# Introducción a Kotlin

*By Alejandro Gomez Florez (@aldajo92)*
*Ingeniero de Control | Ms Matemáticas aplicadas Universidad Nacional de Colombia | EAFIT*

> Guia en desarrollo.

Hola, espero que te encuentres de maravilla. Desde el 2016 he tenido la oportunidad de trabajar en grandes proyectos móviles tales como:

- Zelle [2016-2017]: Aplicación para transacción de dinero entre bancos en EE.UU como Chase, Wells Fargo, Bank of America, entre otros.
- Rappi [2017-2019]: App usada ampliamente en Latinoamérica; contribuí en la infraestructura, así como en diferentes secciones como restaurantes, Rappi Favor y pago de facturas.
- Disney World App [2019-2021]: App para navegar en los parques de Disney como Walt Disney World, Disneyland Resort, Animal Kingdom, EPCOT y Hollywood Studios, entre otros.
- All.Health [2021-2022]: Startup enfocada en diagnóstico y prevención de enfermedades usando teléfonos móviles, empleando un smart band que detecta signos vitales.

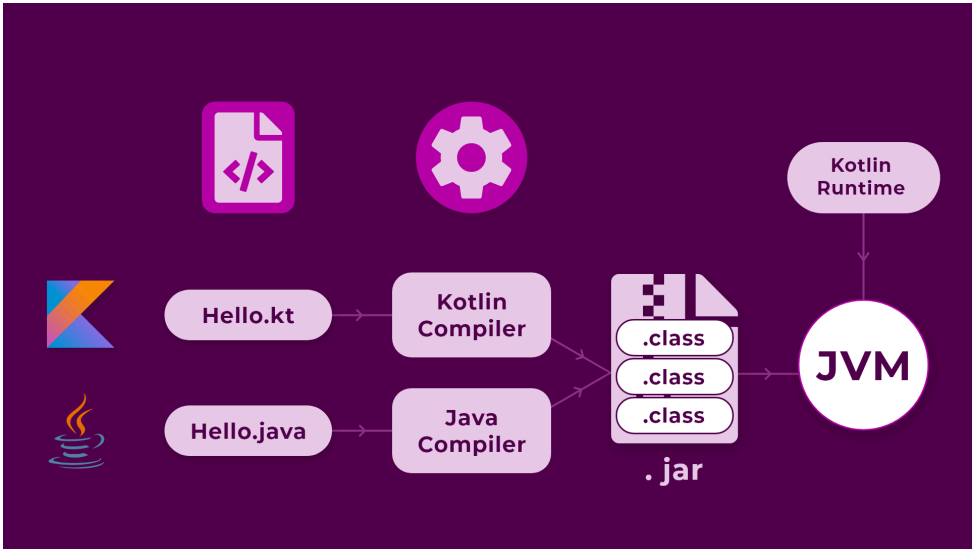


Figura 1: *Fuente:* https://developersbreach.com/kotlin-java-and-interoperability/

Es para mí un gusto poder desarrollar esta guía, inspirada en aspectos que tomé del aprendizaje autónomo, alejada un poco del formalismo académico. Me he puesto en su lugar ya que en algun momento tambien fui un principiante; el desarrollo de apps móviles

no es algo que se imparta formalmente en universidades o academias; por lo tanto, les brindaré consejos útiles para que puedan abordar el aprendizaje de manera practica. Vamos a empezar con un par de aspectos a considerar:

- Kotlin es un lenguaje de programación que se compila usando el compilador `kotlinc` a bytecode (.class) para ser ejecutado en la JVM (Java Virtual Machine), como se muestra en la Figura 1.
- Kotlin nace de la necesidad de poder trabajar con la JVM y con el Android Runtime, usando un lenguaje open source que no requiera licencia.
- Hasta 2017 Java era el lenguaje oficial para el desarrollo de apps móviles; luego Google hizo oficial Kotlin ese año.
- Kotlin y Java son compatibles, pero requiere experiencia y paciencia para trabajar con ambos lenguajes. No es necesario saber Java para empezar a programar en Kotlin.
- Actualmente Kotlin se usa para trabajar en diferentes áreas, no solo en mobile:
  - Se usa para crear sistemas del lado del backend con frameworks como Spring.
  - Gracias a la versión Kotlin Multiplatform se puede también usar para generar apps para otros dispositivos más allá de Android, como iOS y web.
- Cuenta con una amplia documentación, además de poder correrse en un playground de forma gratuita y sin necesidad de registrarse. https://play.kotlinlang.org/

