

FUNDAMENTOS DE KOTLIN PARA ANDROID - TEORÍA COMPLETA

INTRODUCCIÓN A KOTLIN

Kotlin es un lenguaje de programación moderno desarrollado por JetBrains que se ejecuta sobre la Máquina Virtual de Java (JVM). A pesar de compilarse a bytecode Java, Kotlin introduce características de lenguajes funcionales, seguridad de tipos mejorada y una sintaxis concisa que lo hace especialmente atractivo para el desarrollo de aplicaciones Android. Este documento cubre los fundamentos esenciales del lenguaje, desde conceptos básicos hasta estructuras de control avanzadas, con énfasis en la preparación para desarrollo Android mediante Android Studio.

FUNDAMENTOS DEL SISTEMA DE TIPOS EN KOTLIN

Jerarquía de tipos

El sistema de tipos de Kotlin se basa en una jerarquía clara y consistente:

- **Any:** tipo base de todas las clases en Kotlin (equivalente a Object en Java)
- **Tipos primitivos:** Int, Long, Short, Byte, Float, Double, Boolean, Char, String
- **Tipos compuestos:** Array, List, Set, Map, Pair, Triple
- **Tipos nulos:** Any?, String? (tipos nullable con soporte para null-safety)
- **Tipos funcionales:** (Int, String) -> Boolean (funciones como tipos de primera clase)
- **Null-safety:** Kotlin diferencia tipos que pueden ser nulos (String?) de los que no pueden (String)
- **Type inference:** el compilador deduce tipos automáticamente en la mayoría de casos
- **Smart casting:** conversión automática de tipos en contextos seguros

Arquitectura de memoria en Kotlin:

- **Stack:** almacenamiento de referencias y tipos primitivos (acceso rápido)
 - **Heap:** almacenamiento de objetos complejos (acceso más lento)
 - **Recolector de basura:** JVM libera automáticamente memoria no utilizada
 - **Variance:** covarianza y contravarianza en tipos genéricos para mayor seguridad
-

CREACIÓN Y ASIGNACIÓN DE VARIABLES

Declaración de variables con val

val (value) declara variables inmutables: una vez asignadas, no pueden cambiar. El compilador garantiza que el valor no se modifique, permitiendo optimizaciones agresivas.

Limitaciones importantes:

- La variable no puede reasignarse: nombre = "Juan" genera error de compilación
- El objeto apuntado puede ser mutable (por ejemplo, una lista dentro de val)
- val no implica que el contenido sea inmutable, solo que la referencia no cambia

Cuándo usar val:

- Variables de configuración: val PUERTO = 8080
- Valores computados una sola vez: val resultado = calcularAlgo()
- Parámetros de función: todos son val implícitamente
- Evitar: nunca para valores que cambien durante ejecución

Descripción técnica detallada:

La compilación de val ciudad = "Madrid" genera bytecode que:

1. Crea una referencia en el stack
2. Asigna la cadena al heap
3. Marca la referencia como final (no reasignable)

4. El compilador inserta verificaciones en tiempo de compilación si se intenta modificar

Declaración de variables con var

var (variable) declara variables mutables: pueden cambiar de valor múltiples veces. El compilador permite reasignación pero realiza menos optimizaciones.

Limitaciones y consideraciones:

- Reasignación cambia solo el valor, no el tipo de la variable
- `var x: Int = 5; x = "texto"` genera error (type mismatch)
- Variables `var` en closures capturan referencias mutables (posibles race conditions en threads)
- No thread-safe: acceso desde múltiples hilos requiere sincronización

Cuándo usar var:

- Contadores y acumuladores: `var suma = 0`
- Estado que cambia: `var posicion = 10`
- Iteradores: `var indice = 0`
- Evitar: preferir `val` siempre que sea posible (mejor rendimiento y seguridad)

Descripción técnica de reasignación:

Cuando declaras `var x: Int = 10` y luego `x = 20`:

1. La referencia en el stack apunta inicialmente a 10
2. Al ejecutar `x = 20`, se crea nuevo valor y la referencia se reasigna
3. El valor 10 se marca para recolección de basura si no hay otras referencias
4. El compilador no puede asumir que el valor no cambie (menos optimizaciones)

Inferencia de tipos

El compilador de Kotlin deduce tipos automáticamente analizando el contexto de asignación. Este proceso denominado **type inference** ocurre en tiempo de compilación, sin impacto en rendimiento en tiempo de ejecución.

