i in 1..4)': Itera desde el número 1 hasta el número 4 (inclusive).
```

- '1..4': Representa un rango de números en orden ascendente.

- 'i': Variable que contiene el valor actual en cada iteración.

- 'print(i)': Imprime el valor de 'i' en cada iteración sin saltos de línea.

- Resultado: "1234" (imprime todos los valores del rango consecutivamente).

*/

```
for (i in 4..1) print(i)
```

/*

- 'for (i in 4..1)': Intenta iterar desde el número 4 hasta el número 1 (orden descendente).

- '4..1': Representa un rango de números, pero en orden ascendente, por lo que está vacío.

- Como el rango no contiene números válidos, no se ejecuta ninguna iteración.

- Resultado: Nada es impreso.

*/

Documentación: Estos fragmentos de código utilizan bucles for en Kotlin para iterar sobre un rango de valores. La diferencia en el orden del rango afecta la salida producida por cada bucle.

Ejemplo 79:

```
for (i in 4 downTo 1) print(i)
```

/*

- 'for (i in 4 downTo 1)': Itera desde el número 4 hasta el número 1 (inclusive) en orden descendente.

- '4 downTo 1': Crea un rango que comienza en 4 y decrece hasta 1.

- 'i': Variable que contiene el valor actual en cada iteración.

- 'print(i)': Imprime el valor de 'i' en cada iteración sin saltos de línea.

- Resultado: "4321" (imprime todos los valores del rango en orden decreciente).

*/

Documentación: Este fragmento de código utiliza un bucle for en Kotlin con un rango descendente (downTo) para iterar sobre números en orden decreciente y luego imprimirlos.

Ejemplo 80:

```
for (i in 1..4 step 2) print(i)

/*
   - 'for (i in 1..4 step 2)': Itera desde el número 1 hasta el número 4 (inclusive) en pasos de 2.
      - '1..4': Representa un rango de números en orden ascendente.
      - 'step 2': Salta un número en cada iteración (incremento de 2).
      - 'i': Variable que contiene el valor actual en cada iteración.
   - 'print(i)': Imprime el valor de 'i' en cada iteración sin saltos de línea.
   - Resultado: "13" (imprime el primer y tercer valor del rango, omitiendo los demás).
*/


for (i in 4 downTo 1 step 2) print(i)

/*
   - 'for (i in 4 downTo 1 step 2)': Itera desde el número 4 hasta el número 1 (inclusive) en pasos de 2.
      - '4 downTo 1': Crea un rango descendente que comienza en 4 y decrece hasta 1.
      - 'step 2': Salta un número en cada iteración (decremento de 2).
      - 'i': Variable que contiene el valor actual en cada iteración.
   - 'print(i)': Imprime el valor de 'i' en cada iteración sin saltos de línea.
   - Resultado: "42" (imprime el primer y tercer valor del rango en orden descendente).
*/
```

Documentación: Estos fragmentos de código utilizan bucles for en Kotlin con el modificador step para controlar el intervalo entre iteraciones. Esto permite saltar valores mientras se recorre un rango.

Ejemplo 81:

```
for (i in 1 until 10) {
   /*
```

- 'for (i in 1 until 10)': Itera desde el número 1 hasta el número 9 (el número 10 no está incluido).

- '1 until 10': Crea un rango que comienza en 1 y termina antes de 10.

- 'i': Variable que contiene el valor actual en cada iteración del rango.

*/

println(i)

/*

- 'println(i)': Imprime el valor actual de 'i' en cada iteración, seguido de un salto de línea.

- Cada número del rango [1, 10) será impreso en una línea separada.

*/

}

Documentación: Este fragmento de código utiliza un bucle for en Kotlin con la palabra clave until para iterar sobre un rango de números donde el límite superior no está incluido en las iteraciones.

Ejemplo 82:

class Consumer<in T> {

/*

- 'class Consumer<in T>': Define una clase genérica con un parámetro contravariante `T`.

- `in T`: Contravarianza; asegura que la clase solo pueda aceptar objetos de tipo `T` o sus subtipos.

- La contravarianza es útil para situaciones donde el parámetro genérico se usa como entrada (no como salida).

*/

fun consume(t: T) { ... }

/*

- 'fun consume(t: T)': Método que recibe un parámetro de tipo `T`.

- Esto permite que la clase utilice el tipo genérico como entrada en sus operaciones.

*/

}

fun charSequencesConsumer(): Consumer<CharSequence> = ...

/*

- 'fun charSequencesConsumer()': Devuelve un objeto de tipo 'Consumer<CharSequence>'.

- Es una fábrica o inicializador para instancias específicas de `Consumer`.

*/

val stringConsumer: Consumer<String> = charSequencesConsumer()

/*

- 'val stringConsumer': Una instancia de 'Consumer<String>'.

- Asignación permitida porque el modificador `in` permite que un `Consumer<CharSequence>` actúe como `Consumer<String>` (proyección contravariante).

*/

val anyConsumer: Consumer<Any> = charSequencesConsumer()

/*

- 'val anyConsumer': Una instancia de 'Consumer<Any>'.

- Error: La contravarianza no permite que un `Consumer<CharSequence>` sea tratado como `Consumer<Any>`, ya que `Any` es un supertipo de `CharSequence`.

- Esto violaría la seguridad de tipos en tiempo de ejecución.

*/

val outConsumer: Consumer<out CharSequence> = ...

/*

- 'val outConsumer': Intenta declarar un `Consumer` con proyección covariante (`out CharSequence`).

- Error: No es posible porque `T` está marcado como `in` en la clase `Consumer`, lo que significa que solo puede ser contravariante.

- La proyección covariante (`out`) contradice la naturaleza de `in` en este caso.

*/

Documentación: Este fragmento utiliza la proyección de tipos genéricos en Kotlin, específicamente con el modificador in. La clase Consumer acepta tipos genéricos contravariantes (in T), lo que permite que un objeto de tipo Consumer pueda consumir un subtipo de T. Esto tiene implicaciones en cómo se definen y utilizan las instancias.

Ejemplo 83:

```
val takeList: MutableList<out SomeType> = ...

/*

- 'MutableList<out SomeType>': Declara una lista mutable con una proyección covariante (`out`).

    - 'out SomeType': Permite que la lista contenga objetos de tipo `SomeType` o cualquier subtipo de `SomeType`.

    - Este diseño restringe las operaciones que podrían violar la seguridad de tipos.

- En Java, esto equivale a 'List<? extends SomeType>'.

*/


val takenValue: SomeType = takeList[0]

/*

- 'takeList[0]': Accede al primer elemento de la lista.

- 'val takenValue: SomeType': Almacena el valor leído de la lista en una variable de tipo `SomeType`.

    - Esto es válido porque la proyección `out` asegura que todos los elementos son de tipo `SomeType` o un subtipo de este.

*/
```

```
takeList.add(takenValue) // Error, lower bound for generic is not specified

/*

   - 'takeList.add(takenValue)': Intenta añadir un elemento a la lista.

      - Error: La proyección `out` no especifica un límite inferior para el tipo genérico.

      - Debido a esto, Kotlin no puede garantizar que el tipo del elemento añadido sea seguro para todos los subtipos posibles de `SomeType`.

*/
```

Documentación: Este fragmento de código utiliza una lista mutable (MutableList) con una proyección de tipo genérico (out SomeType) en Kotlin, lo que es equivalente a List en Java. La proyección solo permite leer elementos, pero no añadirlos, debido a la naturaleza de la restricción de tipo out.

Ejemplo 84:

```
open class Thing {

   /*

      - 'open': Permite que la clase sea heredada por otras clases.

      - 'class Thing': Define una clase llamada 'Thing'.

      - Comentario: "I can now be extended!": Indica que esta clase está lista para ser utilizada como base de herencia.

   */

}
```

Documentación: Esta declaración define una clase abierta (open class) en Kotlin, lo que permite que otras clases hereden de ella. Por defecto, las clases en Kotlin son final (no heredables), pero usar la palabra clave open habilita la herencia.

Ejemplo 85:

```
open class BaseClass {

   /*

      - 'open class': Declara una clase que puede ser extendida por otras clases.

      - 'BaseClass': Nombre de la clase base.

   */
```

```
    val x = 10
    /*
        - 'val x = 10': Define una propiedad inmutable llamada 'x' con un valor inicial de 10.
        - Las propiedades declaradas con 'val' son de solo lectura (no se pueden modificar después de la inicialización).
    */
}
```

Documentación: Esta declaración define una clase abierta (open class) en Kotlin, que permite que otras clases hereden de ella. Contiene una propiedad inmutable (val) llamada x.

Ejemplo 86:

```
class DerivedClass: BaseClass() {
    /*
        - 'class DerivedClass: BaseClass()': Declara una clase llamada 'DerivedClass' que extiende a 'BaseClass'.
            - 'BaseClass()': Indica que la clase base tiene un constructor al cual se llama al inicializar 'DerivedClass'.
    */

    fun foo() {
        /*
            - 'fun foo()': Define un método llamado 'foo' dentro de 'DerivedClass'.
            - Este método no recibe parámetros ni devuelve un valor.
        */

        println("x is equal to " + x)
        /*
            - 'println("x is equal to " + x)': Imprime un mensaje concatenado con el valor de 'x'.
```

- 'x': Hace referencia a la propiedad heredada de 'BaseClass'.

*/

}

}

Documentación: Este código define una clase DerivedClass que hereda de la clase BaseClass. Incluye un método foo que imprime el valor de la propiedad x de la clase base.

Ejemplo 87:

```
fun main(args: Array<String>) {
  /*
    - 'fun main(args: Array<String>)': Define la función principal del programa.
      - 'args: Array<String>': Es un arreglo de cadenas que puede contener argumentos pasados desde la línea de comandos al programa.
  */

  val derivedClass = DerivedClass()
  /*
    - 'val derivedClass': Declara una constante que almacena una instancia de 'DerivedClass'.
    - 'DerivedClass()': Llama al constructor predeterminado de la clase 'DerivedClass', que hereda de 'BaseClass'.
  */

  derivedClass.foo() // prints: 'x is equal to 10'
  /*
    - 'derivedClass.foo()': Invoca el método 'foo' en la instancia de 'DerivedClass'.
    - 'foo()' accede a la propiedad heredada 'x' desde 'BaseClass' y la imprime.
    - Imprime: 'x is equal to 10'.
  */
```

```
}
```

Documentación: Este fragmento de código define una función main en Kotlin que crea una instancia de la clase DerivedClass, invoca el método foo en esa instancia y muestra el valor heredado de x.

Ejemplo 88:

