UD07 – INTRODUCCIÓN A KOTLIN

Contenido

1 Lenguaje de programación Kotlin

Kotlin es un lenguaje de programación moderno, conciso y seguro desarrollado por JetBrains. Se diseñó para ser completamente interoperable con Java, permitiendo a los desarrolladores aprovechar las bibliotecas y herramientas existentes. Su sintaxis clara y expresiva lo hace fácil de aprender para quienes tienen experiencia en lenguajes como Java, C#, Swift o JavaScript.

Kotlin se destaca por su enfoque en la seguridad de tipos, eliminando errores comunes como los NullPointerException, además de ofrecer potentes características como corutinas para programación asíncrona y una gran compatibilidad con el desarrollo de aplicaciones Android, backend y multiplataforma.

1.1 Características del lenguaje

Kotlin es un lenguaje de programación moderno y orientado a objetos que también admite paradigmas funcionales. Sus características facilitan la construcción de aplicaciones seguras, eficientes y mantenibles.

- Recolección de basura (Garbage Collection): Al igual que Java, Kotlin se ejecuta en la JVM y utiliza un recolector de basura para liberar automáticamente la memoria ocupada por objetos que ya no son accesibles, evitando problemas de administración manual de memoria.
- Manejo de excepciones: Kotlin proporciona un sistema estructurado para la captura y manejo de errores mediante try-catch-finally, asegurando una recuperación efectiva ante fallos en la ejecución.
- Seguridad de tipos: Kotlin minimiza errores comunes con su sistema de tipos seguro, evitando problemas como el acceso a variables no inicializadas, desbordamiento de arrays o conversiones de tipos inseguras.
- Sistema de tipos unificado: Aunque Kotlin diferencia entre tipos primitivos (Int, Double, etc.) y tipos de referencia, el compilador optimiza el uso de memoria, reduciendo la sobrecarga innecesaria. Además, Kotlin permite la creación de clases de datos, tipos sellados y tipos nulos seguros para mayor flexibilidad y seguridad.
- Soporte para programación funcional: Kotlin incorpora características como funciones de orden superior, expresiones lambda e inmutabilidad para mejorar la expresividad del código y facilitar el desarrollo de software robusto.
-

1.2 Programa *Hello World* en Kotlin

Para crear un proyecto de consola en Kotlin utilizando IntelliJ IDEA, sigue estos pasos:

1. Abrir IntelliJ IDEA y seleccionar "Nuevo Proyecto".
2. Elegir la plantilla "Aplicación Kotlin/JVM".
3. Asignar el nombre del proyecto como Hello.
4. Seleccionar el JDK (Java Development Kit) compatible, como JDK 17 o superior.
5. Pulsar "Crear".

Normalmente, los archivos de código fuente de Kotlin tienen la extensión .kt.

1.2.1 Programa "Hola Mundo" en Kotlin

Podemos escribir el código directamente en el archivo Main.kt y ejecutarlo. En Kotlin, println es la función que nos permite escribir en la consola.

```
fun main() {
    println("Hola Mundo")
}
```

1.2.2 Espacios de nombres en Kotlin

Para importar el espacio de nombres, en Kotlin utilizamos import para importar paquetes específicos. Por ejemplo, si queremos trabajar con entradas de usuario, podemos importar:

```
import java.util.Scanner
```

Kotlin organiza sus bibliotecas de manera similar a los paquetes en Java, permitiendo la reutilización y estructuración del código.

1.2.3 Compilación y ejecución

Cuando compilamos nuestro proyecto en IntelliJ IDEA o con Kotlin Compiler, se generará un archivo .jar ejecutable.

Si usamos la línea de comandos, podemos compilar y ejecutar con:

```
kotlinc Main.kt -include-runtime -d Hello.jar

java -jar Hello.jar
```

1.3 Entrada y salida de datos por consola

En Kotlin, la entrada y salida estándar se maneja con funciones de la biblioteca estándar, sin necesidad de una clase específica como Console en C#. A continuación, te muestro los métodos equivalentes en Kotlin:

Métodos de salida

Kotlin	Descripción
print(valor)	Escribe el valor sin salto de línea.
println(valor)	Escribe el valor seguido de un salto de línea.

Métodos de entrada

Kotlin	Descripción
System.`in.read()`	Lee un carácter de la entrada estándar.
readLine()	Lee una línea completa de la entrada estándar.

Ejemplo en Kotlin

```
fun main() {
    print("Introduce tu nombre: ")  // print() no agrega salto de línea
    val nombre = readLine()  // Captura la entrada del usuario
    println("Hola, $nombre")  // println() agrega un salto de línea
}
```

2 Estructura de un programa en Kotlin

En Kotlin, los principales conceptos organizativos incluyen paquetes, clases, funciones y módulos.

- Los programas de Kotlin constan de archivos fuente con la extensión .kt.
- Los paquetes organizan las clases y funciones de manera jerárquica, similar a los espacios de nombres en C#.
- Los tipos en Kotlin incluyen clases, interfaces, objetos y enumeraciones.
- Los módulos agrupan varios archivos fuente y pueden compilarse en JARs o bibliotecas compartidas.

2.1 Organización de un proyecto en Kotlin

2.1.1 Creación de un módulo de biblioteca en Kotlin

En IntelliJ IDEA, sigue estos pasos para crear una biblioteca reutilizable:

- Crear un nuevo proyecto en IntelliJ IDEA.
- Seleccionar "Aplicación Kotlin/JVM".
- Agregar un nuevo módulo de tipo "Biblioteca", nombrándolo Acumulador.
- Crear una clase Valor.kt dentro del paquete com.ejemplo.acumulador.

```
package com.ejemplo.acumulador


class Valor {
    var valor: Int = 0


    fun acumular(aumento: Int) {
        valor += aumento
    }
}
```

2.1.2 Creación de una aplicación de consola en Kotlin (Equivalente a AcumuladorApp.exe)

- Crear un nuevo proyecto llamado AcumuladorApp.
- Agregar una dependencia al JAR Acumulador.jar generado previamente.
- Escribir el siguiente código en Main.kt:

```
package com.ejemplo.acumuladorapp


import com.ejemplo.acumulador.Valor


fun main() {
    val miValor = Valor()
    miValor.acumular(10)
    println("El valor acumulado es: ${miValor.valor}")
}
```

2.1.3 Manejo de múltiples archivos en Kotlin

Ejemplo de dividir la clase Valor en dos archivos:

- Archivo Valor.kt

```
package com.ejemplo.acumulador


class Valor {
   var valor: Int = 0
}
```

- Archivo ValorMetodos.kt

```
package com.ejemplo.acumulador


fun Valor.acumular(aumento: Int) {
   this.valor += aumento
}
```

Al compilar, Valor tendrá acceso a acumular(), aunque esté definido en otro archivo.

