LENGUAJE DE PROGRAMACIÓN

Tour de Kotlin: Fundamentos del Lenguaje

Guía completa de los conceptos esenciales de Kotlin
Basado en la documentación oficial de JetBrains

7 Módulos | Ejemplos Prácticos | Nivel Principiante

```
fun main () { println ( "¡Hola, Kotlin!" ) }
```

¿Qué es Kotlin?

Kotlin es un lenguaje de programación **moderno**, **conciso**, **seguro** y **pragmático** desarrollado por JetBrains.

Tipado Estático

Inferencia de tipos automática y compilación en tiempo de desarrollo

Interoperabilidad

100% compatible con Java y su ecosistema de librerías

Null Safety

Detección de errores en tiempo de compilación para evitar NullPointerException

Conciso y Expresivo

Reduce el código boilerplate y aumenta la legibilidad

```
// Kotlin combina simplicidad y poder
val mensaje = "Aprende Kotlin paso a paso"
```

Hello World – Primer Programa

Main.kt

```kotlin
fun main() {
  println("Hello, world!")
  // Output:
  // Hello, world!
}
```

fun

Palabra clave para declarar una función

main()

Punto de entrada del programa. Todo código Kotlin comienza aquí

println()

Imprime el argumento en la salida estándar con salto de línea

{ }

Cuerpo de la función encerrado entre llaves

Estructura Básica de un Programa

Funciones

Conjunto de instrucciones que realizan una tarea específica

Declaraciones

Definen variables, valores y estructuras del programa

Expresiones

Producen valores y pueden ser evaluadas

Punto de Entrada

La función `main()` donde inicia la ejecución

Program.kt

```kotlin
fun main() {
    // Declaración de variable
    val saludo = "Kotlin"

    // Expresión y función
    println("¡Hola, $saludo!")
}
```

Puntos Clave

- Las llaves `{ }` delimitan el cuerpo de funciones
- Cada programa inicia desde `main()`

Tipos Básicos – Introducción

¿Qué son los Tipos?

Cada variable y estructura de datos en Kotlin tiene un **tipo**.

Validación en Compilación

El compilador sabe qué operaciones son válidas según el tipo

Inferencia de Tipos

Kotlin detecta automáticamente el tipo por el valor asignado

Tipos Principales

`Int`	`String`	`Boolean`
`Double`	`Float`	`Long`

TypesExample.kt

```kotlin
// Inferencia de tipos automática
val clientes = 10
    // Kotlin infiere que es Int

// Tipos explícitos
val precio: Double = 19.99
val mensaje: String = "Hola"
val activo: Boolean = true

// Operaciones según tipo
val total = clientes * 2
    // Operaciones aritméticas permitidas
```

Beneficio Clave

Los tipos previenen errores detectando operaciones inválidas **en tiempo de compilación**, no en ejecución.

Variables var y val

var Mutable

- Valor **modificable**
- Se puede reasignar
- Útil para contadores, acumuladores

val Inmutable

- Valor de **solo lectura**
- No se puede reasignar
- Preferido para constantes, configuraciones

Variables.kt

```
// Ejemplo con var (mutable)
var contador = 1
contador = 2 // ✓ Válido
contador = contador + 1

// Ejemplo con val (inmutable)
val PI = 3.14159
PI = 3.14 // ✗ Error

// Recomendación
val nombre = "Kotlin" // ✓ Preferido
```

Buena Práctica

Usa `val` por defecto. Solo cambia a `var` cuando necesites modificar el valor.

Tipos Numéricos

Int

- Enteros de 32 bits
- Rango: -2^{31} a $2^{31}-1$

Ejemplo: 10, -5, 1000

Long

- Enteros de 64 bits
- Rango: -2^{63} a $2^{63}-1$

Ejemplo: 100L, 999999L

Float

- Decimales de 32 bits
- 6-7 dígitos de precisión

Ejemplo: 3.14f, 2.5f

Double

- Decimales de 64 bits
- 15-16 dígitos de precisión

Ejemplo: 3.14159, 2.5

Operaciones Aritméticas

`+` | `-` | `*` | `/` | `%` | `++` | `--`

NumericTypes.kt

```kotlin
// Enteros
val edad: Int = 25
val poblacion: Long = 8_000_000L

// Decimales
val precio: Double = 19.99
val temp: Float = 36.5f

// Operaciones
val suma = edad + 5
val total = precio * 2
val resto = edad % 3

// Conversión de tipos
val x: Double = edad.toDouble()
```