```
open class Person {
    /*
        - 'open class Person': Declara una clase abierta llamada 'Person'.
            - 'open': Permite que otras clases hereden de esta clase.
            - 'Person': Nombre de la clase que representa a una persona.
    */

    fun jump() {
        /*
            - 'fun jump()': Define un método llamado 'jump'.
                - Este método simula la acción de saltar al imprimir un mensaje.
        */

        println("Jumping...")
        /*
            - 'println("Jumping...")': Imprime el mensaje "Jumping..." en la consola.
            - Representa una salida que indica la acción realizada por la clase.
        */
    }
}
```

Documentación: Esta clase abierta (open class) en Kotlin permite herencia y contiene un método llamado jump. El método imprime un mensaje cuando se llama, lo que simula la acción de saltar.

Ejemplo 89:

```
class Ninja: Person() {
    /*
        - 'class Ninja: Person()': Define una clase llamada 'Ninja' que hereda de la clase 'Person'.
            - ': Person()': Indica que 'Ninja' utiliza el constructor de la clase base 'Person'.
    */


    fun sneak() {
        /*
            - 'fun sneak()': Declara un método llamado 'sneak'.
            - Este método es específico de la clase 'Ninja' y no está presente en la clase base.
        */


        println("Sneaking around...")
        /*
            - 'println("Sneaking around...")': Imprime el mensaje "Sneaking around..." en la consola.
            - Este mensaje simula la acción de moverse sigilosamente.
        */
    }
}
```

Documentación: Esta clase Ninja extiende la funcionalidad de la clase base Person y añade un nuevo comportamiento con el método sneak. Representa un **Ejemplo** de herencia en Kotlin.

Ejemplo 90:

```
fun main(args: Array<String>) {
    /*
        - 'fun main(args: Array<String>)': Define la función principal del programa.
```

- 'args: Array<String>': Permite recibir argumentos desde la línea de comandos (no utilizados en este **Ejemplo**).

*/

val ninja = Ninja()

/*

- 'val ninja': Declara una constante que almacena una instancia de la clase 'Ninja'.

- 'Ninja()': Llama al constructor predeterminado de la clase 'Ninja', heredada de 'Person'.

*/

ninja.jump() // prints: 'Jumping...'

/*

- 'ninja.jump()': Invoca el método 'jump' de la clase base 'Person'.

- Imprime: 'Jumping...' en la consola.

*/

ninja.sneak() // prints: 'Sneaking around...'

/*

- 'ninja.sneak()': Llama al método específico de la clase 'Ninja', el cual imprime un mensaje de acción sigilosa.

- Imprime: 'Sneaking around...' en la consola.

*/

}

Documentación: Este fragmento de código ejecuta la función principal (main) de un programa en Kotlin. Crea una instancia de la clase Ninja, heredada de Person, y llama a los métodos jump y sneak, los cuales muestran mensajes de acción en la consola.

Ejemplo 91:

val str = "Hello, World!"

```
/*
   - 'val str': Declara una variable inmutable llamada 'str'.
   - '"Hello, World!"': Cadena de texto que contiene el mensaje inicializado.
*/


println(str[1]) // Prints e
/*
   - 'str[1]': Accede al segundo carácter de la cadena (índice 1, ya que los índices comienzan en 0).
      - 'str[1]': Devuelve 'e', el segundo carácter de "Hello, World!".
   - 'println(...)': Imprime el carácter obtenido ('e') en la consola.
   - Resultado: 'e'.
*/
```

Documentación: Este fragmento accede a un carácter específico de una cadena en Kotlin utilizando el índice de la cadena. El resultado muestra cómo se puede obtener directamente un carácter en la posición deseada.

Ejemplo 92:

```
val s = "Hello, world!\n"
/*
   - 'val s': Declara una variable inmutable llamada 's'.
      - Las variables con 'val' no pueden ser reasignadas después de su inicialización.
   - '"Hello, world!\n"': Cadena de texto que contiene el mensaje "Hello, world!" seguido de un salto de línea.
      - '\n': Carácter especial que agrega un salto de línea al final del texto.
*/
```

Documentación: Este fragmento declara una variable inmutable (val) que contiene una cadena de texto. La cadena incluye un carácter especial de nueva línea (\n), que indica un salto de línea.

Ejemplo 93:

```
val i = 10

/*

   - 'val i': Declara una variable inmutable llamada 'i'.

   - '10': Asigna el valor entero 10 a la variable 'i'.

*/


val s = "i = $i" // evaluates to "i = 10"

/*

   - 'val s': Declara una variable inmutable llamada 's'.

   - '"i = $i"': Utiliza interpolación de cadenas para incluir el valor de 'i' dentro de la cadena.

      - '$i': Se sustituye por el valor de la variable 'i', que es 10.

   - Resultado: La cadena evaluada es "i = 10".

*/
```

Documentación: Este fragmento define una variable inmutable (val) y utiliza la interpolación de cadenas para construir un mensaje dinámico que incluye el valor de una variable dentro de una cadena.

Ejemplo 94:

```
val str1 = "Hello, World!"

/*

   - 'val str1': Declara una variable inmutable llamada 'str1'.

   - '"Hello, World!"': Asigna directamente una cadena con el texto "Hello, World!".

*/


val str2 = "Hello," + " World!"

/*

   - 'val str2': Declara una variable inmutable llamada 'str2'.

   - '"Hello," + " World!"': Crea una cadena concatenando dos cadenas separadas ("Hello," y " World!").
```

- El operador '+' combina ambas cadenas en una sola.

- Resultado: "Hello, World!".

*/

println(str1 == str2) // Prints true

/*

- 'str1 == str2': Compara el contenido de las cadenas 'str1' y 'str2' utilizando el operador de igualdad.

- En Kotlin, el operador '==' verifica si el contenido de las cadenas es igual (no solo si son la misma referencia en memoria).

- Ambas cadenas tienen el mismo contenido: "Hello, World!", por lo que la comparación devuelve 'true'.

- 'println(...)': Imprime el resultado de la comparación en la consola.

- Resultado: 'true'.

*/

Documentación: Este fragmento compara dos cadenas en Kotlin utilizando el operador de igualdad (==) para determinar si son iguales. Aunque se crean de formas diferentes, las cadenas tienen el mismo contenido, lo que resulta en un valor true.

Ejemplo 95:

interface MyInterface {

/*

- 'interface MyInterface': Declara una interfaz llamada 'MyInterface'.

- Las interfaces permiten definir métodos abstractos que deben ser implementados por las clases que las implementen.

*/

fun bar()

/*

- 'fun bar()': Declara un método abstracto llamado 'bar'.

- No tiene implementación en la interfaz y debe ser definido en las clases que implementen 'MyInterface'.

- No recibe parámetros ni retorna valores en este caso.

*/

}

Documentación: Este fragmento define una interfaz en Kotlin llamada MyInterface. Las interfaces se utilizan para especificar contratos que las clases pueden implementar. Contiene un método abstracto llamado bar.

Ejemplo 96:

```
interface MyInterface {
    /*
        - 'interface MyInterface': Declara una interfaz llamada 'MyInterface'.
        - Las interfaces en Kotlin pueden contener métodos con implementación.
    */

    fun withImplementation() {
        /*
            - 'fun withImplementation()': Declara un método llamado 'withImplementation' dentro de la interfaz.
            - Este método tiene una implementación por defecto.
        */

        print("withImplementation() was called")
        /*
            - 'print("withImplementation() was called")': Imprime el mensaje "withImplementation() was called" cuando se llama al método.
            - Este cuerpo de implementación permite que las clases que implementen la interfaz hereden el comportamiento predeterminado.
        */
```

```
    }
}
```

Documentación: Este fragmento define una interfaz llamada MyInterface en Kotlin, que incluye un método con implementación. En Kotlin, las interfaces pueden contener métodos con cuerpo de implementación, lo que las hace más versátiles que en algunos lenguajes de programación.

Ejemplo 97:

```
@Rule @JvmField val myRule = TemporaryFolder()

/*
  - '@Rule': Anotación de JUnit que indica que el campo o método anotado es una regla de prueba.
      - Las reglas de prueba en JUnit permiten definir comportamientos o configuraciones reutilizables para las pruebas.

  - '@JvmField': Anotación de Kotlin que expone el campo como un miembro de clase sin el generador de propiedad predeterminado.
      - Es necesaria porque JUnit requiere que las reglas de prueba sean campos públicos (y no propiedades de Kotlin).

  - 'val myRule': Declara un campo inmutable que actúa como una regla de prueba.
      - 'TemporaryFolder()': Una regla de JUnit que crea y gestiona automáticamente carpetas temporales durante una prueba.
          - Al final de la prueba, estas carpetas y su contenido se eliminan automáticamente.
*/
```

Documentación: Este fragmento de código utiliza la anotación @Rule junto con la declaración de un campo @JvmField en Kotlin para definir una regla de prueba con TemporaryFolder, una utilidad proporcionada por JUnit para manejar carpetas temporales en pruebas unitarias.

Ejemplo 98:

```
val i: Int = 42

/*
```

- 'val i': Declara una variable inmutable llamada 'i'.

  - Las variables con 'val' no pueden ser reasignadas después de su inicialización.

- ': Int': Especifica explícitamente el tipo de la variable como 'Int' (entero).

  - Aunque Kotlin puede inferir el tipo en este caso, añadirlo mejora la claridad.

- '= 42': Asigna el valor entero 42 a la variable 'i'.

*/

Documentación: Este fragmento de código declara una variable inmutable (val) llamada i en Kotlin, asignándole un valor entero de 42.

Ejemplo 99:

```
val i = if (someBoolean) 33 else 42
```

/*

- 'val i': Declara una variable inmutable llamada 'i'.

  - La variable no puede ser reasignada después de su inicialización.

- 'if (someBoolean) 33 else 42': Expresión condicional que evalúa la condición 'someBoolean'.

  - 'someBoolean': Una variable booleana (true o false) que determina qué valor se asigna a 'i'.

  - '33': Si 'someBoolean' es true, se asigna el valor 33.

  - '42': Si 'someBoolean' es false, se asigna el valor 42.

- Resultado: El valor final de 'i' depende del estado de 'someBoolean'.

*/

Documentación: Este fragmento de código utiliza una expresión condicional (if) en Kotlin para asignar un valor a la variable i según el estado de una condición booleana (someBoolean).

Ejemplo 100:

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }

/*

  - 'val allowedUsers': Declara una variable inmutable llamada 'allowedUsers' que almacenará los resultados filtrados.

  - 'users.filter { it.age > MINIMUM_AGE }': Aplica el método `filter` sobre la colección `users`.

      - 'users': Una lista o colección que contiene objetos de usuario.

      - 'filter': Función estándar de Kotlin que retorna una nueva lista con los elementos que cumplen la condición especificada.

      - '{ it.age > MINIMUM_AGE }': Bloque lambda que define la condición de filtrado.

          - 'it': Representa cada elemento dentro de la colección `users`.

          - 'it.age': Accede a la propiedad `age` de cada elemento (usuario).

          - '> MINIMUM_AGE': Compara la edad del usuario con el valor constante `MINIMUM_AGE`.

      - Resultado: Solo los usuarios cuya edad sea mayor que `MINIMUM_AGE` serán incluidos en la lista resultante.

*/
```

Documentación: Este fragmento utiliza la función filter de Kotlin para filtrar una lista llamada users. La condición dentro del bloque lambda selecciona solo aquellos usuarios cuya edad sea mayor que un valor mínimo especificado (MINIMUM_AGE).