3 Un vistazo a lenguaje Kotlin

3.1 Comentarios

En Kotlin, los comentarios siguen la misma sintaxis que en C# y Java:

```
// Comentario en una línea
/* Comentario en
 * varias líneas
 */
```

También se pueden usar **comentarios de documentación** con `/** ... */`, similares a `///` en C#.

```
/**
 * Esto es un comentario de documentación en Kotlin
 */
fun ejemplo() {}
```

3.2 Tipos y variables

Kotlin también distingue entre **tipos de valor (primitivos)** y **tipos de referencia (objetos)**. Sin embargo, a diferencia de C#, los tipos primitivos en Kotlin son **representados internamente como tipos primitivos de JVM** cuando es posible, lo que mejora la eficiencia.

3.2.1 Tipos de valor en Kotlin

Categoría	Lenguaje Kotlin
Enteros con signo	Byte, Short, Int, Long
Enteros sin signo	UByte, UShort, UInt, ULong (desde Kotlin 1.3)
Caracteres Unicode	Char
Punto flotante binario IEEE	Float, Double
Punto flotante decimal de alta precisión	No tiene equivalente directo
Booleanos	Boolean

Ejemplo de declaración de variables en Kotlin:

```
val numero: Int = 10   // Variable inmutable
var texto: String = "Hola"  // Variable mutable
```

3.2.2 Tipos de referencia en Kotlin

Categoría	Lenguaje Kotlin
Clase base de todos los tipos	Any
Cadenas Unicode	String
Clases definidas por el usuario	class MiClase {}
Interfaces	interface MiInterfaz {}
Arrays	Array<Int> o IntArray

Ejemplo de clases en Kotlin:

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

Ejemplo de una interfaz en Kotlin:

```
interface Animal {
    fun hacerSonido()
}

class Perro : Animal {
    override fun hacerSonido() {
        println("Guau!")
    }
}
```

3.3 Expresiones

Las expresiones en Kotlin también se construyen con operandos y operadores. Los operadores indican qué operaciones se aplican a los operandos, y la precedencia y asociatividad determinan el orden de evaluación.

3.3.1 Precedencia y Asociatividad

- Precedencia de operadores en Kotlin es similar a la de C#.
- Asociatividad en Kotlin sigue estas reglas:

Los operadores binarios (como +, -, *, /) son asociativos a la izquierda:

```
val resultado = 10 - 5 - 2  // Se evalúa como (10 - 5) - 2 = 3
```

Los operadores de asignación (=) y el operador ternario ?: son asociativos a la derecha:

```
val x: Int = 5
val y: Int = 10
val z: Int = x + y  // Se evalúa como (x + y)
```

3.3.2 Operadores aritméticos

Estos operadores funcionan igual que en Java:

Operador	Descripción	Ejemplo
+	Suma	`val suma = 5 + 3` → 8
-	Resta	`val resta = 5 - 3` → 2
*****	Multiplicación	`val producto = 5 * 3` → 15
/	División	`val division = 5 / 2` → 2 (*división entera*)
%	Módulo (resto)	`val resto = 5 % 2` → 1

En Kotlin, la división entre Int descarta la parte decimal. Para obtener un resultado decimal, uno de los operandos debe ser Double o Float:

```
val resultado = 5 / 2.0  // Resultado: 2.5
```

3.3.3 Operadores lógicos

Se usan para trabajar con valores booleanos (true o false).

Operador	Descripción	Ejemplo
!	Negación lógica (NOT)	val esFalso = !true → false
&&	AND lógico	val resultado = true && false → false

3.3.4 Operadores de comparación

Operador	Descripción	Ejemplo
==	Igual a	val esIgual = (5 == 5) → true
!=	No igual a	val esDistinto = (5 != 3) → true
<	Menor que	val menor = (3 < 5) → true
<=	Menor o igual que	val menorIgual = (3 <= 3) → true
>	Mayor que	val mayor = (5 > 3) → true
>=	Mayor o igual que	val mayorIgual = (5 >= 5) → true

```
val a = "Hola"

val b = "Hola"

println(a == b)  // true (compara valores)

println(a === b) // false (compara referencias)
```

3.4 Instrucciones

Kotlin tiene muchas similitudes con Java en cuanto a las instrucciones de control de flujo, pero con una sintaxis más concisa y expresiva. Veamos las principales diferencias y similitudes.

3.4.1 Bloques e Instrucciones de Declaración

- Bloques en Kotlin se definen con { }.
- Declaraciones de variables: En Kotlin, usamos val (inmutable) y var (mutable) en lugar de const y tipos explícitos.

Ejemplo en Kotlin:

```
fun declarations() {
    var a: Int
    var b = 2
    val c = 3  // `val` es como `const` en C#

    a = 1
    println(a + b + c)
}
```

Diferencias clave:

val es inmutable pero no una constante en tiempo de compilación.

Para valores constantes, se usa const val (solo dentro de object o companion object).

```
const val PI = 3.1415927
```

3.4.2 Instrucciones de Expresión

En Kotlin, cualquier expresión puede convertirse en una instrucción:

```
fun expressions() {
    var i: Int
    i = 123
    println(i)
    i++
    println(i)
}
```

Kotlin no requiere punto y coma ; al final de cada línea (a diferencia de Java).

Las acciones de un programa se expresan mediante *instrucciones*. C# admite varios tipos de instrucciones diferentes, varias de las cuales se definen en términos de instrucciones insertadas.

Un *bloque* permite que se escriban varias instrucciones en contextos donde se permite una única instrucción. Un bloque se compone de una lista de instrucciones escritas entre los delimitadores **{ }**.

