UD00. PROGRAMACIÓN EN KOTLIN

¿Qué es Kotlin?

Kotlin es un lenguaje de programación moderno desarrollado por JetBrains. Tiene una sintaxis clara y concisa que hace que el código sea fácil de leer.

A pesar de su "juventud", Kotlin es ampliamente usado y su popularidad está en auge.

El conocimiento de Kotlin permite a los desarrolladores el crear aplicaciones para dispositivos móviles, para servidor y aplicaciones de escritorio, así como frameworks y librerías. Como lenguaje de propósito general, puede ser usado en múltiples áreas como servicios financieros, telecomunicaciones, sistemas embebidos, medicina, herramientas de desarrollo (como IntelliJ IDEA),…

Breve Historia

En Julio de 2011, JetBrains desveló el Proyecto Kotlin, un nuevo lenguaje para la Plataforma Java. El nombre hace referencia a la isla Kotlin, cerca de San Petersburgo, Rusia. El objetivo primero del proyecto era proporcionar una alternativa más concisa y segura a Java en todos los contextos donde se estaba usando ese lenguaje.

En 2016, se libera la primera versión estable (Kotlin v1.0). La comunidad de desarrolladores ya estaba interesada en usar este lenguaje, especialmente en Android.

En la conferencia Google I/O de 2017, Google anuncia soporte first-class para Kotlin en Android

En la conferencia Google I/O de 2019, Google anuncia Kotlin como su lenguaje preferente para Android:

En el siguiente documento, Google justifica su elección de Kotlin como lenguaje preferente para la plataforma Android.

En este momento, Kotlin es considerado un lenguaje de propósito general para muchas plataformas, no sólo para Android. El lenguaje tiene varias *"releases"* nuevas cada año. La última versión está disponible en el sitio oficial

Características

Kotlin se ha diseñado como un lenguaje práctico, lo que significa que su objetivo principal es la resolución de problemas reales en lugar de tener un propósito académico.

Kotlin soporta múltiples paradigmas de programación, como la programación impertiva, programación estructurada, programación orientada a objetos, programación genérica y programación funcional.

Por otro lado. Kotlin es un lenguaje tool-friendly, lo que significa que es compatible con herramientas de desarrollo populares como IntelliJ IDEA, Eclipse, Android Studio y otras.

Capitulo 1

KOTLIN BÁSICO

EXPRESIONES Y VARIABLES

Tipos, Variables y Constantes

Al igual que Java, Kotlin es un en lenguaje de **tipado estático**. Por tanto, todas las variables o constantes declaradas en nuestros programas deberán tener un tipo de dato asociado que determinará tanto la naturaleza del almacenamiento como las operaciones posibles sobre dichos datos.

La sintaxis general para la declaración de nuestras variables es la siguiente:

```
val|var nombre [: tipo] [= valor]
```

donde:

- **val** se emplea para declarar variables de **sólo lectura**. Es decir, una vez inicializadas, no pueden modificar su valor (equivalente al empleo de *final* en Java). En general deberíamos optar por emplear este tipo de variables salvo que necesitemos modificar su valor posteriormente.
- **var** se emplea para declarar variables que pueden ser reasignadas (pueden cambiar su valor)
- **nombre** es el **identificador** de la variable. En general, se aplican las mismas reglas que en Java:
    - Debe empezar por una letra o guión bajo (_) seguido por letras o dígitos
    - No se permiten espacios en blanco ni caracteres especiales
    - Son *case-sensitive*.

    - En general, usaremos notación *lowerCamelCase* para variables y funciones, y *UpperCamelCase* (ó *PascalCase*) para los nombres de las clases
- en caso de que no inicialicemos la variable con un valor, deberemos indicar el **tipo** de la misma
- podemos inicializar la variable con un **valor** en el momento de su declaración. En este caso, no estamos obligados a especificar su tipo ya que Kotlin se encargará de asignar el más adecuado en función de dicho valor. Es lo que se conoce como **inferencia de tipo**

Algunos ejemplos:

1. Variables(`var`):

```
var edad: Int = 25
edad = 26 // El valor de 'edad' puede ser modificado
```

En este ejemplo, se declara una variable llamada edad de tipo Int y se le asigna un valor inicial de 25. Luego, el valor de edad se actualiza a 26.

2. Constantes(`val`):

```
val nombre: String = "Juan"
// nombre = "Pedro" // Esto generaría un error, ya que no puedes cambiar el valor de una constante
```

3. Inferencia de tipos:

```
var cantidad = 10 // El tipo se infiere como Int
val pi = 3.14     // El tipo se infiere como Double
```

TIPOS BÁSICOS

En Kotlin, todo son objetos. Así, todas las variables que creemos son de **tipos referenciados** (no existen **tipos primitivos** como en Java).

En todo caso, dado que Kotlin emplea representaciones internas equivalentes a las de Java para números, caracteres y booleanos, se aplicarán los rangos de valores correspondientes.

Tipo	Descripción	Ejemplo de Declaración
`Byte`	Número entero de 8 bits con signo	`val numero: Byte = 10`
`Short`	Número entero de 16 bits con signo	`val valor: Short = 100`
`Int`	Número entero de 32 bits con signo (muy común)	`val edad: Int = 25`
`Long`	Número entero de 64 bits con signo	`val saldo: Long = 100000`

Tipo	Descripción	Ejemplo de Declaración
`Float`	Número en coma flotante de 32 bits (precisión simple)	`val pi: Float = 3.14f`
`Double`	Número en coma flotante de 64 bits (precisión doble)	`val precio: Double = 9.99`
`Char`	Carácter Unicode	`val letra: Char = 'A'`
`Boolean`	Valor booleano (`true` o `false`)	`val esVerano: Boolean = true`
`String`	Cadena de caracteres	`val mensaje: String = "Hola, Kotlin!"`
`Unit`	Representa la ausencia de valor (similar a `void`)	`val resultado: Unit = realizarAlgo()`
`Nothing`	Indica que una función nunca retorna ni termina	-

Todas las variables inicializadas con enteros que no superen el valor máximo de `Int` tendrán el tipo inferido `Int`. En caso de que el valor del literal sea superior a dicho máximo, se asignará el tipo `Long`. Para especificar el tipo `Long` de forma explícita, se añadirá una **L** como sufijo del valor

Ejemplo de declaración de variables:

```
val edad: Int = 25
val precio: Double = 9.99
val inicial: Char = 'A'
val esVerano: Boolean = true
val mensaje: String = "Hola, Kotlin!"
val resultado: Unit = realizarAlgo()
```

Recuerda que en muchos casos, Kotlin es capaz de inferir automáticamente el tipo de variable, por lo que no siempre es necesario especificar el tipo explícitamente al declarar variables.

```
val cantidad = 42   // Infiere Int
val temperatura = 23.5 // Infiere Double
val letra = 'B'    // Infiere Char
```

Conversión de Tipos

En Kotlin, a diferencia de Java y otros lenguajes como C++, no existen conversiones automáticas entre tipos numéricos. Las conversiones deben ser realizadas de forma **explícita**. Para ello, las clases correspondientes nos proporcionan toda una serie de métodos

Aquí tienes una tabla que muestra algunas de las funciones de conversión más comunes en Kotlin:

Función de Conversión	Descripción	Ejemplo
`toByte()`	Convierte a tipo `Byte`	`val numero: Int = 42; val byteValue: Byte = numero.toByte()`

Función de Conversión	Descripción	Ejemplo
`toShort()`	Convierte a tipo `Short`	`val numero: Int = 42; val shortValue: Short = numero.toShort()`
`toInt()`	Convierte a tipo `Int`	`val numeroString = "42"; val intValue: Int = numeroString.toInt()`
`toLong()`	Convierte a tipo `Long`	`val numero: Int = 42; val longValue: Long = numero.toLong()`
`toFloat()`	Convierte a tipo `Float`	`val numero: Double = 3.14; val floatValue: Float = numero.toFloat()`
`toDouble()`	Convierte a tipo `Double`	`val numero: Int = 42; val doubleValue: Double = numero.toDouble()`
`toString()`	Convierte a tipo `String`	`val numero: Int = 42; val numeroString: String = numero.toString()`
`toChar()`	Convierte a tipo `Char`	`val charValue: Char = 'A'; val intValue: Int = charValue.toInt()`
`toBoolean()`	Convierte a tipo `Boolean`	`val booleanString = "true"; val booleanValue: Boolean = booleanString.toBoolean()`

Para practicar

Ejercicio 1: Declaración de Variables

Declarar variables para almacenar información personal sobre una persona:

- Nombre: Juan
- Edad: 30
- Altura: 1,75
- Estado civil: soltero

Ejercicio 2: Conversión de Tipos

Declarar una variable que almacene la cantidad de dólares (100) y otra que almacene el tipo de cambio (20), luego calcular el equivalente en pesos mexicanos.