Ejemplo 101:

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }

/*

  - 'val isOfAllowedAge': Declara una función lambda que verifica si un usuario tiene la edad permitida.

      - 'user: User': Parámetro de entrada de tipo `User`.

      - 'user.age > MINIMUM_AGE': Condición que compara la propiedad 'age' del usuario con la constante `MINIMUM_AGE`.

          - Retorna true si la edad del usuario es mayor que el valor mínimo permitido.

*/
```

```
val allowedUsers = users.filter(isOfAllowedAge)

/*

  - 'val allowedUsers': Declara una variable inmutable que almacenará la lista filtrada de usuarios.

  - 'users.filter(isOfAllowedAge)': Aplica el método `filter` a la colección `users` usando la lambda `isOfAllowedAge`.

      - 'users': Una lista o colección de objetos de tipo `User`.

      - 'filter': Retorna una nueva lista con los elementos que cumplen la condición definida en la lambda.

  - Resultado: 'allowedUsers' contendrá solo los usuarios cuya edad sea mayor que `MINIMUM_AGE`.

*/
```

Documentación: Este fragmento de código utiliza una función lambda para definir una condición de filtrado, que luego se aplica a una lista de usuarios (users) utilizando el método filter. El resultado es una nueva lista que incluye solo los usuarios cuya edad cumple con el criterio especificado.

Ejemplo 102:

```
object Benchmark {

  /*

    - 'object Benchmark': Declara un objeto singleton llamado 'Benchmark'.

      - En Kotlin, los objetos singleton son instancias únicas que se crean automáticamente al ser referenciadas.

      - Ideal para utilidades o componentes compartidos.


    - Este objeto proporciona la funcionalidad de medir el tiempo de ejecución de bloques de código.

  */


  fun realtime(body: () -> Unit): Duration {
```

```
/*
    - 'fun realtime(body: () -> Unit): Duration': Declara una función llamada 'realtime'.
        - 'body: () -> Unit': Parámetro que representa un bloque de código sin argumentos y sin valor de retorno.
        - 'Duration': Tipo de retorno que representa la duración entre dos instantes de tiempo.
    */

    val start = Instant.now()
    /*
        - 'val start = Instant.now()': Captura el instante de tiempo actual al comienzo del bloque de código.
        - 'Instant.now()': Método proporcionado por Java Time API para obtener el instante actual.
    */

    try {
        body()
        /*
            - 'body()': Ejecuta el bloque de código proporcionado como argumento.
            - Si ocurre alguna excepción durante la ejecución, el bloque 'finally' aún se ejecutará.
        */
    } finally {
        val end = Instant.now()
        /*
            - 'val end = Instant.now()': Captura el instante de tiempo actual al final del bloque de código.
        */
```

```
        return Duration.between(start, end)
        /*
            - 'Duration.between(start, end)': Calcula la duración entre los instantes 'start' y 'end'.
            - Devuelve la duración calculada como el resultado de la función 'realtime'.
        */
    }
  }
}
```

Documentación: Este fragmento define un objeto singleton llamado Benchmark en Kotlin, que contiene una función llamada realtime. Esta función mide el tiempo de ejecución de un bloque de código y devuelve la duración total.

Ejemplo 103:

```
fun IntArray.addTo(dest: IntArray) {
  /*
    - 'fun IntArray.addTo(dest: IntArray)': Declara una función de extensión para `IntArray`.
        - 'IntArray': Es un arreglo de enteros en Kotlin.
        - 'addTo(dest: IntArray)': Define una función que toma un argumento `dest`, que es otra matriz de enteros.
  */

  for (i in 0 .. size - 1) {
    /*
        - 'for (i in 0 .. size - 1)': Itera desde el índice 0 hasta el último índice válido del arreglo.
            - '0 .. size - 1': Rango inclusivo basado en el tamaño del arreglo (`this`).
    */
```

```
        dest[i] += this[i]
        /*
            - 'dest[i] += this[i]': Suma el valor en el índice `i` del arreglo actual (`this`) al valor correspondiente en el índice `i` de la matriz de destino (`dest`).
            - Modifica la matriz `dest` in-place.
        */
    }
}
```

Documentación: Este fragmento de código define una función de extensión (addTo) para la clase IntArray en Kotlin, que permite sumar los elementos de una matriz de enteros (IntArray) a otra matriz de enteros de destino (dest), índice por índice.

Ejemplo 104:

```
open class Super
/*
    - 'open class Super': Declara una clase abierta (heredable) llamada 'Super'.
*/

class Sub : Super()
/*
    - 'class Sub : Super()': Declara una clase llamada 'Sub', que extiende la clase 'Super'.
*/

fun Super.myExtension() = "Defined for Super"
/*
    - 'fun Super.myExtension()': Define una función de extensión para la clase 'Super'.
        - Retorna la cadena "Defined for Super".
        - Esta función no afecta ni anula métodos de las clases derivadas, ya que las funciones de extensión son estáticas.
```

```
*/


fun Sub.myExtension() = "Defined for Sub"
/*
   - 'fun Sub.myExtension()': Define una función de extensión para la clase 'Sub'.
      - Retorna la cadena "Defined for Sub".
      - Las extensiones son resueltas en tiempo de compilación según el tipo declarado, no el tipo del objeto real.
*/


fun callMyExtension(myVar: Super) {
   /*
      - 'fun callMyExtension(myVar: Super)': Define una función que acepta un parámetro de tipo 'Super'.
      - Este parámetro será utilizado para llamar a las extensiones de función relacionadas.
   */


   println(myVar.myExtension())
   /*
      - 'myVar.myExtension()': Llama a la función de extensión basada en el tipo declarado de 'myVar'.
      - Dado que 'myVar' tiene el tipo declarado 'Super', se ejecutará la extensión 'myExtension' definida para 'Super'.
      - Resultado: "Defined for Super".
   */
}


callMyExtension(Sub())
```

/*

- 'callMyExtension(Sub())': Llama a la función `callMyExtension` pasando una instancia de `Sub`.

- Aunque el objeto real es de tipo `Sub`, el tipo declarado del parámetro es `Super`.

- Esto significa que se ejecutará la extensión definida para `Super`, no para `Sub`.

- Resultado: "Defined for Super".

*/

Documentación: Este fragmento combina herencia, extensiones de funciones en Kotlin y el concepto de "dispatch estático" (resolución del tipo de llamada en tiempo de compilación). Aunque podría parecer que el método myExtension de Sub debería ejecutarse, el resultado será **"Defined for Super"** debido a cómo Kotlin maneja las funciones de extensión.

Ejemplo 105:

```
fun Long.humanReadable(): String {
    /*
        - 'fun Long.humanReadable()': Declara una función de extensión para números de tipo 'Long'.
            - Transforma un número en una cadena legible para humanos con unidades de tamaño (bytes, kilobytes, etc.).
    */

    if (this <= 0) return "0"
    /*
        - 'if (this <= 0) return "0"': Devuelve "0" si el número es menor o igual a 0.
            - 'this': Referencia al valor del número en el que se aplica la función.
    */

    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    /*
        - 'units': Define un arreglo que contiene las unidades utilizadas para representar tamaños.
```

- Las unidades siguen un orden ascendente desde bytes ('B') hasta exabytes ('EB').

*/

val digitGroups = (Math.log10(this.toDouble()) / Math.log10(1024.0)).toInt()

/*

- 'Math.log10(this.toDouble()) / Math.log10(1024.0)': Calcula el número de grupos de dígitos necesarios para determinar la unidad adecuada.

- Usa logaritmos base 10 y base 1024 para convertir el número en su representación escalada.

- '.toInt()': Convierte el resultado en un entero, que representa el índice dentro del arreglo 'units'.

*/

return DecimalFormat("#,##0.#").format(this / Math.pow(1024.0, digitGroups.toDouble())) + " " + units[digitGroups]

/*

- 'DecimalFormat("#,##0.#")': Formatea el valor escalado para incluir separadores de miles y un único decimal.

- 'this / Math.pow(1024.0, digitGroups.toDouble())': Escala el valor original dividiéndolo por 1024 elevado al índice correspondiente.

- 'units[digitGroups]': Determina la unidad adecuada según el índice calculado.

- Combina el valor formateado con la unidad, devolviendo el resultado como cadena legible para humanos.

*/

}

fun Int.humanReadable(): String {

/*

- 'fun Int.humanReadable()': Declara una función de extensión para números de tipo 'Int'.

- Convierte el número en 'Long' y llama a la función 'humanReadable()' definida para 'Long'.

*/

return this.toLong().humanReadable()

/*

- 'this.toLong().humanReadable()': Cambia el tipo del número actual de 'Int' a 'Long' y aplica la función 'humanReadable'.

*/

}

Documentación: Este fragmento define dos funciones de extensión en Kotlin para transformar números (Long e Int) en representaciones legibles para humanos con unidades como "B", "KB", "MB", etc., dependiendo de su tamaño.

Ejemplo 106:

fun Path.exists(): Boolean = Files.exists(this)

/*

- 'fun Path.exists()': Declara una función de extensión para verificar si un archivo o directorio existe en la ruta especificada.

- 'Files.exists(this)': Utiliza el método estático de `java.nio.file.Files` para determinar la existencia del archivo o directorio asociado a `this`.

- Retorna `true` si el archivo o directorio existe; de lo contrario, `false`.

*/

fun Path.notExists(): Boolean = !this.exists()

/*

- 'fun Path.notExists()': Declara una función de extensión que verifica si un archivo o directorio no existe.

- '!this.exists()': Llama a la función de extensión `exists()` y devuelve el resultado negado.

- Retorna `true` si el archivo o directorio **no** existe; de lo contrario, `false`.

*/

fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()

/*

- 'fun Path.deleteRecursively()': Declara una función de extensión para eliminar el archivo o directorio en la ruta especificada, junto con todo su contenido.

- 'this.toFile()': Convierte el objeto `Path` en un objeto `File` para acceder a métodos específicos de la clase `File`.

- 'deleteRecursively()': Llama al método proporcionado por `File` para eliminar recursivamente el archivo o carpeta.

- Retorna `true` si la eliminación fue exitosa; de lo contrario, `false`.

*/

Documentación: Este fragmento define tres funciones de extensión en Kotlin para la clase Path, lo que permite realizar operaciones comunes de manera más directa utilizando métodos personalizados.

Ejemplo 107:

val x: Path = Paths.get("dirName").apply {

/*

- 'val x: Path': Declara una variable inmutable llamada 'x' de tipo `Path`.

- `Path`: Clase de Java NIO que representa una ruta en el sistema de archivos.

- 'Paths.get("dirName")': Crea un objeto `Path` que apunta al directorio especificado ("dirName").

*/

if (Files.notExists(this)) throw IllegalStateException("The important file does not exist")

/*

- 'apply { ... }': Ejecuta un bloque de código sobre el objeto recién creado y retorna el mismo objeto.

- Dentro del bloque, `this` hace referencia al objeto `Path` actual.

- 'Files.notExists(this)': Comprueba si el archivo o directorio no existe en el sistema de archivos.

- `Files`: Clase de utilidades de Java NIO para operaciones con archivos.

- `this`: Referencia al objeto `Path` actual.

- 'throw IllegalStateException("...")': Lanza una excepción si el directorio o archivo no existe.

- `IllegalStateException`: Indica un estado no válido en el programa.

- Mensaje de la excepción: "The important file does not exist".

*/

}

Documentación: Este fragmento de código define una variable inmutable x que representa una ruta (Path) en el sistema de archivos. Utiliza apply, una función de alcance de Kotlin, para realizar operaciones adicionales sobre el objeto Path antes de asignarlo a x.

Ejemplo 108:

fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)

/*

- 'fun Temporal.toIsoString()': Declara una función de extensión para la interfaz 'Temporal' de la API de Java Time.

- Permite aplicar esta funcionalidad a cualquier objeto que implemente la interfaz 'Temporal'.

- 'DateTimeFormatter.ISO_INSTANT': Utiliza un formateador estándar para representar instantes en formato ISO-8601.

- **Ejemplo** de salida: "2023-04-06T12:34:56.789Z".

- '.format(this)': Aplica el formateador al objeto actual (`this`) del tipo 'Temporal'.

- 'this': Referencia al objeto sobre el cual se llama la función de extensión.

*/

Documentación: Este código define una función de extensión en Kotlin para la interfaz Temporal. La función convierte un objeto de tipo Temporal a su representación en formato ISO usando el formateador DateTimeFormatter.ISO_INSTANT.

Ejemplo 109:

```
class Something {
    companion object {}
    /*
        - 'class Something': Define una clase llamada 'Something'.
        - 'companion object': Declara un objeto de compañía asociado a la clase 'Something'.
            - Los objetos de compañía actúan como miembros estáticos de la clase y permiten agregar extensiones o métodos personalizados.
    */
}

class SomethingElse {
    /*
        - 'class SomethingElse': Define una clase llamada 'SomethingElse'.
        - No incluye un objeto de compañía, lo que influye en cómo se pueden usar las funciones de extensión.
    */
}

fun Something.Companion.fromString(s: String): Something = ...
/*
    - 'fun Something.Companion.fromString(s: String)': Declara una función de extensión para el objeto de compañía de la clase 'Something'.
        - Permite llamar a esta función directamente desde el objeto de compañía como si fuera un miembro estático.
```

- 's: String': Representa el argumento de entrada (cadena) para esta función.

- 'Something': Devuelve una instancia de la clase 'Something' basada en la entrada.

*/

fun SomethingElse.fromString(s: String): SomethingElse = ...

/*

- 'fun SomethingElse.fromString(s: String)': Declara una función de extensión para instancias de la clase 'SomethingElse'.

- Se aplica a objetos de tipo 'SomethingElse'.

- 's: String': Representa el argumento de entrada (cadena) para esta función.

- 'SomethingElse': Devuelve una instancia de la clase 'SomethingElse'.

*/

fun main(args: Array<String>) {

/*

- 'fun main(args: Array<String>)': Define la función principal del programa, donde se prueban los casos descritos.

*/

Something.fromString("") // válido

/*

- Llama a la función de extensión 'fromString' en el objeto de compañía de 'Something'.

- Esto es válido porque la extensión está definida para el objeto de compañía.

*/

SomethingElse().fromString("") // válido

/*

- Llama a la función de extensión 'fromString' en una instancia de 'SomethingElse'.

- Esto es válido porque la extensión está definida para instancias de la clase 'SomethingElse'.

*/

SomethingElse.fromString("") // inválido

/*

- Intenta llamar a 'fromString' directamente desde la clase 'SomethingElse', como si fuera un miembro estático.

- Esto no es válido porque 'fromString' está definido solo para instancias y no existe un objeto de compañía en 'SomethingElse'.

*/

}

Documentación: Este fragmento explora el uso de funciones de extensión en Kotlin aplicadas a objetos de compañía (companion objects) y a instancias de clase. También demuestra las limitaciones del uso estático de funciones de extensión en clases sin objetos de compañía.

Ejemplo 110:

class KColor(val value: Int)

/*

- 'class KColor': Declara una clase llamada `KColor`.

- 'val value: Int': Propiedad inmutable que almacena un valor entero asociado al color.

*/

private val colorCache = mutableMapOf<KColor, Color>()

/*

- 'private val colorCache': Declara una caché mutable privada para mapear instancias de `KColor` a `Color`.

- 'mutableMapOf<KColor, Color>()': Crea un mapa vacío que permite agregar, actualizar y eliminar pares clave-valor.

- La caché se usa para almacenar instancias `Color` asociadas a cada instancia `KColor`, mejorando la eficiencia.

*/

val KColor.color: Color

get() = colorCache.getOrPut(this) { Color(value, true) }

/*

- 'val KColor.color': Declara una propiedad de extensión llamada `color` para la clase `KColor`.

- Esta propiedad no almacena valores directamente, sino que los calcula dinámicamente.

- 'get() =': Define un getter personalizado para la propiedad `color`.

- 'colorCache.getOrPut(this)': Intenta obtener el valor correspondiente a `this` (instancia actual de `KColor`) en la caché.

- Si no existe una entrada en la caché, ejecuta el bloque `{ Color(value, true) }` para crear y almacenar el valor.

- 'Color(value, true)': Crea una nueva instancia de `Color` con el valor de `value` de `KColor`.

- 'true': Argumento booleano que podría representar una configuración relacionada con transparencia, colores ARGB, o similar (dependiendo del contexto de `Color`).

*/

Documentación: Este fragmento de código utiliza extensiones de propiedades en Kotlin para asociar objetos de tipo KColor con objetos Color. Además, usa una caché mutable (colorCache) para optimizar el almacenamiento y recuperación de instancias Color, evitando la recreación innecesaria de objetos.

Ejemplo 111:

public val name = "Avijit"

/*

- 'public': Es el modificador de visibilidad predeterminado.

- Significa que la propiedad 'name' es accesible desde **cualquier lugar** del programa.

- **Ejemplo** de acceso: Puede ser llamado dentro del mismo archivo, módulo o proyecto, sin restricciones.

*/
private val name = "Avijit"

/*

- 'private': Restringe el acceso a la propiedad 'name' al **archivo** donde está declarado.

- Solo puede ser utilizado por funciones o clases dentro del mismo archivo donde se definió.

*/

protected val name = "Avijit"

/*

- 'protected': Restringe el acceso a la propiedad 'name' a la **clase** donde se define, y a sus **subclases**.

- No es accesible fuera de esta jerarquía.

- Nota: Solo se puede usar en contextos dentro de clases o interfaces.

*/

internal val name = "Avijit"

/*

- 'internal': Restringe el acceso a la propiedad 'name' al **módulo** donde se declara.

- Un módulo puede ser un proyecto IntelliJ IDEA o una librería que se compile junto al código.

*/

Documentación: En Kotlin, las palabras clave public, private, protected e internal definen los **modificadores de visibilidad** de clases, propiedades, y funciones. Controlan qué partes del código pueden acceder a ciertos elementos.

Ejemplo 112:

data class User(var firstname: String, var lastname: String, var age: Int)

/*

- 'data class User': Define una clase de datos llamada `User`.

  - Las clases de datos son ideales para modelar objetos que almacenan datos simples.

- 'var firstname: String': Declara una propiedad mutable llamada `firstname` de tipo `String`, que representa el nombre del usuario.

- 'var lastname: String': Declara una propiedad mutable llamada `lastname` de tipo `String`, que representa el apellido del usuario.

- 'var age: Int': Declara una propiedad mutable llamada `age` de tipo `Int`, que representa la edad del usuario.

- Ventajas de las clases de datos:

  - Kotlin genera automáticamente las siguientes funcionalidades:

    - `equals()`: Compara objetos basándose en las propiedades.

    - `hashCode()`: Genera un código hash para el objeto.

    - `toString()`: Proporciona una representación de cadena de las propiedades del objeto.

    - Componentes (`component1`, `component2`, etc.) para destructuración.

*/

Documentación: Este fragmento declara una clase de datos en Kotlin llamada User. Las clases de datos están diseñadas para almacenar datos y proveen funcionalidades útiles de manera automática, como métodos de equals, hashCode, y toString, además de destructuración.

Ejemplo 113:

```
val list = listOf(1, 2, 3, 4, 5, 6)
```

/*

- 'listOf(1, 2, 3, 4, 5, 6)': Crea una lista inmutable que contiene los números enteros 1, 2, 3, 4, 5 y 6.

- 'val list': Declara una variable inmutable llamada 'list'.

*/

val even = list.filter { it % 2 == 0 }

/*

- 'list.filter { it % 2 == 0 }': Aplica el método `filter` a la lista 'list'.

- `filter`: Retorna una nueva lista que contiene solo los elementos que cumplen la condición especificada.

- '{ it % 2 == 0 }': Bloque lambda que define la condición de filtrado.

- 'it': Representa cada elemento de la lista durante la iteración.

- 'it % 2 == 0': Verifica si el elemento es divisible entre 2 (es decir, si es un número par).

- Resultado: Una nueva lista con los números pares [2, 4].

*/

println(even)

/*

- 'println(even)': Imprime la lista resultante de números pares en la consola.

- Resultado: [2, 4].

*/

Documentación: Este fragmento utiliza la función filter de Kotlin para filtrar elementos de una lista. En este caso, selecciona solo los números pares (even) mediante una condición en un bloque lambda.

Ejemplo 114:

class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table {

/*

- 'class MyTable': Declara una clase llamada `MyTable`.

- 'private constructor(table: Table<Int, Int, Int>)':

- Define un constructor privado que recibe un objeto de tipo `Table<Int, Int, Int>`.

- Restringe el acceso directo al constructor desde fuera de la clase, lo que fuerza el uso de constructores secundarios o métodos de fábrica.

- ': Table<Int, Int, Int> by table':

- Implementa la interfaz `Table<Int, Int, Int>` mediante delegación al objeto `table`.

- Delegación significa que las llamadas a los métodos de `Table` se redirigen automáticamente al objeto `table` proporcionado.

*/

constructor() : this(TreeBasedTable.create())

/*

- 'constructor()':

- Define un constructor secundario que no toma parámetros.

- Llama al constructor privado y pasa una nueva instancia creada por `TreeBasedTable.create()`.

- 'TreeBasedTable.create()': Método de fábrica que crea una instancia específica de `Table`.

- Este enfoque proporciona una manera predeterminada de crear instancias de `MyTable`.

*/

}

Documentación: Este fragmento define una clase en Kotlin llamada MyTable que utiliza delegación para implementar la interfaz Table. También se emplea un constructor privado y un constructor secundario para flexibilidad en la creación de instancias.

Ejemplo 115:

class MySpecialCase : Serializable {

/*

- 'class MySpecialCase': Declara una clase llamada `MySpecialCase`.

- ': Serializable': Implementa la interfaz `Serializable`.

- Permite convertir objetos de esta clase a una secuencia de bytes y viceversa, útil para almacenamiento o transmisión.

*/

companion object {

/*

- 'companion object': Declara un objeto de compañía asociado a la clase `MySpecialCase`.

- Este objeto proporciona un espacio para definir miembros estáticos o compartidos por todas las instancias de la clase.

*/

private const val serialVersionUID: Long = 123

/*

- 'private const val serialVersionUID': Declara una constante de tipo `Long` con visibilidad privada.

- `serialVersionUID`: Es un identificador único que se utiliza para verificar la compatibilidad de versiones durante el proceso de deserialización.

- Garantiza que los objetos deserializados correspondan a la misma versión de la clase.

*/

}

}

Documentación: Este fragmento define una clase en Kotlin llamada MySpecialCase que implementa la interfaz Serializable, haciéndola apta para la serialización. Además, declara un objeto de compañía que incluye una constante serialVersionUID para garantizar la consistencia en el proceso de deserialización.

Ejemplo 116:

class MyClass {

```
    fun doSomething(): MyClass {
        // Ejecuta otra acción en el contexto de la clase
        someOtherAction()
        return this // Retorna la instancia actual de la clase
    }

    private fun someOtherAction() {
        println("Acción realizada!")
    }
}
```

Documentación: Este código intenta definir una función doSomething. Sin embargo, contiene un problema: el uso de return this no es válido en un contexto fuera de una clase, ya que this solo tiene sentido cuando se refiere a una instancia de una clase en Kotlin. Aquí está la versión corregida, junto con los comentarios y la explicación de su comportamiento.

Ejemplo 117:

```
val str = "foo"
/*
    - 'val str = "foo"': Declara una variable inmutable llamada `str` y le asigna la cadena `"foo"`.
*/

str.let {
    /*
        - 'str.let': Llama a la función de extensión `let` en la variable `str`.
        - La función `let` toma un bloque lambda como argumento y lo ejecuta con `str` como receptor.
        - Dentro del bloque, `it` hace referencia al objeto `str`.
    */
```

```
    println(it) // it
    /*
        - 'println(it)': Imprime el valor de `it`, que en este contexto es el valor de `str` ("foo").
        - Resultado: "foo".
    */
}
```

Documentación: Este código demuestra el uso de la función de alcance let en Kotlin. let permite realizar operaciones en el contexto de un objeto y devuelve el resultado del bloque lambda. La palabra clave it se refiere al objeto sobre el cual se llama la función.

Ejemplo 118:

```
import java.io.File
/*
    - 'import java.io.File': Importa la clase `File` de la biblioteca estándar de Java.
        - Permite realizar operaciones relacionadas con archivos y directorios en el sistema de archivos.
*/


fun makeDir(path: String): File {
    /*
        - 'fun makeDir(path: String): File': Declara una función llamada `makeDir` que crea un directorio.
            - 'path: String': Recibe como parámetro la ruta del directorio a crear (en forma de cadena).
            - 'File': Indica que el tipo de retorno de la función será un objeto `File`.
    */


    val result = File(path)
    /*
        - 'val result = File(path)': Crea un objeto `File` asociado a la ruta especificada por `path`.
```

- No necesita utilizar `new`, ya que Kotlin simplifica la creación de objetos.

*/

result.mkdirs()

/*

- 'result.mkdirs()': Intenta crear el directorio especificado en la ruta `path`.

- Incluye la creación de todos los directorios padres necesarios si no existen.

- Retorna `true` si el directorio fue creado con éxito, o `false` si ya existía o si hubo un error.

*/

return result

/*

- 'return result': Devuelve el objeto `File` que representa el directorio (nuevo o existente).

- Este objeto puede ser usado posteriormente para realizar operaciones sobre el directorio.

*/

}

Documentación: Este código define una función llamada makeDir en Kotlin, que crea un directorio en la ruta especificada. Utiliza la clase File de la API de entrada/salida de Java para realizar las operaciones de manejo de archivos y directorios.

Ejemplo 119:

object CommonUtils {

/*

- 'object CommonUtils': Declara un objeto llamado `CommonUtils`.

- Los objetos en Kotlin son instancias únicas (singletons).

- Se utilizan comúnmente para almacenar constantes, métodos utilitarios o estados globales.

```
    */


    var anyname: String = "Hello"
    /*
        - 'var anyname: String': Declara una propiedad mutable llamada `anyname` de tipo `String`.
            - Inicialmente se le asigna el valor "Hello".
            - Al ser mutable (`var`), su valor puede cambiarse en tiempo de ejecución.
    */


    fun dispMsg(message: String) {
        /*
            - 'fun dispMsg(message: String)': Declara una función llamada `dispMsg` que toma un parámetro `message` de tipo `String`.
            - Esta función no retorna ningún valor, ya que es de tipo `Unit` (implícito en Kotlin).
        */


        println(message)
        /*
            - 'println(message)': Imprime el contenido del parámetro `message` en la consola.
            - Esto permite mostrar el mensaje en tiempo de ejecución.
        */
    }
}
```

Documentación: Este código define un objeto en Kotlin llamado CommonUtils. Los objetos en Kotlin son instancias únicas (singletons) creadas automáticamente al referenciarlas, lo que los hace ideales para utilidades compartidas como esta.

Ejemplo 120:

```
public enum SharedRegistry {
  /*
    - 'public enum SharedRegistry': Declara un enum llamado `SharedRegistry`.
      - Los enumeradores en Java pueden contener métodos, y esta implementación específica actúa como un singleton.
      - Esto asegura que haya una sola instancia (`INSTANCE`) durante la ejecución del programa.
  */

  INSTANCE;
  /*
    - 'INSTANCE': Define el único valor/enumeración de `SharedRegistry`, que es la instancia singleton.
    - Este es el patrón recomendado para implementar singletons en Java de manera segura y eficiente.
  */

  public void register(String key, Object thing) {
    /*
      - 'public void register(String key, Object thing)': Declara un método público llamado `register`.
        - Parámetros:
          - 'key': Una cadena que actúa como la clave para registrar el objeto.
          - 'thing': El objeto asociado a la clave.
        - Nota: En este código, el método no realiza ninguna funcionalidad específica (está vacío).
    */
  }
}
```

Documentación: El código que compartiste utiliza un patrón singleton con enum en Java para garantizar que haya solo una instancia de SharedRegistry. Esto es útil cuando se requiere un registro global centralizado que se puede acceder de forma segura en todo el programa.

Ejemplo 121:

```
fun printNumbers(vararg numbers: Int) {
  /*
    - 'fun printNumbers(vararg numbers: Int)': Declara una función llamada `printNumbers`.
      - 'vararg numbers': Utiliza el modificador `vararg` para permitir que la función reciba una cantidad variable de argumentos enteros.
      - 'Int': Define que los argumentos son de tipo entero.
  */

  for (number in numbers) {
    /*
      - 'for (number in numbers)': Itera a través del arreglo `numbers`.
        - Cada elemento en `numbers` es representado como `number` durante la iteración.
    */

    println(number)
    /*
      - 'println(number)': Imprime cada número en una nueva línea.
    */
  }
}
```

Documentación: Este código utiliza la función de Kotlin vararg para definir un parámetro que puede aceptar un número variable de argumentos. La función printNumbers recorre e imprime todos los números proporcionados.

Ejemplo 122:

```
fun printNumbers(vararg numbers: Int) {
```

```
    /*
        - 'fun printNumbers(vararg numbers: Int)': Declara una función llamada `printNumbers`.
            - 'vararg numbers': Utiliza el modificador `vararg` para permitir la recepción de un número indefinido de argumentos enteros.
            - 'Int': Especifica que los argumentos son de tipo entero.
    */


    for (number in numbers) {
        /*
            - 'for (number in numbers)': Inicia un bucle `for` que recorre cada elemento en el arreglo `numbers`.
                - `number`: Representa el valor actual de la iteración dentro del arreglo.
        */


        println(number)
        /*
            - 'println(number)': Imprime el valor actual de `number` en la consola.
            - Cada número será impreso en una línea separada.
        */
    }
}
```

Documentación: Este código define una función llamada printNumbers en Kotlin, que utiliza el modificador vararg para aceptar una cantidad variable de números enteros como parámetros. La función recorre todos los números proporcionados y los imprime uno por uno.

Ejemplo 123:

```
val foo: Int by lazy { 1 + 1 }
/*
    - 'val foo: Int': Declara una variable inmutable llamada `foo` de tipo entero (`Int`).
```

- 'by lazy': Utiliza la delegación `lazy` para diferir la inicialización de la variable.

  - Esto significa que `foo` no se inicializa hasta que sea accedida por primera vez.

- '{ 1 + 1 }': Bloque lambda que define la lógica de inicialización de `foo`.

  - En este caso, evalúa la expresión `1 + 1` y asigna el resultado a `foo`.

*/

println(foo)

/*

- 'println(foo)': Accede a la variable `foo` e imprime su valor en la consola.

  - Al acceder por primera vez, el bloque definido dentro de `lazy` se ejecuta, inicializando `foo` con el resultado de `1 + 1`.

  - En futuras accesos, el valor ya calculado se reutiliza.

*/

Documentación: Este código utiliza la delegación con lazy en Kotlin para inicializar la variable foo únicamente cuando se accede por primera vez. Esto es útil para optimizar el rendimiento, ya que evita inicializaciones innecesarias hasta que la variable sea requerida.

Ejemplo 124:

import kotlin.properties.Delegates

/*

- 'import kotlin.properties.Delegates': Importa la clase `Delegates` que contiene métodos de delegación, como `observable` y `vetoable`.

*/

var foo: String by Delegates.observable("1") { property, oldValue, newValue ->

  /*

  - 'var foo: String': Declara una propiedad mutable llamada `foo` de tipo `String`.

  - 'by Delegates.observable("1")': Utiliza la delegación para observar los cambios en `foo`.

    - `"1"`: Valor inicial asignado a `foo`.