3.4.3 Instrucciones de Selección (if, when)

- IF:

```
fun ifStatement(args: Array<String>) {
    if (args.isEmpty()) {
        println("No arguments")
    } else {
        println("One or more arguments")
    }
}
```

- switch en Kotlin → when

Kotlin reemplaza switch por when, que es más flexible:

```
fun switchStatement(args: Array<String>) {
    when (args.size) {
        0 -> println("No arguments")
        1 -> println("One argument")
        else -> println("${args.size} arguments")
    }
}
```

when no necesita break porque se detiene automáticamente después de una coincidencia.

3.4.4 Instrucciones de Iteración (`while`, `do-while`, `for`)

- while en Kotlin

```
fun whileStatement(args: Array<String>) {
    var i = 0
    while (i < args.size) {
        println(args[i])
        i++
    }
}
```

- `do-while` en Kotlin

```
fun doStatement() {
    var s: String?
    do {
        s = readlnOrNull()
        println(s)
    } while (!s.isNullOrEmpty())
}
```

- for en Kotlin

En Kotlin, el for estándar usa rangos en lugar de índices:

```
fun forStatement(args: Array<String>) {
    for (i in args.indices) {
        println(args[i])
    }
}
```

También se puede escribir con `forEach`:

```
args.forEach { println(it) }
```

3.4.5 Instrucciones de Salto (`break`, `continue`, `return`)

- break en Kotlin

```
fun breakStatement() {
    while (true) {
        val s = readlnOrNull()
        if (s.isNullOrEmpty()) break
        println(s)
    }
}
```

- continue en Kotlin

```
fun continueStatement(args: Array<String>) {
    for (arg in args) {
        if (arg.startsWith("/")) continue
        println(arg)
    }
}
```

- return en Kotlin

```
fun add(a: Int, b: Int): Int {
    return a + b
}

fun returnStatement() {
    println(add(1, 2))
    return
}
```

3.4.6 Manejo de Excepciones (`try-catch-finally`)

Lanzar una excepción en Kotlin

```
fun divide(x: Double, y: Double): Double {
    if (y == 0.0) throw ArithmeticException("Cannot divide by zero")
    return x / y
}
```

Capturar Excepciones (try-catch)

```
fun tryCatch(args: Array<String>) {
    try {
        if (args.size != 2) {
            throw IllegalArgumentException("Two numbers required")
        }
        val x = args[0].toDouble()
        val y = args[1].toDouble()
        println(divide(x, y))
    } catch (e: IllegalArgumentException) {
        println(e.message)
    } finally {
        println("Goodbye!")
    }
}
```

4 Clases y objetos

En Kotlin, las clases también son tipos fundamentales. Una clase combina estado (propiedades o campos) y comportamiento (métodos y otros miembros) en una unidad. Kotlin también soporta herencia y polimorfismo.

Las clases se definen con la palabra clave class, y su cuerpo se encierra entre llaves {}. Kotlin permite declarar propiedades directamente en el constructor.

Ejemplo de una clase sencilla:

```
class Point(val x: Int, val y: Int)
```

Para crear objetos se usa el operador `new` **implícitamente** (no se escribe). Ejemplo:

```
val p1 = Point(0, 0)
```

```
val p2 = Point(10, 20)
```

4.1 Miembros de una clase

Los miembros pueden ser:

- Constantes: con const val (solo en contexto companion object)
- Propiedades: con val (solo lectura) o var (lectura/escritura)
- Métodos: funciones declaradas dentro de la clase
- Constructores: inicializan objetos
- Bloques init: lógica de inicialización
- Destructores: Kotlin no tiene destructores como C#, pero se puede usar finalize() de Java si es absolutamente necesario

```
class Circunferencia() {
    var radio: Double = 0.0

    constructor(radio: Double) : this() {
        this.radio = radio
    }
}
```

Acceder a la propiedad `radio`:

```
val c = Circunferencia(5.0)
println(c.radio)     // lectura
c.radio = 10.0       // escritura
```

4.2 Propiedades

En Kotlin:

Se usan val (inmutable) y var (mutable)

Los getters y setters se definen automáticamente, pero se pueden personalizar:

```
var radio: Double = 0.0
    get() = field
    set(value) {
```

```
        field = if (value > 0) value else 0.0
    }
```

O bien propiedades autoimplementadas:

```
var nombre: String = "Kotlin"
```

4.3 Constructores

Si no defines un constructor, Kotlin crea uno por defecto. Puedes declarar un constructor primario y uno o más constructores secundarios.

Ejemplo:

```
class Persona(val nombre: String) {
    var edad: Int = 0

    constructor(nombre: String, edad: Int) : this(nombre) {
        this.edad = edad
    }
}
```

4.4 Destructores

Kotlin no usa destructores como C#. La recolección de basura es automática y no determinista.

Si realmente necesitas lógica de limpieza, puedes usar try/finally, use para recursos que implementan Closeable, o sobreescribir finalize() (no recomendado).

4.5 Modificadores de acceso

Modificador	Kotlin
public	Acceso desde cualquier parte
private	Acceso solo dentro de la clase

protected	Acceso desde la clase o subclases
internal	Acceso dentro del mismo módulo

Kotlin no tiene protected internal o private protected, pero internal cubre varios casos similares.

5 Tipos de datos integrados

Tipos de valor y tipos de referencia

En Kotlin, todos los tipos son objetos, pero los primitivos (Int, Float, etc.) se representan internamente como valores primitivos cuando es posible (como optimización).

Los tipos como Int, Double, Char, Boolean son inmutables y por valor.

Los objetos (clases, arrays, listas, etc.) se manejan por referencia.