Solución Ejercicio 1: Declaración de Variables

```
fun main() {
    val nombre: String = "Juan"
    val edad: Int = 30
    val altura: Double = 1.75
    val estadoCivil: Boolean = false

    println("Nombre: $nombre, Edad: $edad, Altura: $altura, Casado: $estadoCivil")
}
```

Solución Ejercicio 2: Conversión de Tipos

```
fun main() {
    val dolares: Double = 100.0
    val tipoCambio: Double = 20.0
```

```
    val pesos: Double = dolares * tipoCambio

    println("Equivalente en pesos mexicanos: $pesos")
}
```

OPERACIONES BÁSICAS

Las operaciones que se pueden realizar en Kotlin quedan resumidas en esta tabla :

Operación	Descripción	Ejemplo
Suma (+)	Suma de valores numéricos o concatenación de strings	`val suma = 5 + 3` / `val nombreCompleto = nombre + " " + apellido`
Resta (-)	Resta de valores numéricos	`val resta = 10 - 4`
Multiplicación (*)	Producto de valores numéricos	`val producto = 6 * 7`
División (/)	División de valores numéricos	`val division = 12 / 3`
Módulo (%)	Resto de la división de valores numéricos	`val resto = 10 % 3`
Incremento (++)	Incremento en 1 de una variable	`contador++`
Decremento (--)	Decremento en 1 de una variable	`indice--`
Asignación (=)	Asigna el valor de la derecha al objeto de la izquierda	`x = 42`
Comparación (==, !=)	Compara dos valores para igualdad o desigualdad	`val igual = a == b` / `val diferente = x != y`
Comparación (<, <=, >, >=)	Compara dos valores para ver si uno es menor, menor o igual, mayor o mayor o igual que el otro	`val menor = a < b` / `val mayorIgual = x >= y`
Operaciones Lógicas (&&, `, !`)		
Bitwise (and, or, xor, inv)	Operaciones a nivel de bits AND, OR, XOR e inversión	`val resultado = a and b`
Desplazamiento de Bits (shl, shr)	Desplazamiento de bits a la izquierda y a la derecha	`val resultado = x shl 2`

Estas son solo algunas de las operaciones básicas que puedes realizar en Kotlin. El lenguaje también admite una variedad de otras operaciones más avanzadas, como funciones matemáticas, operaciones en colecciones, manipulación de cadenas y más. Consultar documentación oficial para ver más.

Para practicar

Ejercicio 1: Incremento y Decremento

Declarar una variable que represente la cantidad de productos en un carrito de compras (5), luego incrementarla y decrementarla.

Solución Ejercicio 1: Incremento y Decremento

```
fun main() {
   var productosEnCarrito: Int = 5
   println("Productos en el carrito: $productosEnCarrito")

   productosEnCarrito++ // Incremento en 1
   println("Productos en el carrito después de agregar uno: $productosEnCarrito")

   productosEnCarrito-- // Decremento en 1
   println("Productos en el carrito después de quitar uno: $productosEnCarrito")
}
```

En Kotlin, las cadenas de caracteres se manejan de manera flexible y eficiente. Kotlin ofrece una serie de características para trabajar con cadenas de forma conveniente y efectiva:

Representación de Cadena

Las cadenas de caracteres se representan utilizando el tipo `String`. Puedes declarar una cadena de varias maneras, incluyendo literales de cadena y funciones como `toString()` para convertir valores a cadenas.

```
val mensaje: String = "Hola, Kotlin!"
val numero: Int = 42
val numeroString: String = numero.toString()
```

Interpolación de Cadena

Kotlin admite la interpolación de cadenas, lo que significa que puedes insertar valores directamente dentro de cadenas usando la sintaxis `${}`. Esto es especialmente útil para combinar valores en cadenas.

```
val nombre = "Juan"
val edad = 30
val mensaje = "Mi nombre es $nombre y tengo $edad años."
```

Operaciones y Funciones de Cadena

Kotlin ofrece una variedad de funciones de extensión para manipular cadenas, como `length`, `toUpperCase()`, `toLowerCase()`, `substring()`, `replace()`, `trim()`, etc.

```
val texto = "   Hola, Mundo!   "
val longitud = texto.length // Devuelve 18
val minusculas = texto.toLowerCase() // Devuelve "   hola, mundo!   "
val mayusculas = texto.toUpperCase() // Devuelve "   HOLA, MUNDO!   "
val recortado = texto.trim() // Devuelve "Hola, Mundo!"
```

Comparación de Cadenas

Puedes comparar cadenas utilizando los operadores de igualdad (`==`) y desigualdad (`!=`), o las funciones `equals()` e `equalsIgnoreCase()` para comparaciones más flexibles.

```
val cadena1 = "Hola"
val cadena2 = "hola"
val sonIguales = cadena1 == cadena2 // Devuelve false
val igualesIgnoringCase = cadena1.equals(cadena2, ignoreCase = true) // Devuelve true
```

Raw Strings

Para crear cadenas que contengan caracteres de escape o formato especial sin necesidad de escaparlos, puedes usar las raw strings. Se definen entre triple comillas (`"""`).

```
val ruta = """C:\Users\Usuario\Documentos"""
```

Para eliminar los espacios al principio de las líneas, permitiéndonos formatear el literal libremente, podemos emplear el carácter **|** junto con el método `trimMargin()`.

String Templates

```
val i = 10
val name = "Joe"
println("i = $i")
println("Hola, mi nombre es $name y tiene ${s.length} caracteres")
```

Los literales de texto pueden funcionar como plantillas, conteniendo elementos (*placeholders*) que sean expresiones y sean evaluadas y sustituidas por su resultado en el literal. Estas expresiones se identifican mediante el signo dólar (**$**). Pueden ser únicamente el nombre de una variable o una expresión más compleja, en cuyo caso se encierran entre llaves (**{}**)

Puedes consultar una lista y uso de los métodos de la clase `String` en la documentación oficial

Para practicar

Ejercicio 1: Interpolación de Cadenas

Declarar variables para almacenar tu nombre y una calificación (8.5), luego mostrar un mensaje que combine ambos valores, por ejemplo: “Hola Pedro, tu calificación es de 8.5”.

☐ Solución Ejercicio 1: Interpolación de Cadenas

```
fun main() {
    val nombre: String = "Pedro"
    val calificacion: Double = 8.5

    val mensaje = "¡Hola $nombre! Tu calificación es $calificacion."
    println(mensaje)
}
```

PAIR Y TRIPLE

Es tan común el necesitar representar parejas o tríos de valores (coordenadas 2D/3D, entradas de un mapa/diccionario, devolución de múltiples valores por parte de un método o función,...) que Kotlin nos proporciona los tipos `Pair` y `Triple`.

Ambas son clases genéricas por lo que tendremos que especificar los tipos de cada uno de los elementos de la pareja/trío (Kotlin también los puede inferir)

Podemos acceder a sus valores mediante las propiedades `first`, `second`, `third`

Pair

Pair es una clase que puede contener dos elementos diferentes y los agrupa juntos en un objeto. Cada elemento se identifica como “first” (primero) y “second” (segundo).

```
val pair: Pair<String, Int> = Pair("Manzana", 5)
val fruta = pair.first // "Manzana"
val cantidad = pair.second // 5
```

También puedes usar la función de extensión to() para crear una instancia de Pair de manera más concisa:

```
val pair = "Manzana" to 5
```

Triple

Triple es similar a Pair, pero puede contener tres elementos en lugar de dos.

```
val triple: Triple<String, Int, Double> = Triple("Manzana", 5, 2.5)
val fruta = triple.first // "Manzana"
val cantidad = triple.second // 5
val precio = triple.third // 2.5
```

Ambas clases son útiles para casos en los que necesitas retornar o almacenar múltiples valores juntos y mantener su relación en tu código. Sin embargo, si necesitas agrupar más de tres elementos, considera usar estructuras más específicas y legibles, como clases personalizadas o colecciones.

Para practicar

Por supuesto, aquí tienes los ejercicios con enunciados más detallados:

Ejercicio 1: Creación de Pairs y Triples

Declara un `Pair` llamado `nombreEdad` que contenga el nombre de una persona como una cadena y su edad como un número entero. Crea un `Pair` llamado `punto` que represente las coordenadas de un punto en un plano bidimensional (x, y) utilizando números en coma flotante. Además, declara un `Triple` llamado `coordenadas` que almacene tres valores enteros que representen las coordenadas x, y, y z de un punto en un espacio tridimensional.

Luego, accede a los elementos dentro de cada `Pair` y `Triple` para imprimir la información correspondiente.

Ejercicio 2: Uso de Pairs y Triples en Cálculos

Declara un `Pair` llamado `dimensiones` que almacene las dimensiones de un rectángulo como dos números enteros: ancho y alto. Crea un `Triple` llamado `medidas` que represente las medidas de un objeto tridimensional (largo, ancho y alto) utilizando valores en coma flotante.

Luego, utiliza estos valores para calcular el área del rectángulo y el volumen del objeto tridimensional, y finalmente imprime los resultados.

Ejercicio 3: Uso de Pairs en una Colección

Crea una lista llamada `personas` que contenga tres Pairs. Cada Pair debe almacenar el nombre de una persona como cadena y su edad como número entero.

Utiliza un bucle `for` para iterar sobre la lista `personas` y, en cada iteración, desestructura el Pair para obtener el nombre y la edad de cada persona, luego imprime un mensaje que muestre esta información.

Estos ejercicios te ayudarán a entender cómo trabajar con `Pair` y `Triple` en Kotlin y cómo usarlos para agrupar y manipular datos relacionados. ¡Disfruta practicando!