Nota Importante

Por defecto, los literales enteros son `Int` y los decimales son `Double`

Tipo String y Plantillas

Cadenas de Texto

Representan secuencias de caracteres entre comillas dobles

Plantillas de Cadena

- `$` para variables simples
- `${...}` para expresiones complejas

Caracteres de Escape

`\n`	`\t`	`\\`	`\"`
Nueva línea	Tabulación	Barra	Comillas

StringTemplates.kt

```kotlin
// Declaración básica
val nombre = "Kotlin"
val version = 2.0

// Plantilla simple con $
val saludo = "Hola, $nombre!"
    // Resultado: "Hola, Kotlin!"

// Plantilla con expresión ${}
val mensaje = "Versión: ${version + 1}"
    // Resultado: "Versión: 3.0"

// Concatenación tradicional
val completo = nombre + " v" + version

// Caracteres de escape
val lineas = "Línea 1\nLínea 2"
```

Recomendación

Prefiere las plantillas de cadena sobre la concatenación para código más legible y eficiente.

Tipo Boolean

true

Representa un valor verdadero o afirmativo

false

Representa un valor falso o negativo

Operadores Lógicos

&&	||	!
AND	OR	NOT
Ambos verdaderos	Al menos uno verdadero	Invierte el valor

Uso Común

- Condicionales y control de flujo
- Validaciones y verificaciones

BooleanExample.kt

```kotlin
// Declaración de booleanos
val estaActivo = true
val tienePermiso = false

// Operadores lógicos
val puedeAcceder = estaActivo &&
tienePermiso
    // false (ambos deben ser true)
val puedeVer = estaActivo || tienePermiso
    // true (al menos uno es true)
val estaInactivo = !estaActivo
    // false (niega el valor)

// Expresiones de comparación
val edad = 20
val esMayor = edad >= 18
    // true

// Uso en condicionales
if (esMayor) {
    println("Adulto")
}
```

Colecciones – Introducción

Agrupación de datos para procesamiento

List

Ordenada

Colección ordenada que permite elementos duplicados

Set

Única

Colección de elementos únicos sin orden específico

Map

Clave-Valor

Conjunto de pares clave-valor con claves únicas

Mutabilidad

Solo Lectura

- `List`, `Set`, `Map`
- No se pueden modificar

Mutable

- `MutableList`, `MutableSet`
- `MutableMap`
- Se pueden agregar, eliminar, modificar

Las colecciones de solo lectura son más seguras y eficientes

Listas (List y MutableList)

List (Solo Lectura)

- Se crea con `listOf()`
- Ordenada, permite duplicados
- Acceso por índice: `[0]`

MutableList (Modificable)

- Se crea con `mutableListOf()`
- Se puede agregar/eliminar elementos
- Acceso por índice: `[0]`

Σ Métodos Comunes

`add()` `remove()` `contains()`

Lists.kt

```kotlin
// Lista de solo lectura
val frutas = listOf("Manzana", "Banana", "Naranja")
println(frutas[0]) // Manzana

// Lista mutable
val numeros = mutableListOf(1, 2, 3)

// Agregar elemento
numeros.add(4)
    // [1, 2, 3, 4]

// Eliminar elemento
numeros.remove(2)
    // [1, 3, 4]

// Verificar si existe
val existe = numeros.contains(3)
    // true

// Tamaño de la lista
val tamano = numeros.size // 3
```

Los índices comienzan en `0`. El último elemento está en `list.size - 1`

Sets (Set y MutableSet)

Set (Solo Lectura)

- Se crea con `setOf()`
- Elementos únicos, sin duplicados
- No garantiza orden específico

MutableSet (Modificable)

- Se crea con `mutableSetOf()`
- Se puede agregar/eliminar elementos
- Mantiene unicidad automáticamente

Característica Principal

Los **duplicados se ignoran automáticamente**

Sets.kt

```kotlin
// Set de solo lectura
val colores = setOf("Rojo", "Verde", "Azul")

// Set con duplicados (se ignoran)
val numeros = setOf(1, 2, 2, 3, 3, 3)
    // Resultado: {1, 2, 3}

// MutableSet
val letras = mutableSetOf("A", "B")

// Agregar elemento (no duplica)
letras.add("C")
letras.add("A") // No se agrega (ya existe)
    // {A, B, C}

// Verificar pertenencia
val existe = letras.contains("B")
    // true

// Tamaño del set
val tamano = letras.size // 3
```

Ideal para verificar **existencia** y **eliminar duplicados** de colecciones

Maps (Map y MutableMap)

Map (Solo Lectura)

- Se crea con `mapOf()`
- Pares clave → valor
- Claves únicas, no duplicados

MutableMap (Modificable)