Tipos numéricos

Tipo Kotlin	Rango aproximado	Tamaño	Comentarios
Byte	-128 a 127	8 bits	Igual que sbyte en C#
Short	-32,768 a 32,767	16 bits	
Int	-2^31 a 2^31-1	32 bits	
Long	-2^63 a 2^63-1	64 bits	
UByte	0 a 255	8 bits	(Kotlin/Native o con librerías específicas)
UInt	0 a 2^32-1	32 bits	
ULong	0 a 2^64-1	64 bits	

```
val decimalLiteral: Int = 42
val hexLiteral: Int = 0x2A
val binaryLiteral: Int = 0b00101010

val longLiteral: Long = 3_000_000_000L
```

val uintLiteral: UInt = 300U // Necesita la librería de unsigned

Tipos de coma flotante

Tipo Kotlin	Precisión	Tamaño	Sufijo	Uso
Float	6-7 decimales	32 bits	f	Números menos precisos
Double	15-17	64 bits	—	Default
BigDecimal	Muy alta	Depende	—	Usado para finanzas

val f: Float = 3_000.5f

val d: Double = 3.934_001

val scientific = 0.42e2

val bd = BigDecimal("3000.5")

Conversiones

En Kotlin las conversiones no son automáticas (como C# implícitas), deben hacerse explícitamente:

val x: Double = 1234.7

val a: Int = x.toInt() // 1234

Tipo Char

En Kotlin, Char también representa un carácter Unicode UTF-16:

val a: Char = 'j'

val b: Char = '\u006A'

val c: Char = 106.toChar()

println("ab$c") // jjj

Tipo Boolean

var passed: Boolean = true

```
println(passed)


passed = false
println(passed)
```

Operadores lógicos: !, &&, ||, ^

Strings en Kotlin

Los String se comportan como objetos inmutables (como en C#).

Puedes acceder a caracteres como si fuera un array:

```
val name = "Kotlin"
println(name[0]) // K
```

Cadenas de caracteres en Kotlin

En Kotlin, el tipo String también representa una secuencia inmutable de caracteres.

```
val saludo = "Hola mundo!" // Literal de String
```

Secuencias de escape

Kotlin usa las mismas secuencias de escape que C#:

- \n nueva línea
- \t tabulación
- \\ barra inversa
- \" comillas dobles
- etc.

```
val ejemplo = "Línea 1\nLínea 2\t\"Cita\""
```

Strings sin interpretar (raw strings)

Usando """ puedes escribir strings que no interpretan secuencias de escape:

```
val raw = """
    Línea 1
    Línea 2 con "comillas dobles"
```

""".trimIndent()

Interpolación de cadenas

```
val nombre = "Mark"
val yo = "Joe"
println("Hola, ${nombre.uppercase()}! Soy $yo")
```

Strings vacíos

```
val vacio = "" // o String()
```

Creación de strings a partir de caracteres

```
val chars = charArrayOf('H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o')
val holamundo = String(chars)
println(holamundo)
val hola = String(chars.sliceArray(0..3))
println(hola)
val asteriscos = "*".repeat(10)
println(asteriscos)
```

Comparación y concatenación

```
val a = "hello"
var b = "h"
b += "ello"

println(a == b) // true, compara contenido
println(a === b) // false, referencia

val mensaje = "good " + "morning"
```

Acceso a caracteres

```
val hello = "Hello world!"
for (i in hello.indices) {
   println(hello[i])
}
```

Métodos útiles de String

```
val texto = " Kotlin es Genial "
println(texto.contains("Genial"))
println(texto.startsWith(" "))
println(texto.endsWith(" "))
println(texto.indexOf("es"))
println(texto.lastIndexOf("n"))
println(texto.lowercase())
println(texto.uppercase())
println(texto.trim())
println(texto.replace("Genial", "poderoso"))
```

División y concatenación

```
val partes = "uno,dos,tres".split(",")
val unido = listOf("uno", "dos", "tres").joinToString(separator = ",")
```

Conversión entre números y cadenas

```
val numero = 123.456
println("%.2f".format(numero)) // "123.46"
println("%,.2f".format(1234567.89)) // "1,234,567.89" (depende del locale)
```

Conversión de String a número:

```
val str = "123"
val num = str.toIntOrNull() // evita excepción
println(num)
```

StringBuilder en Kotlin

```
val sb = StringBuilder()
sb.append("Hola")
sb.appendLine(" mundo")
sb.insert(5, " Kotlin")
println(sb.toString())

sb.delete(0, 5)
println(sb)
```

6 Declaración de variables

1. Declaración de variable en Kotlin

Kotlin permite declarar variables de manera similar a C# pero con algunas diferencias. En lugar de utilizar string para cadenas, se usa String (con mayúscula inicial).

Declaración explícita

```
val greeting: String = "Hello"
```

Aquí val indica una variable de solo lectura (inmutable), y var se usa para declarar una variable mutable.

Declaración implícita

Kotlin puede inferir el tipo de una variable usando val o var sin necesidad de declararlo explícitamente:

```
val greeting = "Hello"  // El tipo String es inferido automáticamente.
```

Palabras clave reservadas

Al igual que en C#, no se pueden usar las palabras clave de Kotlin como nombres de variables. Para esto, se pueden usar comillas invertidas ` `. Ejemplo:

```
val `if` = "This is a variable"
```

2. Operadores en Kotlin

Operadores aritméticos

Kotlin utiliza los mismos operadores aritméticos básicos que C#:

Suma (+):

```
println("2 + 1 = ${2 + 1}")
```

Resta (-):

```
println("2 - 1 = ${2 - 1}")
```

Multiplicación (*):

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

División (/) y resto (%):

```
println("7 / 3 = ${7 / 3}")
println("7 % 3 = ${7 % 3}")
```

Operadores de incremento y decremento

Al igual que en C#, Kotlin soporta operadores de incremento (++) y decremento (--):

```
var i = 3
```

```
println(i++)  // Imprime 3, luego i es incrementado a 4
println(++i)  // Imprime 5
```

Operadores lógicos

Los operadores lógicos en Kotlin son similares a los de C#:

AND (&&):

```
val a = true
val b = false
println(a && b)  // false
```

OR (||):

```
println(a || b)  // true
```

NOT (!):

```
println(!a)  // false
```

XOR (^):

```
println(a xor b)  // true
```

Operadores de igualdad y comparación

Los operadores de comparación en Kotlin son:

Igualdad (==):

```
println(1 == 2)  // false
```

Mayor que (>):

```
println(3 > 2)  // true
```

Menor que (<):

```
println(3 < 2)  // false
```

3. Desbordamiento aritmético

Kotlin no lanza excepciones de desbordamiento aritmético de forma predeterminada, pero se puede controlar mediante el uso de la función Int.MAX_VALUE y realizando un control explícito si es necesario. Sin embargo, en Kotlin no existe el checked como en C#. Para manejar el desbordamiento, se deben usar excepciones o validaciones personalizadas.

```
val a = Int.MAX_VALUE
val b = a + 1  // No lanza excepción, pero el resultado es incorrecto
println(b)     // -2147483648
```