Solución Ejercicio 1: Creación de Pairs y Triples

```
fun main() {
    val nombreEdad: Pair<String, Int> = Pair("Juan", 30)
    val punto: Pair<Double, Double> = Pair(3.5, 2.0)
    val coordenadas: Triple<Int, Int, Int> = Triple(1, 2, 3)

    println("Nombre: ${nombreEdad.first}, Edad: ${nombreEdad.second}")
    println("Punto x: ${punto.first}, Punto y: ${punto.second}")
    println("Coordenadas x: ${coordenadas.first}, y: ${coordenadas.second}, z:
${coordenadas.third}")
}
```

Solución Ejercicio 2: Uso de Pairs y Triples en Cálculos

```
fun main() {
    val dimensiones: Pair<Int, Int> = Pair(5, 7)
```

```
    val medidas: Triple<Double, Double, Double> = Triple(2.5, 3.0, 4.0)

    val area = dimensiones.first * dimensiones.second
    val volumen = medidas.first * medidas.second * medidas.third

    println("Área: $area")
    println("Volumen: $volumen")
}
```

Solución Ejercicio 3: Uso de Pairs y Triples en Colección

```
fun main() {
    val personas: List<Pair<String, Int>> = listOf(
        "Juan" to 30,
        "María" to 25,
        "Carlos" to 28
    )

    for ((nombre, edad) in personas) {
        println("$nombre tiene $edad años")
    }
}
```

ANY Y UNIT

Any y *Unit* son tipos especiales que tienen propósitos específicos en el sistema de tipos del lenguaje:

Any

`Any` es la superclase de todos los tipos **no nulos** en Kotlin. En otras palabras, cualquier objeto en Kotlin es de tipo `Any`. Puedes pensar en `Any` como el equivalente de `Object` en otros lenguajes de programación.

```
val valor: Any = 42
val texto: Any = "Hola, Kotlin!"
```

Sin embargo, debido a que `Any` es la superclase de todos los tipos, si asignas un valor a una variable de tipo `Any`, perderás la información de tipo específico y solo podrás acceder a las propiedades y métodos que son comunes a todos los tipos.

Unit

`Unit` es un tipo especial que se utiliza para representar la **ausencia de valor**, similar a lo que se conoce como `void` en otros lenguajes. En funciones, `Unit` se usa para indicar que la función no devuelve ningún valor concreto.

```
fun mostrarMensaje(): Unit {
```

```
    println("Hola desde la función")
}
```

En la práctica, no es necesario declarar explícitamente que una función devuelve `Unit`, ya que si no se especifica un tipo de retorno, se asume `Unit` de forma implícita.

Ambos tipos, `Any` y `Unit`, cumplen roles específicos en el sistema de tipos de Kotlin. `Any` es útil cuando deseas tratar con objetos genéricos sin preocuparte por su tipo concreto, y `Unit` es utilizado para indicar que una función no retorna un valor concreto, sino que solo realiza una acción.

Ejercicio 1: Uso de `Any` para Almacenar Datos Genéricos

Declara una variable llamada `dato1` que almacene un número entero, otra variable llamada `dato2` que almacene una cadena, y una tercera variable llamada `dato3` que almacene un número en coma flotante. Luego, imprime el contenido de cada variable.

Ejercicio 2: Uso de `Unit` para Indicar la Ausencia de Valor

Declara una función llamada `saludar` que imprima "¡Hola, Kotlin!" en la consola. Utiliza el tipo de retorno `Unit` para indicar que la función no retorna un valor concreto. Luego, llama a la función desde `main()`.

Ejercicio 3: Uso de `Any` en Colecciones Genéricas

Crea una lista llamada `elementos` que contenga valores de diferentes tipos, como enteros, cadenas y números en coma flotante. Utiliza el tipo `Any` para que puedas almacenar cualquier tipo de valor. Luego, itera sobre la lista e imprime el tipo y el valor de cada elemento.

Solución Ejercicio 1: Uso de `Any` para Almacenar Datos Genéricos

```
fun main() {
    val dato1: Any = 42
    val dato2: Any = "Hola, Kotlin!"
    val dato3: Any = 3.14

    println("Dato 1: $dato1")
    println("Dato 2: $dato2")
    println("Dato 3: $dato3")
}
```

Solución Ejercicio 2: Uso de `Unit` para Indicar la Ausencia de Valor

```
fun main() {
    saludar()
}

fun saludar(): Unit {
    println("¡Hola, Kotlin!")
```

```
}
```

Solución Ejercicio 3: Uso de `Any` en Colecciones Genéricas

```
fun main() {
    val elementos: List<Any> = listOf(42, "Hola", 3.14)

    for (elemento in elementos) {
        println("Tipo: ${elemento::class.simpleName}, Valor: $elemento")
    }
}
```

EJERCICIOS

- **Ejercicio 1.** Declara dos variables a y b de tipo Double y asígnales un valor. Calcula la media y almacena el valor resultante en una constante denominada media.
- **Ejercicio 2.** Una temperatura expresada en ºC puede ser convertida a ºF multiplicando por 1.8 y sumando 32. En este caso, vamos a hacer el contrario: convertir una temperatura de ºF a ºC. Declara una constante denominada fahrenheit de tipo Double y asígnale un valor. Calcula la temperatura correspondiente en ºC y almacena el resultado en una constante denominada celsius.
- **Ejercicio 3.** Imagina que las casillas de un tablero de ajedrez se numeran de izquierda a derecha, de arriba a abajo, siendo 0 la casilla de arriba-izquierda y 63 la casilla de abajo-derecha. Las filas se numeran de 0 a 7 de arriba a abajo y, las columnas, de 0 a 7 de izquierda a derecha. Declara una constante denominada posicion y asígnale un valor entre 0 y 63. Calcula los números correspondientes a la fila y la columna y almacena los resultados en dos constantes denominadas fila y columna.
- **Ejercicio 4.** Un círculo tiene 2π radianes, correspondiendo con 360º. Pide al usuario que introduzca un valor de grados (de tipo Double) y calcula el valor correspondiente en radianes. Almacena el resultado en una constante denominada radianes e imprímela.
- **Ejercicio 5.** Crea dos constantes denominadas nombre y apellidos e inicializalas con el nombre y apellidos introducidos por el usuario. Crea una constante denominada nombreCompleto cuyo valor sea la concatenación de las dos anteriores separadas por un espacio. Usando templates, crea una constante denominada misDatos que emplee la constante anterior nombreCompleto para producir un texto con tu nombre similar al siguiente: “Hola, tu nombre es John Doe”.
- **Ejercicio 6.** ¿Qué es incorrecto del siguiente código? Corrígelo.

```
val name = "Joe"
name += " Doe"
```

- **Ejercicio 7.** Solicita al usuario que introduzca el día, mes y año de su fecha de cumpleaños. Declara una constante de tipo Triple

que contenga tres enteros: día, mes y año e inicialízala con la fecha introducida previamente. Usa string templates para componer el texto siguiente a partir de los datos almacenados en el Triple: “Tu cumpleaños es el X del X de X” donde sustituyas las X por los valores correspondientes.

CONTROL DE FLUJO

Los condicionales en Kotlin te permiten tomar decisiones en tu código y ejecutar diferentes bloques de código según si una o varias condiciones son verdaderas o falsas. Los condicionales más comunes en Kotlin son `if`, `else if` y `else`. Aquí tienes una explicación de cómo funcionan:

`if` simple

El condicional `if` se utiliza para ejecutar un bloque de código si la condición especificada es verdadera.

```
val edad = 18
if (edad >= 18) {
   println("Eres mayor de edad")
}
```

`if` – `else`

Puedes usar un bloque `else` después de un `if` para especificar qué hacer si la condición es falsa.

```
val edad = 15
if (edad >= 18) {
   println("Eres mayor de edad")
} else {
   println("Eres menor de edad")
}
```

`if` – `else if` – `else`

Si tienes múltiples condiciones, puedes usar bloques `else if` para evaluar otras condiciones después del primer `if`, antes de llegar al `else` final.

```
val puntaje = 85
if (puntaje >= 90) {
   println("Excelente")
} else if (puntaje >= 80) {
   println("Buen trabajo")
} else if (puntaje >= 70) {
   println("Aprobado")
} else {
   println("No aprobado")
```

```
}
```

Operador Ternario (Expresión condicional):

En lugar de utilizar `if` y `else` para asignar valores en una expresión, puedes usar el operador ternario (`? :`).

```
val edad = 20
val mensaje = if (edad >= 18) "Eres mayor de edad" else "Eres menor de edad"
println(mensaje)
```

`when` (Reemplazo de `switch`):

Kotlin reemplaza la instrucción `switch` con la expresión `when`, que es más poderosa y versátil. Puedes usar `when` con o sin argumento y con múltiples condiciones.

```
val dia = 3
when (dia) {
   1 -> println("Lunes")
   2 -> println("Martes")
   in 3..5 -> println("Día de la semana")
   else -> println("Fin de semana")
}
```

Recuerda que los condicionales en Kotlin son una parte fundamental de la lógica de programación y te permiten tomar decisiones basadas en diferentes situaciones.