```
        - El bloque lambda se ejecuta cada vez que el valor de `foo` cambia.
            - 'property': Representa metadatos de la propiedad (como su nombre).
            - 'oldValue': Valor anterior de la propiedad antes del cambio.
            - 'newValue': Nuevo valor asignado a la propiedad.
    */

    println("${property.name} was changed from $oldValue to $newValue")
    /*
        - 'property.name': Obtiene el nombre de la propiedad (`foo` en este caso).
        - 'oldValue': Representa el valor anterior asignado a `foo`.
        - 'newValue': Representa el nuevo valor asignado a `foo`.
        - 'println(...)': Imprime un mensaje en la consola detallando el cambio.
    */
}

foo = "2"
/*
    - 'foo = "2"': Asigna un nuevo valor a la propiedad `foo`.
    - Esto activa el bloque observable, que detecta y registra el cambio.
    - En este caso, se imprimirá:
      foo was changed from 1 to 2
*/
```

Documentación: Este código utiliza la delegación de propiedades en Kotlin mediante Delegates.observable, lo que permite observar y reaccionar a los cambios de valor en una propiedad mutable (var). Cada vez que el valor de la propiedad cambia, se ejecuta un bloque lambda que registra el cambio, proporcionando el nombre de la propiedad, su valor anterior y el nuevo valor.

Ejemplo 125:

```
val map = mapOf("foo" to "1")

/*

   - 'val map = mapOf("foo" to "1")': Crea un mapa inmutable que asocia la clave `"foo"` con el valor `"1"`.

   - Aquí, tanto la clave como el valor son de tipo `String` para ser consistentes con la propiedad delegada `foo`.

*/


val foo: String by map

/*

   - 'val foo: String by map': Declara una propiedad delegada llamada `foo` de tipo `String`.

      - Utiliza el mapa `map` para delegar el valor de `foo`.

      - Cuando se accede a `foo`, se recupera el valor asociado a la clave `"foo"` en `map`.

*/


println(foo)

/*

   - 'println(foo)': Imprime el valor de la propiedad `foo`, que será `"1"`.

*/
```

Documentación: Este código intenta usar la delegación de propiedades con un Map en Kotlin. Sin embargo, hay un problema: el valor asociado a la clave "foo" en el mapa es un Int, mientras que la propiedad foo está declarada como un String. Esto generará un error de tipo en tiempo de compilación porque Kotlin espera que los tipos sean consistentes.

Ejemplo 126:

```
class MyDelegate {

   /*

      - 'class MyDelegate': Declara una clase llamada `MyDelegate`, que implementa la lógica personalizada para la delegación de propiedades.

   */
```

```
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        /*
            - 'operator fun getValue': Sobrecarga del operador `getValue`, requerido para delegación de propiedades.
            - Parámetros:
                - 'owner: Any?': El propietario de la propiedad delegada. Puede ser una instancia de clase u objeto.
                - 'property: KProperty<*>': Representa metadatos de la propiedad delegada, como su nombre.
            - Retorna: El valor asociado con la propiedad delegada, en este caso, una cadena fija.
        */

        return "Delegated value"
        /*
            - Retorna la cadena "Delegated value" como el valor de la propiedad delegada.
        */
    }
}

val foo: String by MyDelegate()
/*
    - 'val foo: String': Declara una propiedad inmutable `foo` de tipo `String`.
    - 'by MyDelegate()': Usa una instancia de `MyDelegate` como delegado para controlar cómo se obtiene el valor de `foo`.
*/

println(foo)
```

```
/*
   - 'println(foo)': Imprime el valor de la propiedad `foo`.
      - Al acceder a `foo`, se llama al método `getValue` de `MyDelegate`, que retorna "Delegated value".
      - Por lo tanto, se imprimirá en consola: Delegated value
*/
```

Documentación: Este código utiliza la delegación personalizada en Kotlin, implementada mediante la sobrecarga del operador getValue. Esto permite asignar una lógica específica a la delegación de una propiedad (foo), controlando cómo se obtiene su valor.

Ejemplo 127:

```
import java.lang.ref.WeakReference
/*
   - 'import java.lang.ref.WeakReference': Importa la clase `WeakReference` de la biblioteca estándar de Java.
      - Se utiliza para mantener una referencia débil a un objeto, lo que permite que el recolector de basura lo elimine si ya no hay referencias fuertes.
*/


class MyMemoryExpensiveClass {
   /*
      - 'class MyMemoryExpensiveClass': Declara una clase llamada `MyMemoryExpensiveClass`.
         - Se utiliza para representar un objeto que ocupa una cantidad significativa de memoria.
   */


   companion object {
      /*
         - 'companion object': Declara un objeto de compañía asociado a la clase.
```

- Los objetos de compañía son útiles para definir propiedades y métodos estáticos.

*/

var reference: WeakReference<MyMemoryExpensiveClass>? = null

/*

- 'var reference': Propiedad mutable que almacena una referencia débil a una instancia de `MyMemoryExpensiveClass`.

- Inicialmente es nula, y puede actualizarse cuando una nueva instancia de la clase es creada.

*/

fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {

/*

- 'fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit)': Declara un método estático que acepta un bloque de código como parámetro.

- El bloque `block` toma un objeto de tipo `MyMemoryExpensiveClass` como argumento.

*/

reference?.let {

/*

- 'reference?.let': Verifica si la referencia no es nula.

- Si no es nula, ejecuta el bloque dentro del `let`.

*/

it.get()?.let(block)

/*

- 'it.get()': Obtiene la instancia de `MyMemoryExpensiveClass` almacenada en la referencia débil, si está disponible.

- '?.let(block)': Si la instancia no ha sido recolectada por el recolector de basura, ejecuta el bloque proporcionado.

```
            */
        }
    }
}


init {
    /*
        - 'init': Bloque de inicialización que se ejecuta cada vez que se crea una instancia de `MyMemoryExpensiveClass`.
    */

    reference = WeakReference(this)
    /*
        - 'reference = WeakReference(this)': Asigna una referencia débil a la instancia actual (`this`) de la clase.
            - Esto actualiza la propiedad `reference` del objeto de compañía con la nueva instancia creada.
    */
}
}
```

Documentación: Este código define una clase en Kotlin llamada MyMemoryExpensiveClass, que utiliza una referencia débil (WeakReference) para gestionar una única instancia. Las referencias débiles son útiles para evitar problemas de memoria, como fugas, cuando se trabaja con objetos que son costosos de mantener en memoria.

Ejemplo 128:

```
class MainActivity : AppCompatActivity() {
    /*
```

- 'class MainActivity : AppCompatActivity()': Declara una clase que extiende `AppCompatActivity`, lo cual es común para actividades en Android con soporte para temas modernos.

*/

lateinit var mRecyclerView: RecyclerView

/*

- 'lateinit var mRecyclerView: RecyclerView': Declara una variable llamada `mRecyclerView` que se inicializa más adelante.

- `lateinit`: Indica que la variable no se inicializa inmediatamente pero se garantiza su inicialización antes de usarla.

- `RecyclerView`: Es un componente de Android para mostrar listas o grids de manera eficiente.

*/

val mAdapter: RecyclerAdapter = RecyclerAdapter()

/*

- 'val mAdapter: RecyclerAdapter': Declara e inicializa un adaptador de tipo `RecyclerAdapter`, que será utilizado por el `RecyclerView`.

- `val`: Declara que `mAdapter` es inmutable después de la inicialización.

*/

override fun onCreate(savedInstanceState: Bundle?) {

/*

- 'override fun onCreate(savedInstanceState: Bundle?)': Método principal de la actividad que se ejecuta al crearla.

- 'savedInstanceState: Bundle?': Contiene el estado previo de la actividad si se recrea después de una interrupción.

*/

super.onCreate(savedInstanceState)

/*

- 'super.onCreate': Llama al método `onCreate` de la clase base (`AppCompatActivity`) para realizar la inicialización estándar.

*/

setContentView(R.layout.activity_main)

/*

- 'setContentView(R.layout.activity_main)': Establece el archivo de diseño XML asociado con la actividad.

- `R.layout.activity_main`: Representa el archivo XML que define el diseño de la interfaz de usuario.

*/

val toolbar = findViewById(R.id.toolbar) as Toolbar

/*

- 'findViewById(R.id.toolbar)': Busca la vista con ID `toolbar` en el archivo de diseño XML.

- 'as Toolbar': Convierte la vista encontrada en el tipo `Toolbar`.

*/

setSupportActionBar(toolbar)

/*

- 'setSupportActionBar(toolbar)': Establece la barra de herramientas como la barra de acción principal de la actividad.

*/

mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

/*

- 'mRecyclerView = findViewById(R.id.recycler_view)': Encuentra y asigna la vista del `RecyclerView` al objeto `mRecyclerView`.

- 'as RecyclerView': Convierte la vista encontrada en el tipo `RecyclerView`.

*/

mRecyclerView.setHasFixedSize(true)

/*

- 'mRecyclerView.setHasFixedSize(true)': Indica que el tamaño del `RecyclerView` no cambiará.

- Esto optimiza el rendimiento cuando los elementos de la lista tienen tamaños consistentes.

*/

mRecyclerView.layoutManager = LinearLayoutManager(this)

/*

- 'mRecyclerView.layoutManager = LinearLayoutManager(this)': Establece el administrador de diseño como un `LinearLayoutManager`, que organiza los elementos en una lista lineal vertical.

*/

mAdapter.RecyclerAdapter(getList(), this)

/*

- 'mAdapter.RecyclerAdapter(getList(), this)': Llama al constructor del adaptador y lo inicializa con la lista de datos proporcionada por `getList()` y el contexto actual (`this`).

- Nota: El uso de `RecyclerAdapter()` aquí parece redundante, ya que el adaptador ya se inicializó previamente.

*/

mRecyclerView.adapter = mAdapter

```
    /*
        - 'mRecyclerView.adapter = mAdapter': Asigna el adaptador al `RecyclerView`, que utiliza el adaptador para manejar y mostrar los datos.
    */
}

private fun getList(): ArrayList<String> {
    /*
        - 'private fun getList(): ArrayList<String>': Declara una función privada que retorna una lista de cadenas.
        - `ArrayList<String>`: Representa una lista mutable que almacena cadenas.
    */

    var list: ArrayList<String> = ArrayList()
    /*
        - 'var list: ArrayList<String> = ArrayList()': Crea una lista vacía de cadenas que será poblada en el bucle.
    */

    for (i in 1..10) {
        /*
            - 'for (i in 1..10)': Itera desde 1 hasta 10, incluyendo ambos extremos.
            - Es equivalente a: `1 <= i && i <= 10`.
        */

        println(i)
        /*
```

- 'println(i)': Imprime el valor actual de `i` en la consola para propósitos de depuración.

*/

list.add("$i")

/*

- 'list.add("$i")': Agrega la representación de cadena del número `i` a la lista.

- `"$i"`: Usa interpolación para convertir `i` en una cadena.

*/

}

return list

/*

- 'return list': Retorna la lista completa poblada con cadenas desde `"1"` hasta `"10"`.

*/

}

}

Documentación: Este código define una actividad llamada MainActivity en Android, que utiliza un RecyclerView para mostrar una lista de elementos. La actividad incluye configuraciones estándar como el uso de una barra de herramientas (Toolbar) y un adaptador personalizado (RecyclerAdapter) para administrar el contenido del RecyclerView.

Ejemplo 129:

```
val c1 = String::class

/*

- 'val c1': Declara una variable inmutable llamada `c1`.