4. Operadores a nivel de bit

En Kotlin también tenemos operadores a nivel de bits:

Complemento bit a bit (~):

```
val x = 5
println(~x)  // -6
```

Desplazamiento a la izquierda (shl):

```
val y = 4  // 0100
println(y shl 1)  // 8 -> 1000
```

Desplazamiento a la derecha (shr):

```
val z = 16  // 10000
println(z shr 1)  // 8 -> 1000
```

Operadores AND, OR, XOR a nivel de bit:

```
val a = 0b00001100
```

```
val b = 0b11110110
println(a and b)  // AND bit a bit
println(a or b)   // OR bit a bit
println(a xor b)  // XOR bit a bit
```

Resumen de Operadores en Kotlin

Operadores Aritméticos:

+, -, *, /, %, ++, --

Operadores Lógicos:

&&, ||, !, xor

Operadores Relacionales:

==, !=, >, <, >=, <=

Operadores a Nivel de Bits:

and, or, xor, shl, shr, inv

7 Tipos de enumeración. Enum.

En Kotlin, las enumeraciones (enums) funcionan de manera similar a C#, pero con su propia sintaxis y características. A continuación te voy a mostrar cómo convertir los ejemplos de C# a Kotlin.

7.1 Declaración de una Enumeración Básica

En Kotlin, las enumeraciones son una forma de declarar un conjunto de constantes que representan una serie de opciones. La sintaxis de una enum en Kotlin es similar a C#:

```
enum class Operation {
    Addition,
```

```
    Subtraction,
    Multiplication,
    Division
}
```

Al igual que en C#, las constantes dentro de una enumeración en Kotlin son asignadas a valores enteros automáticamente, comenzando desde 0. El valor de la constante Addition será 0, Subtraction será 1, y así sucesivamente.

7.2 Acceder a un Valor de Enum por su Índice

En Kotlin puedes obtener un valor de la enumeración usando el índice como en C#:

```
val op: Operation = Operation.values()[1]  // Recupera 'Subtraction'
println("${op.ordinal} - ${op.name}")      // Imprime "1 - Subtraction"
```

En este ejemplo, el índice 1 accede a la constante Subtraction, y el método ordinal muestra el valor numérico asociado, mientras que name muestra el nombre como una cadena.

7.3 Asignar Valores a los Elementos de una Enumeración

Puedes asignar valores personalizados a los elementos de la enumeración de la misma manera que en C#:

```
enum class Operation(val value: Int) {
    Addition(10),
    Subtraction(20),
    Multiplication(30),
    Division(40)
}
```

Aquí, cada constante dentro de la enum tiene un valor entero asociado (value). Puedes acceder al valor de cada constante con el atributo value:

```
val op = Operation.Addition
println(op.value)  // Imprime "10"
```

7.4 Uso de Enumeraciones con when en Kotlin

El operador when de Kotlin se utiliza para trabajar con enums de manera más fluida:

```
fun performOperation(op: Operation) {
    when (op) {
        Operation.Addition -> println("Performing addition")
        Operation.Subtraction -> println("Performing subtraction")
        Operation.Multiplication -> println("Performing multiplication")
        Operation.Division -> println("Performing division")
    }
}
```

7.5 Enumeraciones como Bit Flags en Kotlin

Al igual que en C#, en Kotlin puedes utilizar enumeraciones con bit flags usando el atributo @Flags y valores que sean potencias de 2. En Kotlin, no se tiene el atributo Flags explícito como en C#, pero el concepto es el mismo. Cada valor de la enumeración se representa como una potencia de 2 y puedes combinar estos valores usando operaciones bit a bit.

Ejemplo: Uso de bit flags en Kotlin

Supongamos que tienes una enumeración para los días de la semana, y quieres permitir que un valor pueda combinar varios días. En este caso, asignamos potencias de 2 a cada día:

```
enum class Days(val value: Int) {
    Monday(1),      // 2^0
    Tuesday(2),     // 2^1
    Wednesday(4),   // 2^2
    Thursday(8),    // 2^3
    Friday(16),     // 2^4
    Saturday(32),   // 2^5
    Sunday(64)      // 2^6
}
```

7.6 Operaciones con bit flags en Kotlin

Para combinar los días, puedes usar el operador or (|):

```
val meetingDays = Days.Monday.value or Days.Wednesday.value or Days.Friday.value
println(meetingDays)  // Imprime 21 (1 | 4 | 16)
```

Para verificar si un valor específico está presente, puedes usar el operador and (&):

```
val isMeetingOnTuesday = meetingDays and Days.Tuesday.value ==
Days.Tuesday.value
println(isMeetingOnTuesday)  // Imprime "false"
```

Y para eliminar un valor, puedes usar el operador and con el complemento ~ del valor:

```
val updatedMeetingDays = meetingDays and (Days.Friday.value.inv())
println(updatedMeetingDays)  // Imprime 5 (1 | 4)
```

7.7 Ejemplo de Uso en un Formulario en Kotlin

Supongamos que en una aplicación de escritorio en Kotlin, se usa un CheckBox para aplicar diferentes estilos de fuente (negrita, cursiva, subrayada). En este caso, usarías una enumeración similar a FontStyle en C#.

Primero, definimos la enumeración FontStyle:

```
enum class FontStyle(val value: Int) {
   Regular(0),     // 2^0
   Bold(1),        // 2^1
   Italic(2),      // 2^2
   Underlined(4)   // 2^3
}
```

En un formulario, podrías combinar estos estilos utilizando los operadores de bit:

```
var currentStyle = FontStyle.Regular.value
```

```
// Cambiar a negrita
currentStyle = currentStyle or FontStyle.Bold.value
println(currentStyle)  // Imprime 1 (Bold)

currentStyle = currentStyle or FontStyle.Italic.value
println(currentStyle)  // Imprime 3 (Bold | Italic)

currentStyle = currentStyle and FontStyle.Bold.value.inv()
println(currentStyle)  // Imprime 2 (Italic)
```

En este caso, la variable currentStyle puede almacenar una combinación de estilos, y puedes aplicar estilos múltiples como lo harías con un bit flag.