BUCLES

En Kotlin, tienes varias opciones para implementar bucles que te permiten ejecutar una porción de código repetidamente. Los bucles más comunes son `for`, `while` y `do-while`.

Rangos

Antes de introducir los bucles definidos mediante la sentencia for, necesitamos conocer los datos de tipo rango, que nos permiten representar secuencias de números enteros.

Podemos emplear la siguiente notación: .. para definir rangos cerrados donde los límites superior e inferior del rango están incluídos

```
   val rango = (0..5)
   var i = rango.first
   while (i <= rango.last) {
      println(i++)
   }
```

Bucle `for`:

El bucle `for` se utiliza para recorrer una secuencia (como un rango numérico, una colección, una matriz, etc.) y ejecutar un bloque de código para cada elemento en la secuencia.

```
for (i in 1..5) {
    println("Número: $i")
}

val nombres = listOf("Juan", "María", "Carlos")
for (nombre in nombres) {
    println("Hola, $nombre")
}
```

Adicionalmente, podemos emplear la palabra reservada `until` para definir **rangos medio abiertos**, donde el límite inferior está incluído en el rango pero el el límite superior no

```
val rango = 0 until 5
var i = rango.first
while (i <= rango.last) {
    println(i++)
}
```

Bucle `while`:

El bucle `while` ejecuta un bloque de código siempre que una condición especificada sea verdadera. La condición se verifica antes de cada iteración.

```
var contador = 0
while (contador < 5) {
    println("Iteración $contador")
    contador++
}
```

Bucle `do-while`:

El bucle `do-while` es similar al `while`, pero la condición se verifica después de cada iteración. Esto asegura que el bloque de código se ejecute al menos una vez, incluso si la condición es inicialmente falsa.

```
var intentos = 0
do {
    println("Intento número $intentos")
    intentos++
} while (intentos < 3)
```

Bucle `for` con índices:

Además de recorrer elementos en una secuencia, puedes utilizar un bucle `for` para acceder a los índices de los elementos en una secuencia utilizando `indices`.

```
val numeros = arrayOf(10, 20, 30, 40)
for (indice in numeros.indices) {
    println("Número en el índice $indice es ${numeros[indice]}")
}
```

`break` y `continue`:

Puedes usar `break` para salir de un bucle antes de que se complete, y `continue` para saltar a la siguiente iteración del bucle.

```
for (i in 1..10) {
    if (i == 5) {
        break
    }
    println(i)
}

for (i in 1..5) {
    if (i == 3) {
        continue
    }
    println(i)
}
```

SOLUCIONES EJERCICIOS

- **Ejercicio 1.** Corrige el siguiente código:

```
val firstName = "Joe"

if (firstName == "Howard") {
    val lastName = "Lucas"
} else if (firstName == "Ray") {
    val lastName = "Wenderlich"
}

val fullName = firstName + " " + lastName
```

Solución Ejercicio 1

```
val firstName = "Joe"
val lastName = "" // <----- necesitamos declarar la variable
if (firstName == "Howard") {
    val lastName = "Lucas"
} else if (firstName == "Ray") {
    val lastName = "Wenderlich"
}

val fullName = firstName + " " + lastName
println($fullName) // <----- mostramos el resultado (no es un error)
```

- **Ejercicio 2.** Solicita del usuario los coeficientes a, b y c para calcular las soluciones de la ecuación de segundo grado correspondiente. Ten en cuenta que el diferente número de soluciones (0, 1 ó 2) dependiendo del valor del discriminante. Si necesitas “refrescar” tus matemáticas, puedes consultar el siguiente enlace.

Solución Ejercicio 2

```
val a = readln().toDouble()
val b = readln().toDouble()
val c = readln().toDouble()

if (a == 0.0) println("No es una ecuación de segundo grado")
else {
   val d = b.pow(2) - 4*a*c // discriminante
   when {
      d == 0.0 -> {
         // solución única
         print("Solución única -> ")
         println("X1 = X2 = ${-b/(2*a)}")
      }
      d < 0 -> {
         // soluciones imaginarias (no reales)
         print("Soluciones imaginarias -> ")
         val img = sqrt(-d)/(2*a)
         println("X1 = ${-b/(2*a)} + i$img; X2 = ${-b/(2*a)} - i$img")
      }
      d > 0 -> {
         // soluciones reales
         print("Soluciones reales -> ")
         val rootD = sqrt(d)
         println("X1 = ${(-b + rootD)/(2*a)}; X2 = ${(-b - rootD)/(2*a)}")
      }
   }
}
```

- **Ejercicio 3.** Dado un mes (representado con un String en minúsculas) y el año actual (representado como un Int) introducidos por el usuario, calcula el número de días del mes. Recuerda que en los años bisiestos “febrero” tiene 29 días. Años bisiestos son aquellos que son múltiplos de 4 pero no de 100 y los que son múltiplos de 400

Solución Ejercicio 3

```
val month = readln()
val year = readln().toInt()

val days = when (month) {
   "enero", "marzo", "mayo", "julio", "agosto", "octubre", "diciembre" -> 31
   "abril", "junio", "septiembre", "noviembre" -> 30
   else -> if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) 31 else 30
}

println(if (days > 0) "$month de $year tiene $days días" else "$month no es un mes válido")
```

- **Ejercicio 4.** Dadas las coordenas x, y (de tipo Int) de un punto en el espacio, usa una expresión when para imprimir alguno de los siguientes mensajes:

```
si x=0 e y=0, "Punto en el origen"
si x≠0 e y=0, "Punto sobre el eje-X"
si x=0 e y≠0, "Punto sobre el eje-Y"
en cualquier otro caso, "Punto en la posición (x,y)"
```

Solución Ejercicio 4

```
val x = readln().toInt()
val y = readln().toInt()

println(when {
   x == 0 && y == 0 -> "Punto en el origen"
   x != 0 && y == 0 -> "Punto sobre el eje-X"
   x == 0 && y != 0 -> "Punto sobre el eje-y"
   else -> "Punto en la posición ($x, $y)"
})
```

- **Ejercicio 5.** Imprime una tabla con las 10 primeras potencias de 2.

Solución Ejercicio 5

```
repeat(10) { println(2.0.pow(it).toInt()) }
```

- **Ejercicio 6.** Dado un número n mayor o igual 1, calcula el término n de la secuencia de Fibonacci.

  Solución Ejercicio 6

```
var n = readln().toInt()

var res = 1
if(n >= 3) {
   var f1 = 1; var f2 = 1
   while(n-- > 2) {
      res = f1 + f2; f1 = f2; f2 = res
   }
}
println(res)
```

- **Ejercicio 7.** Dado un número n, calcula el factorial de dicho número n.

Solución Ejercicio 7

```
val n = readln().toInt()
var fact = 1
for(i in 2..n) fact *= i
println("El factorial de $n es $fact")
```

- **Ejercicio 8.** Dado un número n, imprime todos los números primos hasta n.

  Solución Ejercicio 8

```
val n = readln().toInt()
for(i in 1..n) {
   var prime = true
   for(j in 2..i/2)
      if (i % j == 0) {
         prime = false
         break
      }
   if (prime) println(i)
}
```

- **Ejercicio 9.** Escribe un programa que solicite al usuario la introdución de valores numéricos enteros (uno por línea) hasta que se introduzca el valor “00”. A continuación, se mostrará la suma, la media, el mayor y el menor de los valores introducidos

  Solución Ejercicio 9

```
var maxValue = Int.MIN_VALUE
var minValue = Int.MAX_VALUE
var sum = 0; var cont = 0

while(true) {
   var snum = readln()
   if (snum == "00") break

   val num = snum.toInt()
   maxValue = max(maxValue, num)
   minValue = min(minValue, num)
   sum += num
   cont++
}

if (cont > 0)
   println("Suma: $sum\nMáximo: $maxValue\nMínimo: $minValue\nMedia:
${sum.toDouble() / cont}")
```

FUNCIONES

Las funciones son parte central de muchos lenguajes de programación. En esencia, nos permiten definir bloques de código que realizan una determinada tarea. Dichos bloques podrán ser invocados desde diferentes puntos de nuestra aplicación cada vez que lo necesitemos.

Declaración de funciones

Para declarar una función en Kotlin, se utiliza la palabra clave `fun`, seguida del nombre de la función y sus parámetros. El tipo de retorno se especifica después de los parámetros con `: TipoDeRetorno`. Para definir dichas funciones, usamos la palabra reservada fun seguida por el nombre de la misma. A continuación, unos parántesis nos permitirán definir su lista de parámetros (ninguno en este caso). Por último, el bloque de código de la función encerrado entre llaves

Ejemplo de declaración de una función simple:

```
fun nombreFunc(lista_params): tipo_retorno {
   bloque_codigo
}
```

Una vez declarada, para ejecutar dicha función, no tenemos más que invocar su nombre desde algún punto de nuestro programa:

```
nombreFunc(lista_params)
```

Parámetros

Los parámetros de una función pueden ser opcionales o requeridos. Puedes proporcionar valores por defecto para los parámetros, lo que permite la llamada de funciones con menos argumentos.