- 'String::class': Usa el operador `::class` para obtener una referencia a la clase de `String`.

- El resultado es un objeto de tipo `KClass<String>`, que representa la clase `String` en Kotlin.

*/
```

```
val c2 = MyClass::class

/*

   - 'val c2': Declara una variable inmutable llamada `c2`.

   - 'MyClass::class': Usa el operador `::class` para obtener una referencia a la clase de `MyClass`.

      - El resultado es un objeto de tipo `KClass<MyClass>`.

      - Asegúrate de que `MyClass` esté definido previamente en tu código.

*/
```

Documentación: Este código demuestra cómo obtener referencias de tipo KClass en Kotlin usando el operador ::class. El operador proporciona una forma de acceder a la clase asociada con un tipo específico. Aquí String representa una clase predefinida y MyClass sería una clase personalizada.

Ejemplo 130:

```
fun isPositive(x: Int) = x > 0

/*

   - 'fun isPositive(x: Int)': Declara una función llamada `isPositive`, que toma un argumento entero `x`.

   - '= x > 0': Retorna `true` si `x` es mayor que 0, y `false` en caso contrario.

      - Esta es una forma concisa de escribir funciones en Kotlin.

*/


val numbers = listOf(-2, -1, 0, 1, 2)

/*

   - 'val numbers': Declara una lista inmutable llamada `numbers`.

   - 'listOf(-2, -1, 0, 1, 2)': Crea una lista que contiene los valores -2, -1, 0, 1, y 2.

*/


println(numbers.filter(::isPositive))
```

/*

- 'numbers.filter(::isPositive)': Llama al método `filter` de la lista `numbers`.

- `filter`: Retorna una nueva lista que contiene solo los elementos que cumplen la condición proporcionada.

- '::isPositive': Pasa la referencia de la función `isPositive` como argumento para `filter`.

- Esto significa que cada elemento de la lista `numbers` será evaluado con `isPositive`.

- Resultado: Una lista que contiene solo los números positivos [1, 2].

- 'println(...)': Imprime la lista filtrada en la consola.

*/

Documentación: Este código utiliza una función de orden superior en Kotlin para filtrar una lista de números, conservando solo aquellos que son positivos. La función ::isPositive se pasa como una referencia, haciendo que el código sea más limpio y modular.

Ejemplo 131:

```
val stringKClass: KClass<String> = String::class

/*

- 'val stringKClass: KClass<String>': Declara una variable inmutable llamada `stringKClass` de tipo `KClass<String>`.

- 'String::class': Obtiene una referencia de la clase `String` en Kotlin, representada como un objeto de tipo `KClass<String>`.

- `KClass` es parte de la API de reflección de Kotlin y proporciona metadatos sobre la clase asociada.

*/


val c1: Class<String> = stringKClass.java

/*

- 'val c1: Class<String>': Declara una variable inmutable llamada `c1` de tipo `Class<String>`, que pertenece a la API de Java.
```

- 'stringKClass.java': Convierte la referencia de clase de Kotlin (`KClass`) a su equivalente de Java (`Class`).

- Esto es útil para interactuar con bibliotecas o código en Java que espera tipos `Class`.

*/

val c2: Class<MyClass> = MyClass::class.java

/*

- 'val c2: Class<MyClass>': Declara una variable inmutable llamada `c2` de tipo `Class<MyClass>`.

- 'MyClass::class.java': Obtiene la referencia de clase en Kotlin y la convierte directamente a su equivalente de Java.

- Nota: Asegúrate de que `MyClass` esté definido antes de usarlo. Si no está definido, el compilador generará un error.

*/

Documentación: Este código demuestra cómo trabajar con referencias de clase en Kotlin utilizando KClass y su equivalente en Java (Class). Kotlin proporciona una integración fluida con la API de reflección de Java, lo que hace posible obtener metadatos de clase tanto en el contexto de Kotlin como en Java.

Ejemplo 132:

open class BaseExample(val baseField: String)

/*

- 'open class BaseExample': Declara una clase base llamada `BaseExample`.

- 'open': Permite que otras clases hereden de `BaseExample`.

- 'val baseField: String': Declara una propiedad inmutable llamada `baseField` de tipo `String`.

*/

class Example(val field1: String, val field2: Int, baseField: String): BaseExample(baseField) {

/*

- 'class Example': Declara una clase llamada `Example`.

- 'val field1: String': Declara una propiedad inmutable llamada `field1` de tipo `String`.

- 'val field2: Int': Declara una propiedad inmutable llamada `field2` de tipo entero (`Int`).

- 'baseField: String': Recibe un argumento que se pasa al constructor de `BaseExample`.

- ': BaseExample(baseField)': Indica que `Example` hereda de `BaseExample` y pasa el argumento `baseField` al constructor de la clase base.

*/

val field3: String

get() = "Property without backing field"

/*

- 'val field3: String': Declara una propiedad inmutable llamada `field3`.

- 'get()': Define un getter personalizado.

- No utiliza un campo de respaldo (`backing field`), por lo que siempre retorna el valor `"Property without backing field"`.

*/

val field4 by lazy { "Delegated value" }

/*

- 'val field4': Declara una propiedad inmutable llamada `field4`.

- 'by lazy': Utiliza delegación para inicializar `field4` solo cuando se accede por primera vez.

- Esto optimiza el rendimiento y reduce el consumo de memoria, especialmente si la inicialización es costosa.

- El valor inicial es `"Delegated value"`.

*/

private val privateField: String = "Private value"

```
    /*
        - 'private val privateField: String': Declara una propiedad inmutable y privada llamada `privateField` de tipo `String`.
        - La propiedad solo es accesible dentro de la clase `Example`.
        - Su valor es `"Private value"`.
    */
}
```

Documentación: Este código muestra cómo Kotlin maneja herencia y propiedades en clases. Define una clase base BaseExample que es heredada por la clase Example. La clase Example incorpora diversos tipos de propiedades, demostrando técnicas comunes en Kotlin, como propiedades delegadas (by lazy), propiedades con y sin campo de respaldo (backing field), y propiedades privadas.

Ejemplo 133:

```
open class BaseExample(val baseField: String)
/*
  - 'open class BaseExample': Declara una clase base llamada `BaseExample`.
    - 'open': Permite que otras clases hereden de `BaseExample`.
    - 'val baseField: String': Declara una propiedad inmutable llamada `baseField` de tipo `String`.
*/

class Example(val field1: String, val field2: Int, baseField: String): BaseExample(baseField)
{
  /*
    - 'class Example': Declara una clase llamada `Example`.
    - 'val field1: String': Declara una propiedad inmutable llamada `field1` de tipo `String`.
    - 'val field2: Int': Declara una propiedad inmutable llamada `field2` de tipo entero (`Int`).
    - 'baseField: String': Recibe un argumento que se pasa al constructor de `BaseExample`.
    - ': BaseExample(baseField)': Indica que `Example` hereda de `BaseExample` y pasa el argumento `baseField` al constructor de la clase base.
```

```
*/

val field3: String
    get() = "Property without backing field"
    /*
        - 'val field3: String': Declara una propiedad inmutable llamada `field3`.
        - 'get()': Define un getter personalizado.
            - No utiliza un campo de respaldo (`backing field`), por lo que siempre retorna el valor `"Property without backing field"`.
    */

val field4 by lazy { "Delegated value" }
    /*
        - 'val field4': Declara una propiedad inmutable llamada `field4`.
        - 'by lazy': Utiliza delegación para inicializar `field4` solo cuando se accede por primera vez.
            - Esto optimiza el rendimiento y reduce el consumo de memoria, especialmente si la inicialización es costosa.
        - El valor inicial es `"Delegated value"`.
    */

private val privateField: String = "Private value"
    /*
        - 'private val privateField: String': Declara una propiedad inmutable y privada llamada `privateField` de tipo `String`.
        - La propiedad solo es accesible dentro de la clase `Example`.
        - Su valor es `"Private value"`.
    */
```

}

Documentación: Este código muestra cómo Kotlin maneja herencia y propiedades en clases. Define una clase base BaseExample que es heredada por la clase Example. La clase Example incorpora diversos tipos de propiedades, demostrando técnicas comunes en Kotlin, como propiedades delegadas (by lazy), propiedades con y sin campo de respaldo (backing field), y propiedades privadas.

Ejemplo 134:

```
import kotlin.text.Regex

/*

   - 'import kotlin.text.Regex': Importa la clase `Regex` de Kotlin, que permite trabajar con expresiones regulares.

   - Proporciona funciones como `matches`, `find`, `replace`, entre otras, para manipular cadenas.

*/


var string = /* some string */

/*

   - 'var string': Declara una variable mutable llamada `string`.

   - Esta variable debe contener una cadena de texto que se comparará con los patrones definidos.

   - Ejemplo: `var string = "example"`

*/


val regex1 = Regex( /* pattern */ )

val regex2 = Regex( /* pattern */ )

/*

   - 'val regex1' y 'val regex2': Declaran dos expresiones regulares inmutables.

   - 'Regex( /* pattern */ )': Crea una expresión regular basada en el patrón proporcionado.