## Contenido complementario

Aprender cualquier lenguaje requiere de práctica y paciencia. Así como los gimnasios los podemos usar para entrenar nuestros músculos, existen formas equivalentes para entrenar nuestra capacidad de programar y de resolver problemas a nuestro ritmo. Es por ello que recomiendo un par de sitios web donde pueden practicar y repasar conceptos:

- Leetcode.com: Sitio web donde pueden practicar problemas de programación y algoritmos empleados en entrevistas de trabajo de empresas como Google, Amazon, Microsoft, entre otras.
- Kotlin cheat sheet: Repositorio con un cheat sheet de Kotlin para repasar conceptos del lenguaje.
- Kotlin Devhints: Resumen con pequeñas cartas (cards) con la sintaxis y ejemplos prácticos.
- Programiz: Sitio web con tutoriales y ejercicios para aprender Kotlin.

## Primeros pasos

Vamos a revisar el siguiente código (usar este enlace https://pl.kotl.in/sCnSFbDSI para verlo en el navegador):

```
// Conversor de millas a km
fun main(){
```

```
    // Definimos la entrada
    val millas = 36

    // Definimos la operacion: 1 milla  ->  1.609 km. milla*1.609 [km]
    val kilometros = millas * 1.609

    // Definimos la salida
    println(kilometros)
}
```

A partir del codigo anterior se recomienda tener en cuenta los siguientes conceptos para empezar:

- El comentario se realiza con el símbolo `//` seguido del texto del comentario.

```
// Este es un comentario
```

- El bloque `main` es el punto de entrada de una aplicación Kotlin. Se define como una función con la palabra reservada `fun` seguido del nombre de la funcion, en este caso `main`. El nombre de la funcion es libre, pero `main` es un nombre reservado.

```
fun main(){
    // Bloque de codigo
}
```

- Las variables se definen con la palabra reservada `val` para variables inmutables y `var` para variables mutables.

```
val nombre = "Juan"
var edad = 20
```

- Las operaciones se realizan con los operadores aritméticos tradicionales: `+`, `-`, `*`, `/`, `%`.

```
val resultado = 10 + 5
```

- La función `println` se usa para imprimir un mensaje en la consola.

```
println("Hola, mundo!")
```

# Tipos de datos

Aca hay una lista de los tipos de datos más comunes en Kotlin:

```
    val booleanVar: Boolean = true
    val byteVar: Byte = 127
    val shortVar: Short = 32767
    val intVar: Int = 2147483647
    val longVar: Long = 9223372036854775807L
    val floatVar: Float = 3.14f
```

```
val doubleVar: Double = 3.14159265358979323846
val charVar: Char = 'A'
val stringVar: String = "Hello, world!"
```

En el ejemplo anterior se puede observar que las variables se definen con un tipo de dato explícito usando el formato `val nombre: Tipo = valor`. Esta es una forma opcional de declarar el tipo de dato, pero gracias a la inferencia de tipos de Kotlin no es necesario especificarlo explícitamente. Por lo tanto, lo siguiente es equivalente al ejemplo anterior:

```
val booleanVar = true
val byteVar = 127
val shortVar = 32767
val intVar = 2147483647
val longVar = 9223372036854775807L
val floatVar = 3.14f
val doubleVar = 3.14159265358979323846
val charVar = 'A'
val stringVar = "Hello, world!"
```

Las variables en Kotlin también pueden declararse sin un valor inicial, pero en ese caso, el tipo debe especificarse explícitamente:

```
var z: Double // Válido, z no tiene valor inicial
// println(z) // Inválido, z no está inicializada y no tiene valor aún
z = 3.14    // Válido, z es inicializada con un valor
```