Arquitectura del proceso de inferencia:

1. Análisis de contexto derecha a izquierda
2. Unificación de tipos mediante algoritmos de constraint solving

3. Generación de código bytecode con tipos verificados
4. Sin información de tipo en runtime (type erasure, heredado de Java)

Riesgos de truncamiento:

Cuando conviertes de un tipo más grande a uno más pequeño, pueden perderse datos. Es importante validar los rangos.

NULL-SAFETY: EL SISTEMA DE TIPOS OPCIONALES

La característica más revolucionaria de Kotlin es su manejo seguro de valores nulos. En Java, `NullPointerException` es la causa más común de crashes. Kotlin resuelve esto con tipos opcionales (nullable types) en tiempo de compilación.

Tipos nullable vs non-nullable

- **Non-nullable:** no puede ser null (compilador lo garantiza)
- **Nullable:** puede ser null (se declara con ?)
- **Type inference with null:** debe especificarse en declaración real

Operador Elvis (?) y acceso seguro

- **Acceso directo:** solo si es non-nullable (genera error)
- **Acceso seguro con ?.:** Retorna tipo nullable (null si es null)
- **Elvis operator ?::** proporciona valor por defecto
- **!! (not-null assertion):** lanza excepción si es null (debe usarse solo cuando estés 100% seguro)
- **Short-circuit evaluation** (importante para rendimiento): El segundo operando NO se evalúa si el primero lo determina.

Descripción técnica del compilador:

- Non-nullable: no genera código de verificación de null
 - Nullable: genera verificaciones automáticas antes de acceso
 - Elvis operator: compilado a bytecode condicional (sin overhead en runtime)
 - **!! (not-null assertion):** debe usarse solo cuando estés 100% seguro
-

STRINGS Y PLANTILLAS DE INTERPOLACIÓN

Kotlin proporciona sistemas potentes de strings con interpolación de variables nativa, evitando la concatenación engorrosa de Java.

Características principales

- **String básico:** similar a Java
- **String con saltos de línea:** raw strings multilinea con `trimIndent()`
- **Interpolación de variables:** con sintaxis `$nombre`
- **Interpolación de expresiones:** con sintaxis `${expresion}`
- **Expresiones complejas en templates:** operadores, llamadas a función, etc.
- **Raw strings:** sin escape de caracteres especiales (path de archivos, regex)

Ventajas sobre concatenación Java

- Más legible y menos propenso a errores
 - Evaluación segura de nulls dentro de plantillas
 - Rendimiento superior (optimizado por compilador)
 - Menos código boilerplate
-

FUNCIONES Y PARÁMETROS

Declaración de funciones básicas

Una función es un bloque de código reutilizable que realiza una tarea específica. Kotlin trata las funciones como ciudadanos de primera clase, permitiendo asignarlas a variables y pasarlas como parámetros.

Sintaxis fundamental:

- Función con parámetros y valor de retorno
- Función con cuerpo de expresión (return implícito)
- Función sin parámetros
- Función sin valor de retorno (Unit es el equivalente de void)
- Omitir Unit cuando es obvio
- Función con cuerpo de expresión sin parámetros

Descripción técnica de compilación:

Cuando compilas una función:

1. El compilador verifica tipos de parámetros
2. Verifica que el tipo de retorno coincida con la expresión
3. Genera bytecode de método estático o instancia según contexto
4. El JIT compiler en runtime optimiza secuencias comunes

Parámetros por defecto

Kotlin permite asignar valores por defecto a parámetros, reduciendo la necesidad de múltiples sobrecargas de función (como en Java).

Ventajas sobre Java:

- Sin necesidad de múltiples constructores sobrecargados
- Código más legible con argumentos nombrados
- Mantenimiento más fácil (agregar parámetros no rompe código)

Parámetros variables (varargs)

Permite pasar un número variable de argumentos del mismo tipo a una función.

Nota: Solo un parámetro vararg por función, generalmente al final

Funciones de extensión

Una característica poderosa de Kotlin: añadir funciones a tipos existentes sin herencia.