Ejemplo con parámetro opcional y valor por defecto:

```
fun saludar(nombre: String = "Usuario") {
   println("Hola, $nombre")
}
```

Así, en caso de que omitamos dicho argumento en la llamada al método, se utilizará el valor por defecto

Valor de retorno

Nuestras funciones podrán retornar un valor que podremos asignar a una variable (o constante) o usar en una expresión.

Para ello, en la declaración de la función, tendremos que indicar el tipo del valor devuelto añadiendo **:TIPO** después de la lista de parámetros;

```
fun multipleOf(value: Int, mult:Int = 1): Int {
   return value * mult
}

println("2 * 4 = ${multipleOf(2, 4)}")
```

Dentro del bloque de código, emplearemos la sentencia `return <VALUE>` para salir de la función y devolver el valor correspondiente.

Funciones dcomo argumentos

Kotlin admite funciones de orden superior, que son funciones que pueden tomar otras funciones como argumentos o devolverlas como resultado. Esto facilita la programación funcional y la escritura de código más conciso y expresivo.

Ejemplo de función de orden superior:

```
fun operate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}

val sumaResult = operate(5, 3) { a, b -> a + b }
```

Funciones anidadas

Puedes definir funciones dentro de otras funciones, lo que ayuda a encapsular la lógica y evitar la contaminación del espacio de nombres global.

Ejemplo de función anidada:

```
fun calcularSumaPromedio(lista: List<Int>): Pair<Int, Double> {
    fun suma(): Int = lista.sum()
    fun promedio(): Double = lista.average()

    return Pair(suma(), promedio())
}
```

Funciones con tipo de retorno implícito

Si una función devuelve un valor, el tipo de retorno puede omitirse si el compilador puede inferirlo.

Ejemplo de tipo de retorno implícito:

```
fun doble(num: Int) = num * 2 // El tipo de retorno Int se infiere automáticamente.
```

EJERCICIOS

- **Ejercicio 1.** Crea una función denominada isNumberDivisible que acepte dos valores enteros como parámetros: un número y un divisor. La función nos indicará si el número es divisible por el divisor o no.

- **Ejercicio 2.** Empleando la función anterior, crea una nueva función denominada isPrime que reciba un número entero y nos diga si es

primo o no (un número es primo si sólo es divisible por 1 y por si mismo).

- **Ejercicio 3.** Empleando la función anterior, imprime todos los números primos entre 1 y 100.

- **Ejercicio 4.** Crea una función denominada cuadrado que acepte los siguientes parámetros: alto, ancho y un carácter. La función dibujará el perímetro de u n rectángulo de las dimensiones indicadas utilizando el carácter pasado como argumento. En caso de que alguna de las dimensiones sea menor que 1, no se imprimirá nada.

- **Ejercicio 5.** Las funciones recursivas son aquellas que se llaman a sí mismas. Crea una función recursiva denominada fibonacci que nos calcule cualquier término de la secuencia de forma recursiva. Para ello, ten en cuenta lo siguiente:

- el valor de la secuencia es 0 para cualquier término menor que 1
- el término n de la secuencia será: fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

- **Ejercicio 6.** Empleando la función anterior, imprime los primeros 10 términos de la secuencia de Fibonacci.

TRATAMIENTO DE NULOS

Hasta ahora, todas las variables y constantes que hemos venido usando tenían un valor concreto. Sin embargo, en ocasiones, es necesario representar la ausencia de un valor.

Para ello podemos emplear un valor especial. Es lo que se conoce como valores centinela, es decir, valores empleados para representar la ausencia de un valor en una variable.

Los lenguajes de programación suelen emplear el identificador `null` para representar esta ausencia de valor. Mediante su uso es posible indicar que una variable o constante no está inicializada.

Esto tiene la **ventaja** de no emplear valores centinela pero tiene un gran problema, la necesidad de estar constantemente chequeando si la variable en cuestión tiene asignado un valor antes de tratar de acceder a alguno de los atributos o métodos del supuesto objeto referenciado (por ejemplo, los continuos `if (var != null)` de Java). De lo contrario, se producirán estrepitosos **errores en tiempo de ejecución**. Y esto es lo más grave, que no son detectables en tiempo de compilación.

Kotlin resuelve el problema introduciendo un nuevo conjunto de tipos, los **nullable types** o **tipos anulables**. Empleando un tipo **no-nulo** (como

todos los vistos hasta ahora) nos aseguramos que cualquier variable tendrá siempre un valor. Por otro lado, empleando los tipos **anulables**, podremos tener variables no asignadas (=null) pero deberemos manejar dicha situación.

Tipos principales

En Kotlin, hay dos tipos principales para manejar la posibilidad de valores nulos:

Tipos no nulos

Cuando declaras una variable sin indicar que puede ser nula, se asume que el valor no puede ser nulo. Por ejemplo: `val nombre: String = "Juan"` significa que `nombre` nunca será nulo.

Tipos que pueden ser nulos

Si deseas que una variable pueda contener un valor nulo, debes declararla como un tipo que permite nulos. Esto se hace añadiendo un signo de interrogación (`?`) después del tipo de dato. Por ejemplo: `var apellido: String? = null` significa que `apellido` puede contener un valor nulo o una cadena de caracteres.

Trabajar con nulos

Kotlin proporciona varias formas seguras de trabajar con valores nulos:

- **Operador de verificación de nulos (`?.`)**: Permite llamar a un método o acceder a una propiedad en un objeto solo si el objeto no es nulo. Si el objeto es nulo, la expresión completa devuelve nulo.

```
val longitud: Int? = texto?.length
```

- **Operador de elvis (`?:`)**: Se utiliza para proporcionar un valor de respaldo en caso de que una expresión sea nula.

```
val resultado = valorNoNulo ?: valorPorDefecto
```

- **Operador de llamada segura (`let`)**: Permite ejecutar un bloque de código solo si el valor no es nulo.

```
valorNoNulo?.let {
   // Realizar operaciones solo si valorNoNulo no es nulo
}
```

- **Operador de lanzamiento seguro (`as?`)**: Se utiliza para intentar realizar una conversión de tipo, y si la conversión falla, devuelve nulo en lugar de arrojar una excepción.

```
val numero: Int? = texto as? Int
```

- **Operador de verificación no nula (`!!`)**: Se utiliza para indicar que estás seguro de que una variable no es nula. Sin embargo, usarlo incorrectamente puede provocar excepciones de puntero nulo, por lo que se debe utilizar con precaución.

```
val valor: String? = obtenerValor()
val longitud = valor!!.length // Puede generar una excepción si valor es nulo
```

En resumen, Kotlin proporciona una forma segura y robusta de manejar los valores nulos, lo que ayuda a reducir errores en tiempo de ejecución relacionados con punteros nulos y facilita la escritura de código más seguro y legible.

EJERCICIOS

- **Ejercicio 1.** Crea una variable de tipo String anulable denominada myFavouriteSong. Si tienes una canción favorita, asígnasela. Si no tienes, o tienes más de una, establece su valor a null. Empleando smart casts imprime su contenido o el texto “No tengo una canción favorita”
- **Ejercicio 2.**

Crea una función denominada `divideIfWhole` que devuelva el número de veces que un número entero puede ser dividido por otro número entero sin resto. La función debe devolver **null** si el número no es divisible. A continuación, creas una nueva función denominada `checkDivisible` que invoque a la función anterior y muestre el resultado. Utiliza smart casts para distinguir los dos casos: si el primer número es divisible por el segundo debe indicar “Sí, es divisible XXX veces” o, en caso de que no lo sea, “No es divisible :[”. Por ejemplo:

```
checkDivisible(200, 2). Debe imprimir "Sí, es divisible 3 veces"
checkDivisible(200, 3). Debe imprimir "No es divisible :["
```

- **Ejercicio 3.** Reescribe la función `checkDivisible` para que emplee el operador Elvis. En este caso, el mensaje siempre será “Es divisible XXX veces”, siendo XXX el número de veces que podemos realizar la división entera (>=0)

Capitulo 2

COLECCIONES

ARRAYS

Los arrays en kotlin se corresponden con el tipo de array estático disponible en Java. Son estructuras tipadas y permiten almacenar múltiples valores del mismo tipo en espacios contiguos de la memoria, lo que da lugar a altos niveles de eficiencia en determinadas operaciones y procedimientos.

Un array es una **colección ordenada de elementos del mismo tipo**. Los arrays en Kotlin se utilizan para almacenar múltiples valores en una sola estructura de datos.

Declaración de un array

Puedes declarar un array usando la función `arrayOf()` o la función `arrayOfNulls()` para arrays de objetos que pueden contener nulos. Los arrays en Kotlin son **inmutables** por defecto, lo que significa que no puedes cambiar su tamaño después de crearlos. Para un array mutable, puedes usar `mutableListOf()` o `ArrayList`.

Ejemplo de declaración de un array de enteros:

```
val numeros: Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

Acceso a elementos

Puedes acceder a los elementos de un array utilizando el operador de indexación `[índice]`, donde `índice` es la posición del elemento (comenzando desde 0).

```
val primerNumero = numeros[0] // Acceder al primer elemento
```

Tamaño del array

Puedes obtener el tamaño de un array utilizando la propiedad `size`.