- Se crea con `mutableMapOf()`
- Se puede agregar/eliminar pares
- Actualizar valores existentes

Σ Métodos Principales

`put()` `get()` `remove()`

Maps.kt

```kotlin
// Map de solo lectura
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 28
)

// Acceso por clave
val edadJuan = edades["Juan"] // 25

// MutableMap
val capital = mutableMapOf(
    "España" to "Madrid"
)

// Agregar par
capital.put("Francia", "París")

// Obtener valor
val capitalEsp = capital.get("España") // Madrid

// Eliminar par
capital.remove("Francia")
```

Si la clave no existe, `get()` retorna `null`

Flujo de Control – Introducción

Decisiones Condicionales

Kotlin evalúa expresiones como **verdadero** o **falso**

Expresiones Condicionales

- `if` - Condición simple
- `when` - Múltiples ramas

Bucles e Iteraciones

- `for` - Iterar sobre rangos
- `while` - Repetir mientras sea true

ControlFlow.kt

```kotlin
// Expresión condicional
val edad = 18

// Condicional if
if (edad ⩾ 18) {
    println("Mayor de edad")
} else {
    println("Menor de edad")
}

// Condicional when
when (edad) {
    18 → println("Justo 18")
    in 0..17 → println("Menor")
    else → println("Adulto")
}

// Bucle for
for (i in 1..5) {
    println(i)
}
```

El control de flujo permite crear programas **dinámicos** y **adaptativos**

Condicionales: if

Sintaxis Básica

- Condición entre `( )`
- Acción entre `{ }`
- Llaves opcionales si es una línea

if como Expresión

- No existe operador ternario `? :`
- `if` retorna un valor directamente
- Se puede asignar a una variable

Ventaja Principal

if como expresión hace el código más **conciso** y **expresivo**

IfCondition.kt

```kotlin
// if como sentencia
val edad = 20
if (edad ⩾ 18) {
    println("Mayor de edad")
} else {
    println("Menor de edad")
}

// if como expresión
val mensaje = if (edad ⩾ 18) {
    "Adulto"
} else {
    "Menor"
}

// if-else if-else
val calificacion = 85
val nota = if (calificacion ⩾ 90) {
    "A"
} else if (calificacion ⩾ 80) {
    "B"
} else {
    "C"
}
```

Cuando se usa como expresión, `if` debe tener rama `else`

Condicionales: when

Sintaxis con when

- Valor a evaluar entre ( )
- Ramas con → en cada caso
- Cuerpo entre { }

Ventajas de when

- Más **legible** que múltiples if-else
- Fácil de **extender** nuevas ramas
- Menos **errores** en el código

Modos de Uso

Sentencia - ejecuta acciones

Expresión - retorna valor

WhenCondition.kt

```kotlin
// when como sentencia
val dia = 3
when (dia) {
    1 → println("Lunes")
    2 → println("Martes")
    3 → println("Miércoles")
    else → println("Otro día")
}

// when como expresión
val calificacion = 95
val nota = when (calificacion) {
    in 90..100 → "A"
    in 80..89 → "B"
    in 70..79 → "C"
    else → "D"
}

// when sin sujeto
val x = 10
val y = 5
when {
    x > y → println("x mayor")
    x < y → println("y mayor")
    else → println("iguales")
}
```

Se recomienda usar `when` sobre múltiples if-else para mejor legibilidad

Bucles: for

Iteración con for

- Itera sobre **rangos** y **colecciones**
- Sintaxis: `for (item in colección)`
- Operador `in` para iterar

Operadores de Rango

- `..` - Rango ascendente (1..5)
- `downTo` - Rango descendente
- `step` - Incremento personalizado

Casos de Uso

- → Recorrer **listas** y **arrays**
- → Iterar sobre **rangos numéricos**
- → Procesar **caracteres de string**

ForLoop.kt

```kotlin
// Iterar sobre rango ascendente
for (i in 1..5) {
    println(i)
}

// Rango descendente
for (i in 5 downTo 1) {
    println(i)
}

// Rango con step
for (i in 1..10 step 2) {
    println(i) // 1, 3, 5, 7, 9
}

// Iterar sobre lista
val frutas = listOf("Manzana", "Banana", "Naranja")
for (fruta in frutas) {
    println(fruta)
}

// Iterar con índice
for ((index, fruta) in frutas.withIndex()) {
    println("$index: $fruta")
}
```

El operador `..` crea un rango inclusivo (incluye ambos extremos)

Bucles: while y do-while

Bucle while

- Se ejecuta mientras condición sea **true**
- Puede no ejecutarse si condición inicial es **false**