Características:

- Extender tipos primitivos, strings y tipos genéricos
 - Funciones de extensión en tipos genéricos con parámetros tipo
 - Sintaxis: `fun TipoBase.nombreFuncion(): RetornoTipo`
-

ESTRUCTURAS DE CONTROL: CONDICIONALES

Sentencia if-else

La estructura condicional básica que ejecuta diferentes bloques según condiciones booleanas.

Características:

- if simple
- if-else
- if-else if-else encadenado
- if como expresión (retorna valor)
- if sin else retorna Unit

Diferencia con Java: En Kotlin, if es una expresión (retorna valor), no solo una sentencia.

Sentencia when

Equivalente mejorado de switch-case en Java, más potente y flexible.

Características:

- when básico con valores
- when con rangos (in 0..12)
- when con tipos (smart casting con is)
- when con expresiones complejas
- when con múltiples valores por rama

Ventajas sobre switch de Java:

- No requiere break (no hay caída entre ramas)
- Puede usar rangos, tipos, expresiones
- Retorna valor directamente
- Compilador verifica que todos los casos estén cubiertos

Operadores lógicos avanzados

Manejo de condiciones complejas en Kotlin.

Operadores fundamentales:

- Operador AND (&&)
- Operador OR (||)
- Operador NOT (!)
- Combinaciones complejas
- Short-circuit evaluation

Operador in (pertenencia a rango):

- Verifica si un valor está dentro de un rango
 - Negación con !in
-

ESTRUCTURAS DE CONTROL: BUCLES

Bucle for

Itera sobre colecciones, rangos y estructuras iterables.

Variantes:

- for sobre rangos (1..5)
- Rango exclusivo (1.. <5)
- Rango descendente (5 downTo 1)
- Rango con paso (step)
- for sobre lista
- for con índice (withIndex())
- for sobre String
- for sobre mapa (destructuring)

Descripción técnica de compilación:

- for i in 1..5: compilado a bytecode similar a while con iterador
- Rangos son objetos IntRange optimizados, no arrays intermedios
- Destructuring en for ((k, v) in map) genera código de desempaqueado

Bucle while y do-while

Bucles controlados por condición booleana.

- **while:** condición se verifica antes de cada iteración
- **do-while:** cuerpo se ejecuta al menos una vez
- **Condiciones complejas:** combinación de operadores lógicos

Sentencias de control de flujo: break y continue

Permiten controlar el comportamiento del bucle.

- **break:** sale del bucle actual
- **continue:** salta a siguiente iteración
- **break con etiqueta:** sale de bucle externo (outer@)
- **continue con etiqueta:** siguiente iteración del bucle externo

Uso práctico: búsqueda con salida usando return

Iteradores funcionales: forEach, map, filter

Enfoques modernos de iteración con funciones de orden superior.

Funciones disponibles:

- **forEach:** ejecuta bloque para cada elemento
 - **map:** transforma cada elemento
 - **filter:** selecciona elementos que cumplen condición
 - **Encadenamiento de operaciones (composición funcional)**
 - **any:** verifica si algún elemento cumple
 - **all:** verifica si todos cumplen
 - **find:** primer elemento que cumple
 - **count:** cuenta elementos que cumplen
 - **reduce:** combina elementos
-

COLECCIONES: LISTAS, CONJUNTOS Y MAPAS

Listas: creación y operaciones

Las listas son colecciones ordenadas que permiten duplicados.

Tipos de listas:

- Crear lista inmutable (read-only)
- Crear lista mutable
- Acceso a elementos
- Operaciones de lista
- Transformaciones
- Ordenamiento
- Tomar elementos

Operaciones importantes:

- `contains()`, `indexOf()`
- `getOrNull()`, `getOrNull()`
- `map()`, `filter()`
- `sorted()`, `sortedDescending()`
- `take()`, `takeLast()`, `drop()`
- `flatten()`

Conjuntos (Sets): colecciones únicas

Los conjuntos no permiten elementos duplicados, optimizados para búsquedas rápidas.

Características:

- Crear conjunto inmutable
- Crear conjunto mutable
- Operaciones de conjunto
- Operaciones matemáticas de conjuntos
- `union()`, `intersect()`, `subtract()`
- Conversión

Mapas: pares clave-valor

Estructuras que asocian claves únicas a valores.