8 Tipos de valor que aceptan valores null

En Kotlin, los tipos de valor que aceptan valores null se manejan a través de tipos anulables (nullable types). En lugar de utilizar el concepto de T? como en C#, Kotlin tiene un sistema de tipos que permite la asignación de valores null a tipos de valor, y usa un sistema basado en la sintaxis T?, donde T es el tipo de valor y ? indica que puede ser null.

1.1 Tipos Anulables en Kotlin

Un tipo anulable puede contener un valor normal o el valor null. Para declarar un tipo anulable en Kotlin, se usa el operador ? después del tipo:

```
var b: Boolean? = null
```

En este caso, b puede contener true, false, o null.

1.2 Ejemplo de Uso

Kotlin permite que un tipo de valor sea nulo al usar el operador ?:

```
var pi: Double? = 3.14

var letter: Char? = 'a'

var m2: Int = 10

var m: Int? = m2  // Asignamos m2, un tipo no anulable, a m, un tipo anulable.

var flag: Boolean? = null
```

Además, los arreglos pueden contener tipos anulables:

```
val arr: Array<Int?> = arrayOfNulls(10)  // Arreglo de enteros anulables con 10 elementos
```

1.3 Acceso y Verificación de Tipos Anulables

En Kotlin, puedes verificar si un valor de tipo anulable es null usando un patrón similar a C# 7.0 con el operador is:

```
val a: Int? = 42

if (a is Int) {
   println("a is $a")
} else {
   println("a does not have a value")
}
```

9 Structs en Kotlin

Aunque Kotlin no tiene una implementación directa de structs como en C#, tiene un concepto similar con data classes y tipos de valor (como los records en Java). En Kotlin, las data classes se utilizan para modelar datos que contienen varias propiedades, y el lenguaje maneja internamente los detalles de la creación de un objeto de tipo valor.

9.1 Data Classes en Kotlin

Las data class en Kotlin son similares a los structs de C# en el sentido de que contienen solo datos (no comportamientos complejos como las clases normales). Estas clases permiten la creación de instancias de objetos sin necesidad de escribir mucho código repetitivo, ya que Kotlin automáticamente proporciona métodos como toString(), equals(), hashCode(), y copy().

Ejemplo de una data class:

data class Point(val x: Int, val y: Int)

En este caso, Point es una data class que almacena las coordenadas x e y.

9.2 Comportamiento de Asignación en Kotlin

Al igual que en C#, los tipos de valor (como las data class en Kotlin) se copian en lugar de pasarse por referencia:

val a = Point(10, 10)

val b = a // `b` recibe una copia de `a`

a.copy(x = 20) // Copia de `a` con el valor `x` cambiado

println(b.x) // Imprime "10", ya que `b` sigue siendo una copia de `a` antes del cambio.

En este ejemplo:

Si Point fuera una clase (en lugar de una data class), ambas variables a y b apuntarían a la misma referencia en memoria, por lo que b.x cambiaría a 20.

Si Point es una data class, b mantiene una copia independiente de a, y al modificar a, b no se ve afectada.

9.3 Ventajas de Usar structs (Data Classes) en Kotlin

Al igual que los structs en C#, las data classes son más eficientes en memoria y CPU cuando se utilizan para representar datos pequeños e inmutables, ya que los objetos de tipo data class se almacenan de manera similar a los tipos de valor y no requieren un manejo dinámico de memoria en el heap.

9.4 Limitaciones de las Data Classes en Kotlin

No herencia: Las data class no pueden heredar de otras clases (de la misma manera que los structs en C# no pueden heredar de otros structs o clases). Heredan implícitamente de Any, el tipo base de Kotlin.

No se puede declarar un constructor sin parámetros: Al igual que los structs, las data class en Kotlin requieren al menos un parámetro en su constructor.

Copia: Cuando se asigna un valor de una data class a una nueva variable, se crea una copia independiente, lo que significa que los cambios en una no afectan a la otra.

9.5 Estructura de Datos Pequeñas vs. Complejas

Kotlin recomienda usar data classes cuando se necesite un tipo de valor que no cambie después de su creación, como un punto en un sistema de coordenadas o un par clave-valor. Para estructuras más complejas, o si se planea modificar el objeto después de la creación, es más adecuado utilizar clases normales (class).

data class Point(val x: Int, val y: Int)

10 Arrays

10.1. Arrays o Matrices en Kotlin

En Kotlin, los arrays funcionan de manera similar a otros lenguajes, pero con algunas diferencias en cuanto a la declaración y manipulación. Aquí te muestro cómo se trabajan:

Declaración de Arrays:

- Unidimensionales:

val a = IntArray(3) // Array de enteros con 3 elementos, inicializados en 0

a[0] = 1

a[1] = 2

a[2] = 3

También se puede inicializar directamente con valores:

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

- Multidimensionales:

```
val a2 = Array(3) { IntArray(3) } // Matriz 3x3
a2[0][0] = 1
```

- Escalonados: Similar a los arrays de tipo "array de arrays":

```
val a = Array(3) { IntArray(0) } // Array de 3 elementos, cada uno es un array vacío
a[0] = intArrayOf(1, 2, 3)
a[1] = intArrayOf(4, 5)
a[2] = intArrayOf(6, 7, 8, 9)
```

Matrices Transpuestas: Aquí te muestro cómo transponer una matriz 2D en Kotlin:

```
val matrix = arrayOf(
   intArrayOf(3, 5, 7),
   intArrayOf(2, 4, 6),
   intArrayOf(1, 2, 3)
)

val transposed = Array(matrix[0].size) { IntArray(matrix.size) }
for (i in matrix.indices) {
   for (j in matrix[i].indices) {
      transposed[j][i] = matrix[i][j]
   }
}
```

10.2. Listas en Kotlin

En Kotlin, las listas son muy flexibles y se pueden manipular con varias operaciones. Aquí se explica cómo usar las listas y sus características:

Crear listas:

- Lista mutable (se puede modificar):

```
val list = mutableListOf(1, 2, 3)
list.add(4)
list.removeAt(1)
println(list) // [1, 3, 4]
```

- Lista inmutable (no se puede modificar):

```
val list = listOf(1, 2, 3)
```

Buscar y ordenar listas:

Para buscar un elemento en la lista:

```
val found = list.find { it > 2 } // Encuentra el primer número mayor que 2
println(found) // 3
```

Ordenar una lista:

```
val sortedList = list.sorted()
println(sortedList) // [1, 2, 3]
```

10.3. Funciones en Kotlin

Las funciones en Kotlin son similares a otros lenguajes de programación, pero con una sintaxis más concisa. Aquí te explico cómo usar funciones y parámetros:

Definir una función:

```
fun suma(a: Int, b: Int): Int {
   return a + b
}
println(suma(5, 3)) // 8
```

Parámetros de referencia: En Kotlin no usamos ref o out como en C#. Sin embargo, se pueden usar var para pasar por referencia, ya que Kotlin utiliza un modelo de paso por valor y paso por referencia, pero sin la misma sintaxis de ref y out de C#.