Bucle do-while

- Ejecuta el bloque **al menos una vez**
- Verifica condición **después** de ejecutar

Diferencia Clave

`while`	→	`do-while`
Verifica antes		Verifica después

WhileLoop.kt

```kotlin
// Ejemplo con while
var contador = 1
while (contador ≤ 5) {
    println(contador)
    contador++
}
    // Output: 1, 2, 3, 4, 5

// Ejemplo con do-while
var numero = 0
do {
    println(numero)
    numero++
} while (numero < 3)
    // Output: 0, 1, 2 (se ejecuta 3 veces)

// Diferencia práctica
var x = 10
// while no se ejecuta (condición false)
while (x < 5) {
    println("No se imprime")
}
// do-while se ejecuta una vez
do {
    println("Se imprime una vez")
} while (x < 5)
```

Usa `do-while` cuando necesites ejecutar al menos una vez, independientemente de la condición

Funciones – Introducción

Σ Declaración de Funciones

- Usa palabra clave `fun`
- Parámetros entre `( )` con tipos
- Tipo de retorno después de `:`
- Cuerpo entre `{ }`

Componentes Clave

Parámetros - Datos de entrada

Tipo de retorno - Resultado esperado

`return` - Devuelve el valor

Tт Convención de Nombres

Minúscula inicial, **camelCase** sin guiones bajos

Ejemplo: `calcularSuma()`, `procesarDatos()`

Functions.kt

```kotlin
// Función básica
fun saludar() {
    println("¡Hola, mundo!")
}

// Función con parámetros y retorno
fun sumar(x: Int, y: Int): Int {
    return x + y
}

// Llamar a la función
val resultado = sumar(5, 3)
println(resultado) // 8

// Función con múltiples parámetros
fun presentar(nombre: String, edad: Int) {
    println("Nombre: $nombre, Edad: $edad")
}

// Función con retorno de String
fun obtenerSaludo(nombre: String): String {
    return "¡Hola, $nombre!"
}
```

Las funciones permiten **reutilizar código** y organizar mejor el programa

Parámetros y Valores por Defecto

Valores por Defecto

- Asigna valor con **=**
- Permite **omitir parámetros** al llamar
- Útil para **configuraciones opcionales**

Reglas de Uso

- Parámetros sin valor por defecto son **obligatorios**
- Puedes omitir múltiples parámetros con valor por defecto
- Después del primer omitido, usa **argumentos nombrados**

Beneficios

Flexibilidad en llamadas a funciones sin sobrecargar código

DefaultParams.kt

```kotlin
// Función con valores por defecto
fun saludar(
    nombre: String,
    saludo: String = "Hola",
    mayusculas: Boolean = false
) {
    val mensaje = "$saludo, $nombre!"
    if (mayusculas) {
        println(mensaje.uppercase())
    } else {
        println(mensaje)
    }
}

// Llamadas a la función
saludar("María")
    // "Hola, María!"
saludar("Pedro", "Bienvenido")
    // "Bienvenido, Pedro!"
saludar("Ana", mayusculas = true)
    // "HOLA, ANA!"
saludar("Luis", "Saludos", true)
    // "SALUDOS, LUIS!"
```

Los valores por defecto eliminan la necesidad de múltiples `sobrecargas` de funciones

Argumentos Nombrados y Unit

Argumentos Nombrados

- Especifica parámetro con `nombre =`
- **Mejora legibilidad** del código
- Permite **cambiar el orden** de argumentos

Tipo Unit

- `return` es opcional
- Tipo de retorno **se omite** automáticamente

Uso Recomendado

Usa argumentos nombrados cuando haya múltiples parámetros con valores por defecto

NamedArguments.kt

```kotlin
// Función con múltiples parámetros
fun configurar(
    servidor: String = "localhost",
    puerto: Int = 8080,
    ssl: Boolean = false,
    timeout: Int = 30
) {
    println("Servidor: $servidor:$puerto")
    println("SSL: $ssl, Timeout: ${timeout}s")
}

// Llamadas con argumentos nombrados
configurar() // Todos por defecto
configurar(puerto = 3000)
configurar(ssl = true)
configurar(
    servidor = "api.ejemplo.com",
    ssl = true,
    timeout = 60
) // Orden diferente

// Función con Unit (sin retorno explícito)
fun mostrarMensaje(texto: String) {
    println(texto)
    // return Unit implícito
}
```

Los argumentos nombrados hacen que el código sea más **expresivo** y fácil de mantener