Características:

- Crear mapa inmutable
- Acceder a valores
- Crear mapa mutable
- Modificar mapa
- Iterar sobre mapa
- Operaciones de mapa
- Transformaciones

Operaciones importantes:

- size, keys, values
 - containsKey(), containsValue()
 - mapValues(), filter()
 - getOrDefault()
-

FUNCIONES LAMBDA Y FUNCIONES DE ORDEN SUPERIOR

Introducción a lambdas

Las lambdas son funciones anónimas que pueden asignarse a variables o pasarse como parámetros. Son fundamentales en Kotlin y especialmente útiles en callbacks para Android.

Características:

- Lambda básica asignada a variable
- Lambda con un solo parámetro (implícito 'it')
- Lambda sin parámetros
- Lambda con múltiples líneas
- Tipo de función como parámetro

Ventajas sobre Java:

- Sintaxis concisa (sin new o clases anónimas)
- Cierre de variables del contexto automático
- Optimizadas por compilador (inlining)

Pasar lambdas como argumentos

La forma más común de usar lambdas en Kotlin: callbacks y procesamiento funcional.

Características:

- Si la lambda es el último parámetro, puede ir fuera de paréntesis
- Sintaxis preferida: nombreFuncion { }
- Encadenamiento de operaciones
- Lambda con múltiples parámetros
- Destructuring en lambdas

Funciones de orden superior

Funciones que reciben o retornan otras funciones.

Características:

- Función que recibe una función como parámetro
- Función que retorna una función
- Composición de funciones
- Crear nuevas funciones combinando otras

Callbacks y manejo de eventos (preparación para Android)

Patrón común en Android para responder a eventos del usuario.

Evolución del patrón:

- Interfaz de callback (forma antigua, no recomendada)
- Forma moderna con lambda (recomendada)
- Callback con parámetros
- Manejo múltiple de callbacks (onSuccess, onError)
- Patrón builder con lambdas (común en Android)

EXCEPCIONES Y MANEJO DE ERRORES

Try-catch-finally

Mecanismo de manejo seguro de errores en tiempo de ejecución.

Características:

- Try-catch básico
- Múltiples catch para diferentes excepciones
- Try-catch-finally
- Try como expresión (retorna valor)
- Try con recursos (automático try-with-resources de Java)

Excepciones personalizadas

Crear tipos de excepciones específicas para tu aplicación.

Características:

- Excepción personalizada simple
- Excepción personalizada con detalles
- Uso y lanzamiento de excepciones
- Propagar excepciones

Result y Either: alternativas funcionales

Enfoques modernos para manejo de errores sin excepciones.

Características:

- Usando Result<T>
 - Pattern matching con Result
 - Chainable operations
 - onSuccess, onFailure
 - map, recover para transformaciones
-

EJEMPLOS DE INTRODUCCIÓN

Ejemplo 1: Programa de bienvenida

Crea un programa que pide el nombre del usuario y muestre un mensaje personalizado.

Ejemplo 2: Calculadora simple

Implementa una calculadora que realiza operaciones básicas.

Ejemplo 3: Lista de tareas

Programa que gestiona una lista de tareas pendientes.

Ejemplo 4: Procesamiento de números

Filtra y transforma una lista de números.

Ejemplo 5: Validación de datos de usuario

Valida información de usuario con manejo robusto de errores.

EJERCICIOS OBLIGATORIOS

Ejercicio 1: Sistema de gestión de biblioteca (Ejercicio1.kt)

Objetivo: Crear un sistema para gestionar libros en una biblioteca con búsqueda y filtrado avanzado.

Funcionalidades requeridas:

- Buscar libros por autor
- Buscar libros por rango de años
- Obtener solo los libros disponibles
- Calcular estadísticas de la biblioteca

Buenas prácticas:

- Usar data class para Libro
- Aprovechar funciones de orden superior (filter, map)
- Manejo seguro de nulls con operador ?
- Nombres descriptivos de variables
- Comentarios en lógica compleja

Conceptos clave a aprender

- data class
- filter y map
- groupBy
- maxByOrNull
- Acceso a objetos con ?