```
val tamano = numeros.size // Obtener el tamaño del array
```

Iteración a través de un array

Puedes iterar a través de los elementos de un array utilizando un bucle `for` o funciones de orden superior como `forEach`.

Ejemplo de iteración con un bucle `for`:

```
for (numero in numeros) {
    println(numero)
```

```
}
```

Ejemplo de iteración con `forEach`:

```
numeros.forEach { numero ->
    println(numero)
}
```

Array de tipos primitivos

En Kotlin, los arrays pueden contener tipos primitivos como `Int`, `Float`, etc. sin necesidad de usar clases envolventes como en Java.

```
val enteros: IntArray = intArrayOf(1, 2, 3, 4, 5)
```

Array de objetos

Los arrays en Kotlin pueden contener objetos de cualquier tipo, incluidos los objetos personalizados.

Ejemplo de array de objetos personalizados:

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

val personas: Array<Persona> = arrayOf(
    Persona("Juan", 25),
    Persona("María", 30)
)
```

El método main

A partir de la versión 1.3 de Kotlin, la función main tiene un parámetro opcional denominado args de tipo Array

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

Cuando ejecutamos un programa Kotlin desde la línea de comandos, podemos pasarle argumentos que se almacenarán como elementos de dicho array de forma que tengamos acceso a ellos desde el propio programa.

En el caso de los IDE's, normalmente nos permiten configurar la ejecución del programa de forma que podamos enviarle los argumentos que deseemos. En el caso concreto de IntelliJ, desde la configuración del projecto, podemos establecer el valor del atributo Program arguments.

LISTAS

En Kotlin, una lista es una colección de elementos que se **almacenan de manera ordenada** y se pueden acceder mediante un **índice**. Las listas son estructuras de datos **flexibles** y se utilizan comúnmente para almacenar conjuntos de elementos que pueden cambiar en tamaño y contenido. Aquí tienes una descripción general de cómo se trabaja con listas en Kotlin:

Como en Java, el tipo `List` en Kotlin es un **interfaz** que tiene concreciones como la clase `ArrayList` (no existe la clase `LinkedList` en Kotlin). Los arrays suelen ser más eficientes que las listas en términos de rendimiento puro, pero las listas tienen la ventaja de ser estructuras dinámicas, aumentando su capacidad en función de las necesidades.

Declaración de una lista

Puedes declarar una lista en Kotlin utilizando la función `listOf()` para crear una lista inmutable (no modificable) o `mutableListOf()` para crear una lista mutable (modifiable).

Ejemplo de declaración de una lista inmutable:

```
val numeros: List<Int> = listOf(1, 2, 3, 4, 5)
```

Ejemplo de declaración de una lista mutable:

```
val nombres: MutableList<String> = mutableListOf("Juan", "María", "Carlos")
```

Acceso a elementos

Puedes acceder a los elementos de una lista mediante el operador de indexación `[índice]`, donde `índice` es la posición del elemento (comenzando desde 0).

Ejemplo de acceso a un elemento en una lista:

```
val primerNumero = numeros[0] // Acceder al primer elemento
```

Tamaño de la lista

Puedes obtener el tamaño de una lista utilizando la propiedad `size`.

```
val tamano = nombres.size // Obtener el tamaño de la lista
```

Iteración a través de una lista

Puedes iterar a través de los elementos de una lista utilizando bucles `for` o funciones de orden superior como `forEach`.

Ejemplo de iteración con un bucle `for`:

```
for (nombre in nombres) {
   println(nombre)
}
```

Ejemplo de iteración con `forEach`:

```
nombres.forEach { nombre ->
   println(nombre)
}
```

Operaciones con listas

Las listas en Kotlin proporcionan una variedad de funciones útiles para realizar operaciones como agregar elementos, eliminar elementos, buscar elementos, etc.

Ejemplo de agregar un elemento a una lista mutable:

```
nombres.add("Luis")
```

Ejemplo de eliminar un elemento de una lista mutable:

```
nombres.remove("Carlos")
```

Ejemplo de buscar un elemento en una lista:

```
val indice = nombres.indexOf("María")
if (indice != -1) {
   println("María se encuentra en la posición $indice")
} else {
   println("María no está en la lista")
}
```

En resumen, en Kotlin, las listas son una parte fundamental de las colecciones y te permiten almacenar y manipular conjuntos de elementos de manera eficiente. Pueden ser tanto inmutables como mutables, y ofrecen una amplia gama de funciones para trabajar con los datos de la lista de manera conveniente.

NULABILIDAD

Trabajando con arrays, listas u otras colecciones, deberemos tener en consideración el tratamiento de nulos. Ten en cuenta que tanto la colección como los elementos podrán estar definidos como tipos anulables.

Por ejemplo, la siguiente será una lista de tipo anulable:

```
var nullableList: List<Int>? = listOf(1, 2, 3, 4)
nullableList = null
println(nullableList)
```

Sin embargo, en el siguiente caso crearemos una lista que acepte nulos. Es decir, el tipo de los elementos es anulable

```
val listOfNulls = mutableListOf<Int?>(1, 2, 3, null, 4)
listOfNulls[0] = null
println(listOfNulls)
```

MAPA

En Kotlin, un mapa es una colección de pares clave-valor donde cada clave es única. Los mapas son estructuras de datos extremadamente útiles para almacenar y recuperar datos relacionados. En Kotlin, puedes crear mapas utilizando la interfaz `Map` para mapas inmutables o `MutableMap` para mapas mutables. Aquí tienes una descripción general de cómo se utilizan los mapas en Kotlin:

Declaración de un mapa

Puedes declarar un mapa en Kotlin utilizando la función `mapOf()` para crear un mapa inmutable o `mutableMapOf()` para crear un mapa mutable. Los mapas inmutables no pueden ser modificados después de su creación, mientras que los mapas mutables permiten agregar, modificar y eliminar elementos.

Ejemplo de declaración de un mapa inmutable:

```
val frutas: Map<String, Int> = mapOf("Manzana" to 2, "Banana" to 3, "Naranja" to 4)
```

Ejemplo de declaración de un mapa mutable:

```
val edades: MutableMap<String, Int> = mutableMapOf("Juan" to 30, "María" to 25, "Carlos" to 35)
```

Acceso a elementos

Puedes acceder a los valores en un mapa utilizando la clave correspondiente.

Ejemplo de acceso a un valor en un mapa inmutable:

```
val cantidadManzanas = frutas["Manzana"] // Devuelve 2
```

Iteración a través de un mapa

Puedes iterar a través de los elementos de un mapa utilizando un bucle `for` o funciones de orden superior como `forEach`.

Ejemplo de iteración con un bucle `for`:

```
for ((fruta, cantidad) in frutas) {
    println("Tenemos $cantidad $fruta(s)")
}
```

Ejemplo de iteración con `forEach`:

```
frutas.forEach { (fruta, cantidad) ->
    println("Tenemos $cantidad $fruta(s)")
}
```

Operaciones con mapas

Los mapas en Kotlin ofrecen una variedad de funciones útiles para realizar operaciones como agregar elementos, eliminar elementos y verificar si una clave está presente en el mapa.

Ejemplo de agregar un elemento a un mapa mutable:

```
edades["Luis"] = 40
```

Ejemplo de eliminar un elemento de un mapa mutable:

```
edades.remove("Carlos")
```

Ejemplo de verificar si una clave está presente en un mapa:

```
if ("Juan" in edades) {
    println("Juan está en la lista de edades")
}
```

MAPA

En Kotlin, un mapa es una colección de pares clave-valor donde cada clave es única. Los mapas son estructuras de datos extremadamente útiles para almacenar y recuperar datos relacionados. En Kotlin, puedes crear mapas utilizando la interfaz `Map` para mapas inmutables o `MutableMap` para mapas mutables. Aquí tienes una descripción general de cómo se utilizan los mapas en Kotlin:

Declaración de un mapa

Puedes declarar un mapa en Kotlin utilizando la función `mapOf()` para crear un mapa inmutable o `mutableMapOf()` para crear un mapa mutable. Los mapas inmutables no pueden ser modificados después de su creación, mientras que los mapas mutables permiten agregar, modificar y eliminar elementos.

Ejemplo de declaración de un mapa inmutable:

```
val frutas: Map<String, Int> = mapOf("Manzana" to 2, "Banana" to 3, "Naranja" to 4)
```

Ejemplo de declaración de un mapa mutable:

```
val edades: MutableMap<String, Int> = mutableMapOf("Juan" to 30, "María" to 25, "Carlos" to 35)
```

Acceso a elementos

Puedes acceder a los valores en un mapa utilizando la clave correspondiente.

Ejemplo de acceso a un valor en un mapa inmutable:

```
val cantidadManzanas = frutas["Manzana"] // Devuelve 2
```

Iteración a través de un mapa

Puedes iterar a través de los elementos de un mapa utilizando un bucle `for` o funciones de orden superior como `forEach`.

Ejemplo de iteración con un bucle `for`:

```
for ((fruta, cantidad) in frutas) {
    println("Tenemos $cantidad $fruta(s)")
}
```

Ejemplo de iteración con `forEach`:

```
frutas.forEach { (fruta, cantidad) ->
    println("Tenemos $cantidad $fruta(s)")
}
```

Operaciones con mapas

Los mapas en Kotlin ofrecen una variedad de funciones útiles para realizar operaciones como agregar elementos, eliminar elementos y verificar si una clave está presente en el mapa.