Parámetros opcionales:

```
fun saludo(nombre: String, apellido: String = "Desconocido") {
   println("Hola, $nombre $apellido")
}
saludo("Juan")  // "Hola, Juan Desconocido"
saludo("Juan", "Pérez")  // "Hola, Juan Pérez"
```

Uso de vararg (equivalente a params en C#): Para pasar un número variable de parámetros:

```
fun imprimirNumeros(vararg numeros: Int) {
   for (numero in numeros) {
      println(numero)
   }
}
imprimirNumeros(1, 2, 3, 4, 5)
```

10.4. Sobrecarga de Funciones

En Kotlin, la sobrecarga de funciones se maneja de manera similar a otros lenguajes, y se basa en que las funciones tengan una firma única, diferenciada por el tipo o número de parámetros:

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

fun saludar(nombre: String, edad: Int) {
   println("Hola, $nombre. Tienes $edad años.")
}
```

```
saludar("Juan")
saludar("Juan", 30)
```

Ejemplo Completo

Aquí tienes un ejemplo que combina todos los conceptos anteriores:

```
// Función para transponer una matriz
fun transponer(matriz: Array<IntArray>): Array<IntArray> {
    val filas = matriz.size
    val columnas = matriz[0].size
    val transpuesta = Array(columnas) { IntArray(filas) }

    for (i in matriz.indices) {
        for (j in matriz[i].indices) {
            transpuesta[j][i] = matriz[i][j]
        }
    }
    return transpuesta
}

// Crear un array bidimensional
val matriz = arrayOf(
    intArrayOf(1, 2, 3),
    intArrayOf(4, 5, 6),
    intArrayOf(7, 8, 9)
)

// Imprimir la matriz transpuesta
val transpuesta = transponer(matriz)
```

```
for (fila in transpuesta) {
    println(fila.joinToString(", "))
}

// Listas y funciones con parámetros
val lista = mutableListOf(10, 20, 30)
lista.add(40)
println(lista) // [10, 20, 30, 40]

fun saludo(nombre: String, apellido: String = "Desconocido") {
    println("Hola, $nombre $apellido")
}

saludo("Juan")
saludo("Pedro", "Gómez")

// Usando vararg
fun imprimirNumeros(vararg numeros: Int) {
    for (numero in numeros) {
        println(numero)
    }
}

imprimirNumeros(1, 2, 3, 4, 5)
```

11 Funciones

En general, en Kotlin hablaremos de funciones para referirnos a lo que en C# serían métodos o procedimientos. Una función es un miembro que implementa un cálculo o una acción que puede realizar un objeto o una clase. Las funciones estáticas se acceden a través del companion object de la clase, mientras que las funciones de instancia se acceden a través de instancias de la clase.

Las funciones pueden tener una lista de parámetros, que representan valores o referencias a variables que se pasan a la función, y un tipo de valor devuelto, que especifica el tipo del valor calculado y devuelto por la función. El tipo de valor devuelto de una función es Unit si no se devuelve un valor.

Firma de una Función

La firma de una función en Kotlin debe ser única en la clase en la que se declara la función. La firma de una función se compone del nombre de la función y los tipos de sus parámetros. La firma de una función no incluye el tipo de valor devuelto.

Los parámetros se usan para pasar valores o referencias a variables a las funciones. Los parámetros de una función obtienen sus valores reales de los argumentos que se especifican cuando se invoca la función.

En Kotlin, no se permite la sobrecarga de funciones solo por la diferencia en los modificadores como ref, in o out.

Parámetros en Kotlin

Los parámetros en Kotlin se manejan de manera similar a C#, pero con algunas diferencias clave:

- Parámetros por valor: Todos los parámetros se pasan por valor por defecto en Kotlin.
- Parámetros opcionales: En Kotlin, podemos asignar valores predeterminados a los parámetros para que sean opcionales.
- Parámetros vararg: En Kotlin, los parámetros variables se pasan utilizando vararg.
- Parámetros in y out: Kotlin tiene soporte para covariancia y contravariancia en las colecciones mediante in y out, pero no es igual a la forma en que C# utiliza ref o out.

Ejemplos de Funciones en Kotlin

Función básica:

```
// Función de instancia
fun suma(a: Int, b: Int): Int {
   return a + b
}

println(suma(2, 3))  // Salida: 5
```

Función estática (equivalente a un método estático en C#):

```
class MiClase {
   companion object {
      fun saludo() {
         println("¡Hola, mundo!")
      }
   }
}

MiClase.saludo() // Se llama a través de la clase
```

Parámetros opcionales:

En Kotlin, los parámetros de una función pueden tener valores predeterminados, lo que permite que ciertos parámetros sean opcionales:

```
fun saludo(nombre: String, apellido: String = "Desconocido") {
   println("Hola, $nombre $apellido")
}

saludo("Juan")       // Salida: "Hola, Juan Desconocido"
saludo("Juan", "Pérez") // Salida: "Hola, Juan Pérez"
```

Parámetros vararg:

En Kotlin, puedes pasar un número variable de parámetros utilizando vararg:

```
fun imprimirNumeros(vararg numeros: Int) {
   for (numero in numeros) {
      println(numero)
   }
}

imprimirNumeros(1, 2, 3, 4, 5) // Salida: 1 2 3 4 5
```

Parámetros in y out en Kotlin

Kotlin tiene soporte para covariancia y contravariancia a través de los modificadores in y out en los tipos genéricos. Sin embargo, estos no se utilizan de la misma manera que ref o out en C#.

- out: En Kotlin, out se usa en tipos genéricos para indicar que un tipo solo se utilizará para devolver valores (covariancia).