Tipos de conversiones:

```
val str: String = "123"
val num: Int = str.toInt() // Convert String to Int

val dbl: Double = 123.45
val int: Int = dbl.toInt() // Convert Double to Int

val lng: Long = 9876543210
val flt: Float = lng.toFloat() // Convert Long to Float

val bol: Boolean = true
val strBol: String = bol.toString() // Convert Boolean to String

val char: Char = 'A'
val intChar: Int = char.code // Convert Char to Int usando la propiedad .code

val byte: Byte = 127
val short: Short = byte.toShort() // Convert Byte to Short
```

# Funciones

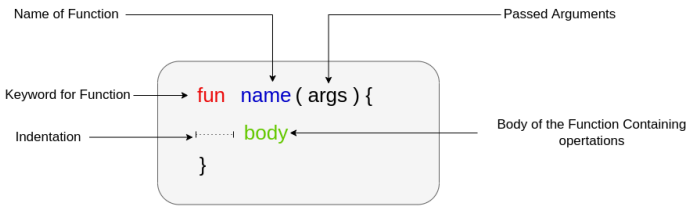


Figura 2: Anatomía de una función:
https://www.geeksforgeeks.org/kotlin/kotlin-functions/

En Kotlin, las funciones se definen usando la palabra reservada `fun` seguido del nombre de la función. Las funciones son bloques de código reutilizables que realizan una tarea específica.

Función básica sin parámetros y sin retorno:

```
fun saludar() {
    println("¡Hola, mundo!")
}

fun main() {
    saludar() // Imprime: ¡Hola, mundo!
}
```

Funciones con parámetros:

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

fun main() {
    saludar("Juan")  // Imprime: ¡Hola, Juan!
    saludar("María") // Imprime: ¡Hola, María!
}
```

Funciones con múltiples parámetros:

```
fun sumar(a: Int, b: Int) {
    val resultado = a + b
    println("La suma es: $resultado")
}

fun main() {
    sumar(5, 3)   // Imprime: La suma es: 8
    sumar(10, 20) // Imprime: La suma es: 30
}
```

Funciones con valor de retorno:

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

fun main() {
    val resultado = sumar(5, 3)
    println(resultado) // Imprime: 8
}
```

Funciones con parámetros por defecto:

```
fun saludar(nombre: String = "Usuario", edad: Int = 0) {
    println("Hola $nombre, tienes $edad años")
}

fun main() {
    saludar()                  // Imprime: Hola Usuario, tienes 0 años
    saludar("Ana")             // Imprime: Hola Ana, tienes 0 años
    saludar("Carlos", 25)      // Imprime: Hola Carlos, tienes 25 años
    saludar(edad = 30, nombre = "Luis") // Argumentos con nombre
}
```

Funciones de expresión única:

```
fun doble(x: Int): Int = x * 2

fun esParOrImpar(numero: Int): String = if (numero % 2 == 0) "Par" else "Impar"

fun main() {
    println(doble(5))       // Imprime: 10
    println(esParOrImpar(4)) // Imprime: Par
    println(esParOrImpar(7)) // Imprime: Impar
}
```

Ejemplo práctico con calculadora simple:

```
fun calcular(operacion: String, a: Int, b: Int): Int {
    return when (operacion) {
        "sumar" -> a + b
        "restar" -> a - b
        "multiplicar" -> a * b
        "dividir" -> a / b
        else -> 0
    }
}

fun main() {
    println(calcular("sumar", 10, 5))      // Imprime: 15
    println(calcular("restar", 10, 5))     // Imprime: 5
    println(calcular("multiplicar", 10, 5)) // Imprime: 50
    println(calcular("dividir", 10, 5))    // Imprime: 2
}
```

## Ejemplos

```
fun main(){
    hola()
    sumatory(10,11)
    val saludo = hi()
    print(saludo)
}

// No reciben parametros y no retornan algo
fun hola(){
    println("Hola")
}

// reciben parametros y no retornan algo
fun sumatory(a: Int, b: Int) {
    val result = a+b
    println(result)
}

// No reciben parametros y retornan algo
fun hi(): String {
    return "Hi"
}

// reciben parametros y retornan algo
fun sum(a: Int, b: Int): Int {
    val result = a+b
    return result
}
```