Ejemplo de agregar un elemento a un mapa mutable:

```
edades["Luis"] = 40
```

Ejemplo de eliminar un elemento de un mapa mutable:

```
edades.remove("Carlos")
```

Ejemplo de verificar si una clave está presente en un mapa:

```
if ("Juan" in edades) {
    println("Juan está en la lista de edades")
}
```

LAMBDA

En Kotlin, una lambda es una **función anónima** que se puede utilizar como un objeto. Las lambdas son una característica poderosa y flexible del lenguaje que te permite expresar y pasar comportamientos o bloques de código como argumentos a otras funciones. Las lambdas son similares a las funciones, pero no tienen un nombre definido y se pueden crear de manera concisa en el lugar donde se necesiten.

Una lambda en Kotlin generalmente tiene la siguiente sintaxis básica:

```
{ parámetros -> cuerpo de la lambda }
```

- `parámetros`: Son los parámetros que toma la lambda. Pueden ser opcionales y omitirse si la lambda no requiere parámetros.
- `cuerpo de la lambda`: Es el código que se ejecuta cuando se llama a la lambda. Puede ser una expresión única o un bloque de código.

Ejemplo de una lambda que toma dos parámetros y devuelve su suma:

```
val suma: (Int, Int) -> Int = { a, b -> a + b }
```

Las lambdas se utilizan comúnmente en Kotlin para:

1. **Pasar comportamientos como argumentos de funciones**:
    - Puedes pasar una lambda como argumento a una función de orden superior. Esto es especialmente útil para funciones como `map`, `filter`, `forEach`, etc.

Ejemplo de uso de lambda con `map` para transformar una lista:

```
val numeros = listOf(1, 2, 3, 4, 5)
val cuadrados = numeros.map { numero -> numero * numero }
```

2. **Definir comportamientos ad hoc**:
    - Puedes crear lambdas en el lugar donde se necesiten, lo que facilita la definición de comportamientos ad hoc en tu código.

Ejemplo de uso de lambda para encontrar el número máximo en una lista:

```
val numeros = listOf(10, 5, 20, 8, 15)
val maximo = numeros.maxBy { numero -> numero }
```

3. **Usar funciones de orden superior**:
    - Las lambdas son fundamentales para trabajar con funciones de orden superior, que son funciones que toman otras funciones como argumentos.

Ejemplo de función de orden superior que utiliza una lambda:

```
fun operar(a: Int, b: Int, operacion: (Int, Int) -> Int): Int {
    return operacion(a, b)
}

val suma = operar(5, 3) { a, b -> a + b }
```

En resumen, una lambda en Kotlin es una función anónima que se puede utilizar para expresar y pasar comportamientos de manera concisa. Esto hace que el código sea más legible y permite una mayor flexibilidad al trabajar con funciones de orden superior y colecciones. Las lambdas son una característica clave de la programación funcional en Kotlin.

Lambdas y Colecciones

Las funciones lambda se emplean con frecuencia para operar sobre fuentes o colecciones de datos.Por ejemplo, en unidades anteriores vimos como ordenar una colección empleando el método **sort()**. Alternativamente, las colecciones disponen de otro método, **sortedWith()** que recibe un objeto de tipo **Comparator** que permite establecer el criterio de comparación. Este objeto lo podemos crear mediante la función **conpareBy()**, la cual acepta un lambda que devuelve el criterio para evaluar el orden de los elementos de la colección.En el siguiente ejemplo se ordena en función de la longitud de los nombres emplando dos comparadores diferentes. Uno define un lambda que devuelve la longitud del nombre y, el otro, la longitud en negativo, de forma que se invierta el orden

```
val names = arrayOf("Brian", "Paul", "Kevin", "Aaron", "Roy", "Russell")

val namesByLenAsc = names.sortedWith(compareBy { it.length })
println(namesByLenAsc)

val namesByLenDesc = names.sortedWith(compareBy { -it.length })
println(namesByLenDesc)
```

En Kotlin, las colecciones implementan diversas funcionalidades asociadas frecuentemente a la programación funcional. Estas funcionalidades están relacionadas con la aplicación de operaciones sobre la colección commo, por ejemplo, transformaciones o filtrado de ciertos elementos.

PROGRAMACIÓN ORIENTADA A OBJETOS

CLASES Y OBJETOS

En Kotlin, puedes crear clases y objetos de manera sencilla. Aquí te muestro cómo hacerlo:

Declaración de una clase

Para declarar una clase en Kotlin, utiliza la palabra clave `class`, seguida del nombre de la clase y el cuerpo de la clase, que contiene sus propiedades y métodos. Las propiedades se declaran utilizando la palabra clave `val` (para propiedades de solo lectura) o `var` (para propiedades mutables).

```
class Persona {
    var nombre: String = ""
    var edad: Int = 0
}
```

En este ejemplo, hemos declarado una clase llamada `Persona` con dos propiedades: `nombre` y `edad`.

Creación de objetos

Para crear un objeto (instancia) de una clase, utiliza el nombre de la clase seguido de paréntesis. Puedes inicializar las propiedades del objeto utilizando el constructor de la clase.

```
val persona1 = Persona()
persona1.nombre = "Juan"
persona1.edad = 30
```

En este ejemplo, hemos creado una instancia de la clase `Persona` llamada `persona1` y luego hemos asignado valores a sus propiedades.

Constructor primario

En Kotlin, puedes definir un constructor primario directamente en la declaración de la clase. Esto te permite inicializar las propiedades de la clase directamente en la declaración del constructor.

```
class Persona(val nombre: String, val edad: Int)
```

Con esta declaración de clase, puedes crear objetos de la clase `Persona` y proporcionar valores para `nombre` y `edad` al crearlos.

```
val persona2 = Persona("María", 25)
```

Métodos

Dentro de la clase, puedes declarar métodos que definen el comportamiento de la clase. Los métodos se declaran utilizando la palabra clave `fun`.

```
class Persona(val nombre: String, val edad: Int) {
    fun saludar() {
        println("Hola, mi nombre es $nombre y tengo $edad años.")
    }
}
```

Luego, puedes llamar a los métodos en objetos de la clase.

```
val persona3 = Persona("Carlos", 35)
persona3.saludar()
```

Clases de datos (Data Classes)

Kotlin proporciona las clases de datos, que son una forma conveniente de crear clases que principalmente almacenan datos. Para declarar una clase de datos, utiliza la palabra clave `data` antes de la declaración de la clase.

```
data class Coche(val marca: String, val modelo: String)
```

Las clases de datos generan automáticamente funciones útiles como `equals()`, `hashCode()`, `toString()`, `copy()`, etc., basadas en las propiedades de la clase.

```
val coche1 = Coche("Toyota", "Camry")
val coche2 = Coche("Toyota", "Camry")
println(coche1 == coche2) // true (comparación de contenido)
```

PROPIEDADES Y MÉTODOS

En Kotlin, las propiedades y los métodos son elementos fundamentales de las clases que te permiten definir el estado y el comportamiento de los objetos. Aquí te explico cómo se definen propiedades y métodos en Kotlin:

Propiedades

1. **Propiedades de solo lectura (val):** Se definen utilizando la palabra clave `val` y no se pueden cambiar después de su inicialización. Se comportan como constantes.

```
class Persona {
    val nombre: String = "Juan"
}
```

4. **Propiedades mutables (var):** Se definen utilizando la palabra clave `var` y se pueden cambiar después de su inicialización.

```
class Coche {
    var marca: String = ""
    var modelo: String = ""
}
```

8. **Propiedades calculadas:** Puedes definir propiedades que no almacenan un valor directamente, sino que lo calculan en función de otras propiedades o datos.

```
class Circulo(val radio: Double) {
    val area: Double
        get() = 3.14159265359 * radio * radio
}
```

12. **Propiedades con campo de respaldo (backing field):** Puedes utilizar un campo de respaldo para una propiedad para almacenar su valor.

```
class Contador {
    private var _valor: Int = 0
    var valor: Int
        get() = _valor
        set(value) {
            _valor = value
        }
}
```

Métodos

Los métodos se definen utilizando la palabra clave `fun`, seguida del nombre del método y su cuerpo.

```
class Calculadora {
    fun sumar(a: Int, b: Int): Int {
        return a + b
    }
}
```

1. **Métodos con parámetros:** Puedes definir métodos que toman uno o más parámetros.

```
class Rectangulo {
    fun calcularArea(ancho: Double, alto: Double): Double {
        return ancho * alto
    }
}
```

6. **Métodos con valor de retorno:** Puedes especificar el tipo de valor de retorno de un método utilizando `: TipoDeRetorno`.

```
class Convertidor {
    fun metrosAPies(metros: Double): Double {
        return metros * 3.28084
    }
}
```

11. **Métodos con cuerpo de expresión:** Si un método tiene una única expresión como cuerpo, puedes simplificar su declaración.

```
class Calculadora {
    fun sumar(a: Int, b: Int): Int = a + b
}
```

14. **Métodos de clases de datos (data classes):** Las clases de datos en Kotlin generan automáticamente métodos como `equals()`, `hashCode()`, `toString()`, etc., basados en las propiedades de la clase.