```
class Caja<out T>(val valor: T)

val caja = Caja(5) // Caja<Int>

val valor: Any = caja.valor // Se puede asignar a un tipo más general
```

- in: Se usa para denotar que el tipo solo se usará para recibir valores (contravariancia).

```
class Caja<in T>(val valor: T)

val caja: Caja<Number> = Caja(5) // Caja<Int> se puede usar donde se espera Caja<Number>
```

Sobrecarga de Funciones

Kotlin permite la sobrecarga de funciones, pero no se puede hacer sobrecarga solo por los modificadores ref, in, o out como en C#. Sin embargo, puedes sobrecargar funciones basándote en el número o el tipo de los parámetros.

```
fun mostrarMensaje(mensaje: String) {
    println(mensaje)
}

fun mostrarMensaje(mensaje: String, nombre: String) {
    println("$mensaje, $nombre")
}

mostrarMensaje("Hola")        // Salida: "Hola"
mostrarMensaje("Hola", "Juan") // Salida: "Hola, Juan"
```

Ejemplo Completo en Kotlin

```
class Ejemplo {
    // Función con parámetros por valor
    fun sumar(a: Int, b: Int): Int {
        return a + b
    }

    // Función con parámetro vararg
    fun imprimirNumeros(vararg numeros: Int) {
        for (numero in numeros) {
            println(numero)
        }
    }

    // Función con parámetros opcionales
    fun saludar(nombre: String = "Desconocido", apellido: String = "Desconocido") {
        println("Hola, $nombre $apellido")
    }
```

```
    // Función estática (companion object)
    companion object {
        fun saludoEstático() {
            println("¡Hola desde un método estático!")
        }
    }
}

fun main() {
    val ejemplo = Ejemplo()

    // Llamada al método de instancia
    println(ejemplo.sumar(5, 3))  // 8

    // Llamada al método vararg
    ejemplo.imprimirNumeros(1, 2, 3, 4, 5)

    // Llamada a un método con parámetros opcionales
    ejemplo.saludar(nombre = "Juan")

    // Llamada al método estático
    Ejemplo.saludoEstático()
}
```

12 Tuplas

En Kotlin, no existe un tipo de tupla como tal, pero puedes usar pares o estructuras similares a tuplas utilizando data classes, Pair, y Triple. A continuación, te muestro cómo puedes manejar tuplas y sus equivalentes en Kotlin, además de cómo trabajar con fechas y horas.

Tuplas en Kotlin

Kotlin no tiene un tipo específico para tuplas, pero puedes usar el tipo Pair (para tuplas de dos elementos) o Triple (para tuplas de tres elementos). También puedes usar una data class si necesitas más flexibilidad en la estructura de los datos.

Pair y Triple

- Pair: Es una clase genérica que te permite almacenar dos elementos de cualquier tipo.

```
// Definición de un Pair
val tupla: Pair<String, Int> = Pair("Juan", 25)

// Acceso a los elementos
println(tupla.first)  // "Juan"
println(tupla.second) // 25
```

- Triple: Es una clase genérica que te permite almacenar tres elementos de cualquier tipo.

```
// Definición de un Triple
val tuplaTriple: Triple<String, Int, Boolean> = Triple("Juan", 25, true)

// Acceso a los elementos
println(tuplaTriple.first)  // "Juan"
println(tuplaTriple.second) // 25
println(tuplaTriple.third)  // true
```

Usando Data Class para más de tres elementos

Si necesitas una estructura con más de tres elementos, puedes crear una data class en Kotlin:

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

val persona = Persona("Juan", 25, "Madrid")

println(persona.nombre)  // "Juan"

println(persona.edad)    // 25

println(persona.ciudad)  // "Madrid"
```

Devolviendo múltiples valores desde una función

Las tuplas o estructuras similares como Pair o Triple son muy útiles para devolver múltiples valores desde una función. Aquí tienes un ejemplo en Kotlin:

```
// Función que devuelve un Pair

fun obtenerDatos(): Pair<String, Int> {

   return Pair("Juan", 25)

}


val datos = obtenerDatos()

println("Nombre: ${datos.first}, Edad: ${datos.second}")
```

En este caso, la función obtenerDatos() devuelve un Pair con un String y un Int.

13 Tipos de fecha y hora

Tipos de Fecha y Hora en Kotlin

Kotlin no tiene una estructura de fecha y hora propia, pero se puede trabajar con las clases de java.time (disponibles desde Java 8), que Kotlin puede usar sin problemas. A continuación, te muestro cómo trabajar con fechas y horas utilizando la biblioteca java.time.

Usando LocalDateTime

El tipo LocalDateTime se utiliza para representar una fecha y hora, sin tener en cuenta la zona horaria.

```
import java.time.LocalDateTime

val fechaHoraActual = LocalDateTime.now()

println("Fecha y hora actual: $fechaHoraActual")
```

Formateando Fechas y Horas

Puedes formatear fechas y horas usando la clase DateTimeFormatter. Aquí tienes ejemplos de cómo hacerlo en Kotlin.

Formato estándar:

```
import java.time.LocalDate

import java.time.format.DateTimeFormatter

val fecha = LocalDate.now()

println("Fecha estándar: $fecha")

val formato = DateTimeFormatter.ofPattern("dd/MM/yyyy")

val fechaFormateada = fecha.format(formato)

println("Fecha personalizada: $fechaFormateada")
```

Formato de hora:

```
import java.time.LocalTime

import java.time.format.DateTimeFormatter

val hora = LocalTime.now()

println("Hora estándar: $hora")

val formatoHora = DateTimeFormatter.ofPattern("HH:mm:ss")

val horaFormateada = hora.format(formatoHora)

println("Hora personalizada: $horaFormateada")
```

Comparación de Fechas y Horas

También puedes comparar fechas y horas en Kotlin de manera sencilla. Ejemplo de comparación entre dos instantes de tiempo:

```
import java.time.LocalDate

val fecha1 = LocalDate.of(2025, 4, 25)

val fecha2 = LocalDate.now()

println("¿Es fecha1 después de fecha2? ${fecha1.isAfter(fecha2)}")

println("¿Es fecha1 antes de fecha2? ${fecha1.isBefore(fecha2)}")
```