# Arrays & Collections

Arrays en Kotlin se definen usando las funciones `arrayOf`, `intArrayOf`, `doubleArrayOf`, `charArrayOf`, `booleanArrayOf`, `shortArrayOf` y `byteArrayOf`.

```
    val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
    val names: Array<String> = arrayOf("John", "Jane", "Jim", "Jill")
    val mixed: Array<Any> = arrayOf(1, "two", 3.0)
```

Los arrays, en general, son un tipo de estructura de dato que permite almacenar un conjunto de elementos del mismo tipo, con un tamaño fijo. En Kotlin los arrays son **mutables en su contenido** (se puede hacer `array[i] = nuevoValor`), pero su **tamaño es inmutable**: no se pueden agregar o eliminar posiciones sin crear un nuevo array. Se pueden declarar de diferentes maneras:

```
// 1. Usando arrayOf con valores iniciales
val array1 = arrayOf(1, 2, 3, 4, 5)

// 2. Usando intArrayOf para arrays de tipo primitivo Int
val array2 = intArrayOf(1, 2, 3, 4, 5)
```

```
// 3. Declarando un array de un tamaño fijo, inicializado con un valor
val array3 = Array(5) { 0 } // Array de 5 elementos, todos ceros

// 4. Arrays de otro tipo específico
val array4 = DoubleArray(5) // Todos los valores serán 0.0 por defecto

// 5. Inicialización personalizada usando un índice
val array5 = Array(5) { i -> i * i } // [0, 1, 4, 9, 16]

// 6. Usando emptyArray para un array vacío, luego se puede llenar
val array6 = emptyArray<String>()
```

Acceso a los elementos de un array:

```
val array = arrayOf(7, 3, 9, 1, 5)
println(array[0]) // Imprime 7
println(array[1]) // Imprime 3
println(array[2]) // Imprime 9
println(array[3]) // Imprime 1
println(array[4]) // Imprime 5
```

Modificación de los elementos de un array:

```
val array = arrayOf(7, 3, 9, 1, 5)
array[0] = 10 // Modifica el primer elemento a 10
println(array[0]) // Imprime 10
```

Eliminación de elementos de un array (creando un nuevo array sin los dos primeros elementos):

```
var array = arrayOf(7, 3, 9, 1, 5)
array = array.drop(2).toTypedArray() // Elimina los dos primeros elementos
println(array.contentToString()) // Imprime [9, 1, 5]
```

Tamaño de un array:

```
val array = arrayOf(7, 3, 9, 1, 5)
println(array.size) // Imprime 5
```

En Kotlin, las **collections** permiten almacenar y gestionar conjuntos de elementos del mismo tipo, con tamaño variable. Existen colecciones **inmutables** (solo lectura), como `List`, `Set` y `Map`, que se crean con `listOf`, `setOf` y `mapOf`, y colecciones **mutables** que permiten modificar su contenido, como `MutableList`, `MutableSet` y `MutableMap`, creadas con `mutableListOf`, `mutableSetOf` y `mutableMapOf`:

```
// Lista inmutable (permite elementos repetidos)
val list: List<Int> = listOf(1, 2, 3, 4, 5)
// Set inmutable (no permite elementos repetidos)
val set: Set<String> = setOf("one", "two", "three")
```

```
// Map inmutable (clave-valor)
val map: Map<String, Int> = mapOf("one" to 1, "two" to 2, "three" to 3)

// Lista mutable
val mutableList: MutableList<Int> = mutableListOf(1, 2, 3, 4, 5)
// Set mutable
val mutableSet: MutableSet<String> = mutableSetOf("one", "two", "three")
// Map mutable
val mutableMap: MutableMap<String, Int> = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
```

Acceso a los elementos de una collection:

```
val list = listOf(1, 2, 3, 4, 5)
println(list[0]) // Imprime 1
println(list[1]) // Imprime 2
println(list[2]) // Imprime 3
println(list[3]) // Imprime 4
println(list[4]) // Imprime 5
```

Modificación de los elementos de una collection (lista mutable):

```
val list = mutableListOf(1, 2, 3, 4, 5)
list[0] = 10 // Modifica el primer elemento a 10
// Ejemplo para eliminar un elemento de una lista mutable:
list.removeAt(0) // Elimina el primer elemento de la lista
println(list[0]) // Imprime 10
```

# Strings

En Kotlin, las cadenas de texto se representan con el tipo `String`, que es una **secuencia inmutable de caracteres**. Aunque no es técnicamente un array, se comporta de manera similar: se puede acceder a caracteres individuales por índice usando corchetes (`str[0]`), obtener su longitud con `str.length`, e iterarlos. Sin embargo, a diferencia de los arrays, los `String` son inmutables, por lo que no puedes modificar un carácter individual directamente.