      - Ejemplo: `val regex1 = Regex("^[a-z]+$")` (patrón para una cadena de solo letras minúsculas).
```

```
*/

when {
    regex1.matches(string) -> /* do stuff */
    /*
        - 'regex1.matches(string)': Verifica si toda la cadena `string` coincide exactamente con el patrón de `regex1`.
        - '-> /* do stuff */': Si hay coincidencia, ejecuta el bloque de código asociado.
        - **Ejemplo**: `println("String matches regex1")`
    */

    regex2.matches(string) -> /* do stuff */
    /*
        - Similar al caso anterior, verifica si la cadena coincide con el patrón de `regex2`.
        - Ejecuta otro bloque de código si encuentra coincidencia.
    */

    /* etc */
    /*
        - Se pueden agregar más patrones y casos según sea necesario.
        - El bloque `when` permite manejar múltiples condiciones sin necesidad de una larga cadena de `if-else`.
    */
}
```

Documentación: Este código demuestra cómo trabajar con expresiones regulares (Regex) en Kotlin y utilizarlas dentro de una estructura de control when para realizar acciones específicas en función de si la cadena de texto (string) coincide con uno o más patrones definidos.

Ejemplo 135:

```
import kotlin.text.Regex

/*
   - 'import kotlin.text.Regex': Importa la clase `Regex` de Kotlin.
      - Permite utilizar expresiones regulares para evaluar, manipular o buscar coincidencias dentro de cadenas de texto.
*/


var string = /* some string */

/*
   - 'var string': Declara una variable mutable llamada `string`.
      - Debería contener el texto que será evaluado en los diferentes casos dentro del bloque `when`.
      - Ejemplo: `var string = "example"`
*/


when {
   Regex( /* pattern */ ).matches(string) -> /* do stuff */
   /*
      - 'Regex( /* pattern */ ).matches(string)': Usa la clase `Regex` para crear un patrón y verifica si la cadena `string` coincide exactamente con él.
         - El método `matches`: Retorna `true` si toda la cadena coincide con el patrón.
         - '/* pattern */': Aquí debería estar definido un patrón válido, como `"^[a-z]+$"` (solo letras minúsculas).
      - '-> /* do stuff */': Ejecuta un bloque de código si hay coincidencia con el patrón.
   */


   Regex( /* pattern */ ).matches(string) -> /* do stuff */
   /*
```

- Similar al caso anterior, evalúa la cadena con otro patrón de expresión regular.

- Esta estructura permite manejar diferentes condiciones en un formato limpio y organizado.

*/

/* etc */

/*

- Se pueden agregar más patrones y bloques según los requisitos.

- La estructura `when` facilita manejar múltiples casos de manera declarativa y modular.

*/

}

Documentación: Este código ejemplifica cómo trabajar con expresiones regulares (Regex) dentro de una estructura when en Kotlin para evaluar una cadena de texto (string) en múltiples casos. Aunque el código no está completo, su estructura permite asociar patrones específicos con bloques de acción (do stuff).

Ejemplo 136:

var string: String = "Hello World!"

/*

- 'var string: String': Declara una variable mutable llamada `string` de tipo no nulo (`String`).

- Se inicializa con el valor `"Hello World!"`.

- En Kotlin, una variable de tipo `String` no puede contener valores nulos.

*/

var nullableString: String? = null

/*

- 'var nullableString: String?': Declara una variable mutable llamada `nullableString` de tipo nulo (`String?`).

- El símbolo `?` indica que la variable puede contener un valor nulo o una cadena.

- Se inicializa con `null`.

*/

string = nullableString

/*

- 'string = nullableString': Intenta asignar la variable `nullableString` (que puede ser nula) a `string` (que no acepta valores nulos).

- Esto produce un error de compilación porque Kotlin no permite asignar un tipo nulo a una variable no nula directamente.

*/

nullableString = string

/*

- 'nullableString = string': Asigna la variable `string` (no nula) a `nullableString` (que acepta valores nulos).

- Esto es válido porque cualquier valor de tipo no nulo puede asignarse a un tipo nulo.

*/

Documentación: Este fragmento de código demuestra cómo Kotlin maneja los tipos nulos (nullable) y no nulos (non-nullable). Kotlin tiene un sistema de tipos estrictos que evita problemas relacionados con la asignación de valores nulos. A continuación, explicamos por qué se produce el error de compilación en el **Ejemplo**.

Ejemplo 137:

val string: String? = "Hello World!"

/*

- 'val string: String?': Declara una variable inmutable llamada `string` de tipo nulo (`String?`).

- El símbolo `?` indica que la variable puede contener un valor nulo o una cadena.

- Se inicializa con la cadena `"Hello World!"`.

*/

```
print(string.length)

/*

   - 'print(string.length)': Intenta acceder directamente a la propiedad `length` del objeto `string`.

   - Esto genera un error de compilación porque `string` es de tipo nulo (`String?`), y Kotlin no permite acceder a propiedades de un tipo nulo sin manejar el caso donde `string` podría ser `null`.

*/


print(string?.length)

/*

   - 'print(string?.length)': Utiliza el operador seguro (`?.`) para acceder a la propiedad `length` de manera segura.

      - Si `string` no es `null`, el valor de `length` se calcula y se imprime.

      - Si `string` es `null`, el resultado de la expresión será `null`, y la impresión mostrará `null`.

*/
```

Documentación: Este código ejemplifica cómo manejar tipos nulos (nullable) en Kotlin y cómo utilizar el operador seguro (?.) para acceder a propiedades o métodos de un tipo que podría ser nulo. Kotlin tiene un sistema de tipos estricto diseñado para evitar errores como NullPointerException.

Ejemplo 138:

```
typealias StringValidator = (String) -> Boolean

/*

   - 'typealias StringValidator': Declara un alias para un tipo funcional que toma un argumento de tipo `String` y retorna un valor de tipo `Boolean`.

   - Uso:

      - Se puede usar como tipo de función para validar cadenas.
```

- Por **Ejemplo**, podría representar un validador para verificar si una cadena cumple con ciertos criterios.

- **Ejemplo** de implementación:

  val validateLength: StringValidator = { it.length > 5 }

*/

typealias Reductor<T, U, V> = (T, U) -> V

/*

- 'typealias Reductor<T, U, V>': Declara un alias genérico para un tipo funcional que toma dos argumentos de tipos `T` y `U`, y retorna un valor de tipo `V`.

- Uso:

  - Útil en casos donde se necesitan funciones de combinación o reducción.

  - Por **Ejemplo**, podría usarse para combinar dos números en un tercer valor, como sumar dos enteros y devolver un entero.

- **Ejemplo** de implementación:

  val sumReducer: Reductor<Int, Int, Int> = { a, b -> a + b }

*/

Documentación: Este código utiliza typealias en Kotlin para declarar alias de tipos, proporcionando nombres más legibles y simplificando el uso de tipos funcionales genéricos y específicos. Esto mejora la claridad del código, especialmente en contextos donde los tipos son complejos.

Ejemplo 139:

typealias Parents = Pair<Person, Person>

/*

- 'typealias Parents': Declara un alias llamado `Parents` que representa un par (`Pair`) de objetos de tipo `Person`.

- `Pair<Person, Person>`: Indica que el alias se utiliza para encapsular dos instancias de la clase `Person`.

  - El primer elemento del par puede representar al padre y el segundo a la madre, por **Ejemplo**.

- Uso:

  - Simplifica la manipulación de relaciones entre dos personas.

  - **Ejemplo**:

    val parents: Parents = Pair(Person("Father"), Person("Mother"))

*/

typealias Accounts = List<Account>

/*

- 'typealias Accounts': Declara un alias llamado `Accounts` que representa una lista (`List`) de objetos de tipo `Account`.

- `List<Account>`: Indica que el alias se utiliza para manejar una colección de cuentas.

- Uso:

  - Facilita el trabajo con grupos de cuentas.

  - **Ejemplo**:

    val accounts: Accounts = listOf(Account("Savings"), Account("Checking"))

*/

Documentación: Este código utiliza typealias en Kotlin para definir alias de tipos que simplifican el uso de estructuras de datos específicas. Estos alias proporcionan nombres más intuitivos y mejoran la claridad del código cuando trabajamos con tipos complejos o genéricos.

Ejemplo 140:

import javax.swing.*

/*

- 'import javax.swing.*': Importa todas las clases del paquete `javax.swing`, que proporciona componentes para construir interfaces gráficas de usuario (GUIs) en Java.

*/

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {

  /*

- 'fun JFrame.menuBar(init: JMenuBar.() -> Unit)': Declara una función de extensión para la clase `JFrame`.

- Permite configurar una barra de menús (`JMenuBar`) de manera declarativa utilizando un bloque lambda.

- 'init: JMenuBar.() -> Unit': El parámetro `init` es un bloque lambda que actúa sobre una instancia de `JMenuBar`.

*/

val menuBar = JMenuBar()

/*

- 'val menuBar = JMenuBar()': Crea una nueva instancia de `JMenuBar`.

*/

menuBar.init()

/*

- 'menuBar.init()': Ejecuta el bloque lambda proporcionado por el parámetro `init` en el contexto de `menuBar`.

- Esto permite configurar la barra de menús directamente en el bloque lambda.

*/

setJMenuBar(menuBar)

/*

- 'setJMenuBar(menuBar)': Asocia el `menuBar` creado con el `JFrame` actual como su barra de menús principal.

*/

}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {

/*

- 'fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit)': Declara una función de extensión para la clase `JMenuBar`.

- Permite agregar un menú (`JMenu`) con un título (`caption`) y configurar su contenido de manera declarativa.

*/

val menu = JMenu(caption)

/*

- 'val menu = JMenu(caption)': Crea una nueva instancia de `JMenu` con el título proporcionado en `caption`.

*/

menu.init()

/*

- 'menu.init()': Ejecuta el bloque lambda `init` en el contexto del menú creado (`menu`), permitiendo agregar elementos al menú.

*/

add(menu)

/*

- 'add(menu)': Agrega el menú configurado a la barra de menús (`JMenuBar`).

*/

}

fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {

/*

- 'fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit)': Declara una función de extensión para la clase `JMenu`.

- Permite agregar un ítem de menú (`JMenuItem`) con un título (`caption`) y configurarlo usando un bloque lambda.

*/

val menuItem = JMenuItem(caption)

/*

- 'val menuItem = JMenuItem(caption)': Crea un nuevo ítem de menú con el texto proporcionado en `caption`.

*/

menuItem.init()

/*

- 'menuItem.init()': Ejecuta el bloque lambda `init` en el contexto del ítem de menú creado (`menuItem`).

*/

add(menuItem)

/*

- 'add(menuItem)': Agrega el ítem configurado al menú (`JMenu`).

*/

}

Documentación: Este código utiliza el concepto de funciones de extensión con receptores (receiver functions) en Kotlin para simplificar la construcción de menús en una aplicación Swing. Las funciones definidas (menuBar, menu y menuItem) permiten construir una estructura de menús anidados de forma declarativa y más legible.