Funciones de Expresión Única

Qué Son

- Funciones con **una sola expresión**
- Sintaxis más **concisa y elegante**
- `return` implícito

Simplificación

- Elimina `llaves { }` y return
- Usa = en lugar de { return }
- Código más **limpio y legible**

Cuándo Usar

Ideal para funciones simples de cálculo o transformación

SingleExpression.kt

```kotlin
// Forma completa
fun sumar(a: Int, b: Int): Int {
    return a + b
}

// Función de expresión única
fun sumar(a: Int, b: Int): Int = a + b

// Otros ejemplos
fun duplicar(x: Int): Int = x * 2
fun esMayor(edad: Int): Boolean = edad >= 18
fun saludar(nombre: String): String = "¡Hola, $nombre!"
fun calcularArea(radio: Double): Double = Math.PI * radio
* radio

// Con inferencia de tipo
fun duplicar(x: Int) = x * 2
    // Comparación visual
// Complejo: 5 líneas
fun cuadrado(n: Int): Int {
    return n * n
}
// Simplificado: 1 línea
fun cuadrado(n: Int): Int = n * n
```

El compilador puede **inferir el tipo de retorno** si se omite, pero es recomendable declararlo

Clases – Introducción

¿Qué es una Clase?

- Plantilla para crear **objetos**
- Define **características y comportamiento**
- Se declara con `class`

Conceptos Clave

Propiedades – Datos de la clase

Métodos – Comportamiento

Instancia – Objeto creado

Ventajas de OOP

Organización, reutilización y encapsulamiento del código

Classes.kt

```kotlin
// Declaración de clase simple
class Persona

// Clase con propiedades
class Usuario(
    val nombre: String,
    var edad: Int
)

// Crear instancia
val usuario = Usuario("María", 25)

// Acceder a propiedades
println(usuario.nombre) // María
usuario.edad = 26

// Clase con método
class Rectangulo(
    val ancho: Double,
    val alto: Double
) {
    fun area(): Double = ancho * alto
}

val rect = Rectangulo(5.0, 3.0)
println(rect.area()) // 15.0
```

Kotlin genera automáticamente un **constructor primario** con las propiedades declaradas

Propiedades y Constructores

Propiedades

- Se declaran en `cabecera ( )`
- `val` para inmutables
- `var` para mutables

Constructor Automático

Kotlin genera constructor con parámetros de la cabecera

Valores por defecto con `= valor`

Acceso a Propiedades

Usa el operador `.` punto

Ejemplo: `objeto.propiedad`

Properties.kt

```kotlin
// Clase con propiedades
class Persona(
    val nombre: String, // Inmutable
    var edad: Int // Mutable
)

// Crear instancia
val persona = Persona("Ana", 30)

// Acceder a propiedades
println(persona.nombre) // Ana
persona.edad = 31 // ✓ Permitido
// persona.nombre = "Luis" // ✗ Error

// Valores por defecto
class Libro(
    val titulo: String,
    val autor: String = "Desconocido",
    var paginas: Int = 0
)

// Instancias con valores por defecto
val libro1 = Libro("Kotlin Avanzado")
val libro2 = Libro("Java Básico", "Juan García")

// Clase con cuerpo
class Cuenta(var saldo: Double) {
    fun depositar(monto: Double) {
        saldo += monto
    }
}
```

Recomendación: usa `val` por defecto, solo `var` si necesitas modificar

Seguridad con null (Null Safety)

Tipos Nullable

- ? indica que acepta null
- Prevención en **tiempo de compilación**
- Evita NullPointerException

Operadores de Seguridad

- ?. Acceso seguro
- !! Forzar no null (riesgoso)
- ?: Elvis operator

★ Mejor Práctica

Usa ?. en lugar de !! siempre que sea posible

NullSafety.kt

```kotlin
// Tipos nullable y no nullable
var nombre: String = "Ana"
// nombre = null // ✗ Error de compilación
var apellido: String? = null // ✓ Nullable

// Operador de acceso seguro ?.
val longitud = apellido?.length
println(longitud) // null

// Operador Elvis ?:
val longitud2 = apellido?.length ?: 0
println(longitud2) // 0

// Operador !! (forzar)
val texto = apellido!!.length
// NullPointerException si apellido es null

// Verificación con if
if (apellido != null) {
    println("Longitud: ${apellido.length}")
}

// Ejemplo práctico
fun obtenerUsuario(id: Int): String? {
    return if (id == 1) "Admin" else null
}
val usuario = obtenerUsuario(2)
println(usuario?.uppercase() ?: "Invitado")
```

Null Safety es una de las características más importantes de Kotlin para código **seguro y robusto**