Se pueden crear de diferentes maneras:

```
val str1 = "Hola, mundo!"
val str2 = """Hola, mundo!"""
val str3 = """
Hola, mundo!
"""
```

Acceso a caracteres individuales:

```
val str = "Hola, mundo!"
println(str[0]) // Imprime 'H'
println(str[1]) // Imprime 'o'
println(str[2]) // Imprime 'l'
```

```
println(str[3]) // Imprime 'a'
println(str[4]) // Imprime ','
```

Longitud de un string:

```
val str = "Hola, mundo!"
println(str.length) // Imprime 12
```

Iteración sobre los caracteres de un string:

```
val str = "Hola, mundo!"
for (char in str) {
    println(char) // Imprime cada carácter del string
}
```

Concatenación de strings:

```
val str1 = "Hola, "
val str2 = "mundo!"
val str3 = str1 + str2
println(str3) // Imprime "Hola, mundo!"
```

Template strings:

```
val name = "Juan"
val age = 20
val str = "Hola, $name! Tienes $age años."
println(str) // Imprime "Hola, Juan! Tienes 20 años."
```

Busqueda de subcadenas:

```
val str = "Hola, mundo!"
println(str.contains("mundo")) // Imprime true
println(str.contains("Hola")) // Imprime true
println(str.contains("hola")) // Imprime false
```

# Flow Control

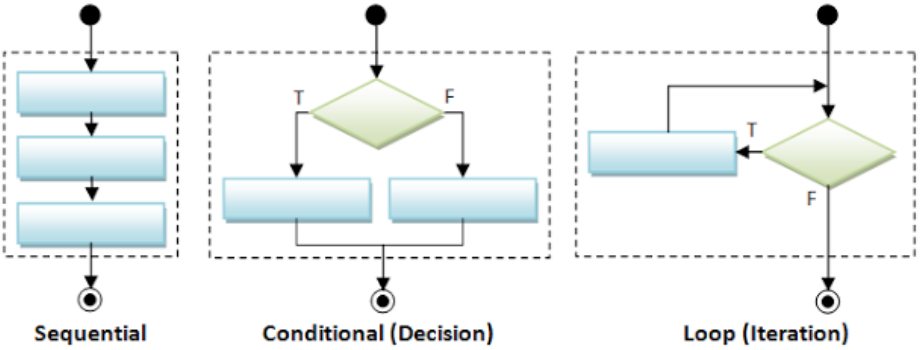


Figura 3: Representación visual «Control Flow».

El control flow se refiere a la forma en la que se condiciona el flujo de ejecución de un programa. En Kotlin, se puede controlar el flujo de ejecución usando las palabras reservadas: `if`, `else`, `when`, `for`, `while`, `do-while`.

```
// Estructura de control if exclusivo
val numero = 7
if (numero > 0) {
    println("El número es positivo")
}
```

```
// Estructura de control if-else
val x = 10
if (x > 5) {
    println("x es mayor a 5")
} else {
    println("x es menor o igual a 5")
}
```

```
// Estructura de control if-else if-else (múltiples condiciones)
val nota = 85
if (nota >= 90) {
    println("Excelente")
} else if (nota >= 80) {
    println("Muy bien")
} else if (nota >= 70) {
    println("Bien")
} else if (nota >= 60) {
    println("Suficiente")
} else {
    println("Insuficiente")
}
```