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

HERENCIA

La herencia en Kotlin funciona de manera similar a otros lenguajes orientados a objetos, como Java, pero con algunas diferencias y características adicionales.

Clase base (superclase)

Para crear una clase base (superclase), utiliza la palabra clave `open` antes de la declaración de la clase. Esto indica que la clase es abierta para la herencia, lo que significa que otras clases pueden heredar de ella.

```
open class Animal {
    open fun hacerSonido() {
        println("El animal hace un sonido.")
    }
}
```

En este ejemplo, `Animal` es una clase base que contiene un método abierto (`hacerSonido`) que puede ser anulado por clases derivadas.

Clase derivada (subclase)

Para crear una clase derivada (subclase), utiliza la palabra clave `class` seguida del nombre de la subclase, dos puntos (`:`) y el

nombre de la superclase. Luego, puedes agregar propiedades y métodos adicionales a la subclase.

```
class Perro : Animal() {
    override fun hacerSonido() {
        println("El perro ladra.")
    }
}
```

En este ejemplo, `Perro` es una subclase de `Animal` y anula el método `hacerSonido` de la clase base.

Uso de la herencia

Puedes crear instancias de clases derivadas y utilizarlas de la misma manera que las instancias de la clase base.

```
val animal: Animal = Perro()
animal.hacerSonido() // Imprimirá "El perro ladra."
```

En este ejemplo, `animal` es una referencia de tipo `Animal` que apunta a una instancia de `Perro`. Kotlin permite la asignación de subclases a referencias de superclases.

Métodos y propiedades anulados

Para permitir que las clases derivadas anulen métodos o propiedades de la superclase, debes marcar los métodos o propiedades en la superclase con la palabra clave `open`, y luego en las clases derivadas, utiliza la palabra clave `override` para anularlos.

```
open class Animal {
    open fun hacerSonido() {
        println("El animal hace un sonido.")
    }
}

class Perro : Animal() {
    override fun hacerSonido() {
        println("El perro ladra.")
    }
}
```

Constructores de clases derivadas

Las clases derivadas pueden tener sus propios constructores. Para inicializar la superclase, debes llamar al constructor de la superclase utilizando la palabra clave `super`.

```
open class Animal(val nombre: String) {
    // ...
```

```
}

class Perro(nombre: String, val raza: String) : Animal(nombre) {
    // ...
}
```

En este ejemplo, el constructor de `Perro` llama al constructor de `Animal` usando `super(nombre)`.

Herencia múltiple

Kotlin no admite la herencia múltiple de clases (una clase derivada no puede heredar de múltiples clases base), pero puedes lograr comportamientos similares utilizando interfaces.

```
interface Nadador {
    fun nadar()
}

interface Volador {
    fun volar()
}

class Pato : Nadador, Volador {
    override fun nadar() {
        println("El pato nada en el agua.")
    }

    override fun volar() {
        println("El pato vuela en el cielo.")
    }
}
```

En este ejemplo, `Pato` implementa las interfaces `Nadador` y `Volador` para obtener funcionalidad de múltiples fuentes.

En resumen, la herencia en Kotlin se basa en clases abiertas (`open`) y permite crear jerarquías de clases donde las subclases pueden heredar y anular comportamientos de las superclases. También se pueden utilizar interfaces para lograr herencia múltiple.

CLASES ABSTRACTAS

En determinadas situaciones, como las clases de ejemplo Animal o Shape, no tiene sentido instanciar directamente objetos de las mismas. Podemos prevenir esto definiéndolas como clases abstractas. Para ello, antepondremos el modificador abstract en la declaración de la clase.

Al contrario de lo que ocurre con una clase normal, las clases abstractas son open por defecto (lo contrario no tendría sentido) Dentro de la clase abstracta podemos tener métodos con cuerpo y

métodos sin él. Estos últimos deberán declarase a su vez como abstract y las subclases estarán obligadas a implementarlos (o a ser declaradas como clases abstractas) Por ejemplo, vamos a redefinir nuestra clase Shape anterior como clase abstracta:

```
abstract class Shape {
   abstract fun area(): Double // no tiene cuerpo
}

class Rect(val width: Double, val height: Double): Shape() {
   override fun area() = width * height
}

class Circle(val radius: Double): Shape() {
   override fun area() = kotlin.math.PI * radius * radius
}

// no podemos crear objetos de Shape porque es abstracta
// la siguiente línea generaría un error
//val geom = Shape()

val r = Rect(2.0, 4.0)
println("area del rectángulo = ${r.area()}")

val c = Circle(2.0)
println("area del círculo = ${c.area()}")
```

Clases selladas (Sealed Classes)

Las clases selladas o sealed classes son clases abstractas de las que sólo existe un conjunto predefinido de subclases. En cierto modo, son similares a los tipos enumerados con la diferencia de que podemos tener múltiples instancias de dichos subtipos.

Por ejemplo, vamos a convertir la jerarquía Shape anterior en clases selladas. Nuestra clase Shape seguirá siendo abstracta pero, ahora, sólo podremos crear objetos de alguna de las clases hijas definidas en su interior. No podemos crear clases que hereden de Shape fuera del conjunto:

```
sealed class Shape {
   abstract fun area(): Double // no tiene cuerpo

   class Rect(val width: Double, val height: Double): Shape() {
      override fun area() = width * height
   }

   class Circle(val radius: Double): Shape() {
      override fun area() = kotlin.math.PI * radius * radius
   }
}
```

INTERFACES

Las interfaces en Kotlin son similares a las interfaces en otros lenguajes de programación orientados a objetos, pero con algunas características adicionales y ventajas. Las interfaces proporcionan un mecanismo para definir un conjunto de métodos que las clases concretas deben implementar. Aquí tienes una descripción de cómo son las interfaces en Kotlin:

Declaración de una interfaz

Para declarar una interfaz en Kotlin, utiliza la palabra clave `interface`, seguida del nombre de la interfaz y el cuerpo de la interfaz, que contiene la declaración de los métodos y, opcionalmente, propiedades abstractas.

```
interface Figura {
    fun area(): Double
    fun perimetro(): Double
}
```

En este ejemplo, `Figura` es una interfaz con dos métodos abstractos: `area()` y `perimetro()`.

Implementación de una interfaz

Las clases pueden implementar una o más interfaces proporcionando implementaciones concretas para los métodos definidos en esas interfaces. Para indicar que una clase implementa una interfaz, utiliza la palabra clave `:`, seguida del nombre de la interfaz.

```
class Circulo(val radio: Double) : Figura {
    override fun area(): Double {
        return 3.14159265359 * radio * radio
    }

    override fun perimetro(): Double {
        return 2 * 3.14159265359 * radio
    }
}
```

En este ejemplo, `Circulo` implementa la interfaz `Figura` proporcionando implementaciones concretas para los métodos `area()` y `perimetro()`.

Herencia de múltiples interfaces

Kotlin permite que una clase implemente múltiples interfaces, lo que facilita la composición de comportamientos de diferentes fuentes.

```
interface Nadador {
    fun nadar()
}

interface Volador {
    fun volar()
}

class Pato : Nadador, Volador {
    override fun nadar() {
        println("El pato nada en el agua.")
    }

    override fun volar() {
        println("El pato vuela en el cielo.")
    }
}
```

En este ejemplo, `Pato` implementa tanto la interfaz `Nadador` como la interfaz `Volador`.

Propiedades en interfaces

Además de los métodos, las interfaces pueden declarar propiedades abstractas y, opcionalmente, proporcionar implementaciones para accesorios (getters y setters).

```
interface Vehiculo {
    val velocidad: Double
    val capacidad: Int
}
```

En este ejemplo, la interfaz `Vehiculo` declara dos propiedades abstractas, `velocidad` y `capacidad`, que deben implementarse en las clases que implementan la interfaz.

Propiedades y métodos concretos en interfaces

A partir de Kotlin 1.0, las interfaces también pueden contener propiedades y métodos concretos (con implementaciones) utilizando la palabra clave `default`.

```
interface Animal {
    val nombre: String
    fun hacerSonido()

    fun comer() {
        println("$nombre está comiendo.")
    }
}
```

En este ejemplo, la interfaz `Animal` contiene una propiedad `nombre` y un método `hacerSonido()`. También proporciona una implementación por

defecto para el método `comer()`, que puede ser heredada por las clases que implementan la interfaz.

https://play.kotlinlang.org/#eyJ2ZXJzaW9uIjoiMi4xLjAiLCJwbGF0Zm9ybSI6ImphdmEiLCJhcmdzIjoiIiwibm9uZU1hcmtlcnMiOnRydWUsInRoZW1lIjoiaWRlYSIsImNvZGUiOiIvKipcbiAqIFlvdSBjYW4gZWRpdCwgcnVuLCBhbmQgc2hhcmUgdGhpcyBjb2RlLlxuICogcGxheS5rb3RsaW5sYW5nLm9yZ1xuICovXG5mdW4gbWFpbigpIHtcbiAgICBwcmludGxuKFwiSGVsbG8sIHdvcmxkIVwiKVxufSIsInZlcnNpb24iOiIifQ==