Ejemplo práctico con argumentos y flujo de control:

```
fun main(args: Array<String>) {
    // Verificar si hay argumentos
    if (args.isEmpty()) {
        println("No se proporcionaron argumentos.")
    } else {
        val comando = args[0]

        // Evaluar el comando recibido
        if (comando == "saludar") {
            println("¡Hola! Bienvenido al programa.")
        } else if (comando == "ayuda") {
            println("Comandos disponibles: saludar, ayuda, salir")
        } else if (comando == "salir") {
            println("¡Hasta luego!")
        } else {
            println("Comando no reconocido: $comando")
        }
    }
}
```

Una forma mas optima:

```
fun main(args: Array<String>) {
    // Verificar si hay argumentos
    if (args.isEmpty()) {
        println("No se proporcionaron argumentos.")
        return
    }

    val comando = args[0]

    // Evaluar el comando recibido
    if (comando == "saludar") {
        println("¡Hola! Bienvenido al programa.")
    } else if (comando == "ayuda") {
        println("Comandos disponibles: saludar, ayuda, salir")
    } else if (comando == "salir") {
        println("¡Hasta luego!")
    } else {
        println("Comando no reconocido: $comando")
    }
```

Ejemplo con números:

```
fun main(args: Array<String>) {
    if (args.isEmpty()) {
        println("No se proporcionaron argumentos.")
        return
    }

    val numero = args[0].toInt()

    if (numero > 0) {
        println("El número es positivo")
```

```
    } else if (numero < 0) {
        println("El número es negativo")
    } else {
        println("El número es cero")
    }
}
```

Aca les dejo un enlace de algunos ejemplos para ejecutar en el playground de Kotlin: https://pl.kotl.in/-VEJh7fdl

# Ejercicios

Crees ser capaz de resolver los siguientes ejercicios?

- https://leetcode.com/problems/convert-the-temperature
- https://leetcode.com/problems/length-of-last-word

Aca les dejo un enlace de algunos ejemplos para ejecutar en el playground de Kotlin: https://pl.kotl.in/-VEJh7fdl

Los loops se usan para repetir un bloque de código un número determinado de veces. En Kotlin, se pueden usar las palabras reservadas: `for`, `while`, `do-while`.

Loop `for` con rango (1 al 5):

```
for (i in 1..5) {
    println("El número es $i")
}
// Imprime: 1, 2, 3, 4, 5
```

Loop `for` con rango descendente:

```
for (i in 5 downTo 1) {
    println("El número es $i")
}
// Imprime: 5, 4, 3, 2, 1
```

Loop `for` con pasos (incremento de 2):

```
for (i in 0..10 step 2) {
    println("El número es $i")
}
// Imprime: 0, 2, 4, 6, 8, 10
```

Loop `for` iterando sobre un array:

```
val frutas = arrayOf("Manzana", "Banana", "Naranja")
for (fruta in frutas) {
    println("Fruta: $fruta")
}
// Imprime cada fruta del array
```

Loop `for` con índice y valor:

```
val colores = listOf("Rojo", "Verde", "Azul")
for ((indice, color) in colores.withIndex()) {
    println("Color $indice: $color")
}
// Imprime: Color 0: Rojo, Color 1: Verde, Color 2: Azul
```

Loop `while`:

```
var i = 1
while (i <= 5) {
    println("El número es $i")
    i++
}
// Imprime: 1, 2, 3, 4, 5
```

Loop `do-while` (ejecuta al menos una vez):

```
var contador = 1
do {
    println("Contador: $contador")
    contador++
} while (contador <= 3)
// Imprime: Contador: 1, Contador: 2, Contador: 3
```

Uso de `break` para salir del loop:

```
for (i in 1..10) {
    if (i == 5) {
        break  // Sale del loop cuando i es 5
    }
    println(i)
}
// Imprime: 1, 2, 3, 4
```

Uso de `continue` para saltar una iteración:

```
for (i in 1..5) {
    if (i == 3) {
        continue  // Salta la iteración cuando i es 3
    }
    println(i)
}
// Imprime: 1, 2, 4, 5 (se salta el 3)
```

Ejemplo práctico - Suma de números pares:

```
var suma = 0
for (i in 1..10) {
    if (i % 2 == 0) {
        suma += i
```

```
    }
}
println("La suma de números pares del 1 al 10 es: $suma")
// Imprime: La suma de números pares del 1 al 10 es: 30
```

# Ejemplos

Secuencia de Fibonacci usando un loop `for`:

```
fun main() {
    val n = 10
    val fibonacci = mutableListOf(0, 1)
    for (i in 2..n) {
        fibonacci.add(fibonacci[i-1] + fibonacci[i-2])
    }
}
```

Sumar elementos de un array:

```
fun main() {
    val array = arrayOf(1, 2, 3, 4, 5)
    var suma = 0
    for (i in array) {
        suma += i
    }
}
```

Buscar un elemento en un array:

```
fun main() {
    val array = arrayOf(1, 2, 3, 4, 5)
    val elemento = 3
    if (elemento in array) {
        println("El elemento $elemento se encuentra en el array")
    }
}
```

Recorrer un array en orden inverso usando indices:

```
fun main() {
    val array = arrayOf(1, 2, 3, 4, 5)
    for (i in array.size - 1 downTo 0) {
        println(array[i])
    }
}
// Imprime: 5, 4, 3, 2, 1
```

Recorrer un array en orden inverso usando el método `reversed()`:

```
fun main() {
    val array = arrayOf(1, 2, 3, 4, 5)
```

```
    for (i in array.reversed()) {
        println(i)
    }
}
```

Calcular el promedio de un array:

```
fun main() {
    val array = arrayOf(1, 2, 3, 4, 5)
    val promedio = array.average()
    println(promedio)
}
```

Obtener la mediana de un array:

```
fun main() {
    val array = arrayOf(1, 2, 3, 4, 5)
    val mediana = array.sorted()[array.size / 2]
    println(mediana)
}
```

Programa para calcular n numeros primos:

```
// Función para verificar si un número es primo
fun esPrimo(numero: Int): Boolean {
    if (numero <= 1) return false
    if (numero == 2) return true
    if (numero % 2 == 0) return false

    for (i in 3..Math.sqrt(numero.toDouble()).toInt() step 2) {
        if (numero % i == 0) {
            return false
        }
    }
    return true
}

// Función para encontrar los primeros n números primos
fun encontrarPrimos(n: Int): List<Int> {
    val primos = mutableListOf<Int>()
    var numero = 2

    while (primos.size < n) {
        if (esPrimo(numero)) {
            primos.add(numero)
        }
        numero++
    }

    return primos
}

// Ejemplo de uso
fun main() {
```

```
    val cantidadPrimos = 10
    val primos = encontrarPrimos(cantidadPrimos)

    println("Los primeros $cantidadPrimos números primos son:")
    println(primos)

    // Verificar si un número específico es primo
    val numeroAVerificar = 17
    if (esPrimo(numeroAVerificar)) {
        println("$numeroAVerificar es un número primo")
    } else {
        println("$numeroAVerificar no es un número primo")
    }
}
// Imprime: Los primeros 10 números primos son:
// [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
// 17 es un número primo
```

# Clases y Objetos

La siguiente es una forma de definir una clase en Kotlin:

```
fun main() {
    val persona = Persona("Juan", 25)
    println(persona.nombre)
    println(persona.edad)
    persona.saludar()
}

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

Usando el constructor secundario:

```
fun main() {
    // Usando constructor primario
    val persona1 = Persona("Juan", 25)
    println(persona1.nombre)
    println(persona1.edad)

    // Usando constructor secundario
    val persona2 = Persona("María")
    println(persona2.nombre)
    println(persona2.edad)
}

class Persona(val nombre: String, var edad: Int) {
    // Constructor secundario con un solo parámetro
    constructor(nombre: String) : this(nombre, 0) {
        println("Constructor secundario llamado")
```

```
    }

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

Clase sin constructor primario (solo secundario):

```
fun main() {
    val persona = Persona("Juan", 25)
    println(persona.nombre)
    println(persona.edad)
    persona.saludar()
}

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

    // Constructor secundario
    constructor(nombre: String, edad: Int) {
        this.nombre = nombre
        this.edad = edad
        println("Constructor secundario llamado")
    }

    fun saludar() {
        println("Hola, soy $nombre y tengo $edad años")
    }
}
```