Pasos principales

1. Define data class Libro ← (completado)
2. Crea lista de libros ← (completado)
3. Implementa **buscarPorAutor()** ← Usa *filter*
4. Implementa **buscarPorAño()** ← Usa *filter* con rango
5. Implementa **librosDisponibles()** ← Filtra por disponible
6. Implementa **calcularEstadisticas()** ← Combina size, map, groupBy
7. Prueba llamando las funciones ← (completado)

Extra

- Agregar búsqueda por título
- Mostrar libros ordenados por año
- Permitir agregar nuevos libros a la lista

Errores comunes

- Usar == en lugar de **contains**
- Olvidar **lowercase()** para búsquedas insensibles a mayúsculas
- No acceder correctamente a **groupBy().maxByOrNull()**

Ejercicio 2: Conversor de temperaturas con validación (Ejercicio2.kt)

Objetivo: Crear un conversor que traduce entre escalas de temperatura con validación robusta de entrada.

Funcionalidades requeridas:

- Convertir Celsius a Fahrenheit
- Convertir Kelvin a Celsius
- Validar si una temperatura es física y teóricamente posible
- Interfaz interactiva para conversiones múltiples

Buenas prácticas:

- Validación de entrada con excepciones personalizadas
- Manejo de valores edge cases (-273.15°C, 0K)
- Result<Double> para manejo seguro de errores
- Menú interactivo con bucles

Conceptos clave a aprender

- Excepciones personalizadas
- Result<T>
- onSuccess y onFailure
- Validación de datos
- Menú interactivo con while y when

Pasos principales

1. Define excepciones ← (completado)
2. Define data class ← (completado)
3. Implementa **celsiusAFahrenheit()**
4. Implementa **kelvinACelsius()**
5. Implementa **validarTemperatura()** ← ≥ -273.15
6. Implementa **convertir()** ← Valida y transforma
7. Implementa **menuInteractivo()**
8. Prueba las conversiones ← (completado)

Extra

- Conversión inversa
- Guardar historial
- Alertas de temperatura extrema

Errores comunes

- División entera ($5/2 = 2$)
- No convertir a Double (usar $5/2.0$)
- No manejar **toDoubleOrNull()**
- Olvidar **return** cuando hay error

Ejercicio 3: Analizador de textos con estadísticas (Ejercicio3.kt)

Objetivo: Analizar un texto y generar estadísticas detalladas sobre su contenido.

Funcionalidades requeridas:

- Analiza un texto completo y retorna objeto con todas las estadísticas
- Cuenta la frecuencia de palabras
- Busca patrones en el texto

Buenas prácticas:

- Usar data class para resultados
- Normalización de texto (minúsculas, sin puntuación)
- Funciones de transformación encadenadas
- Manejo de strings y expresiones regulares

Conceptos clave a aprender:

- Normalización de strings
- Expresiones regulares básicas
- `split()` y `filter()`
- `groupBy` para frecuencias
- `map` para transformaciones
- `average()` para promedios

Pasos principales:

1. Define data class para resultados ← (completado)
2. Implementa **normalizarTexto()** ← Minúsculas + remove puntuación
3. Implementa **contarCaracteres()** ← Replace + length
4. Implementa **contarPalabras()** ← Split + filter + size
5. Implementa **contarLineas()** ← Split por \n
6. Implementa **encontrarPalabraMasFrecuente()** ← groupBy + maxByOrNull
7. Implementa **longitudPromedioPalabras()** ← map + average()
8. Implementa **analizarTexto()** ← Combina todo
9. Prueba con texto de ejemplo ← (completado)

Desafío extra:

Encontrar frases más frecuentes (3+ palabras)

Análisis de sentimiento (palabras positivas/negativas)

Exportar análisis a archivo

Buscar palabras con patrón (comienzan con cierta letra)

Errores comunes:

Olvidar toLowerCase() en búsquedas de frecuencia

No filtrar palabras vacías después de split

Usar [] en lugar de .filter {} después de split

Regex incorrecta (olvidar backslash o caracteres especiales)

Ejercicio 4: Gestor de contactos (Ejercicio4.kt)

Objetivo: Crear un gestor completo de contactos con búsqueda, filtrado y guardado.

Funcionalidades requeridas:

- Validar formato de email

- Validar formato de teléfono
- Crear nuevo contacto con validación
- Buscar contactos por nombre (búsqueda parcial)
- Obtener contactos favoritos
- Ordenar contactos alfabéticamente
- Menú interactivo principal

Buenas prácticas:

- Validación robusta de datos
- Use Result<T> para manejo de errores
- Operaciones CRUD completas
- Menú interactivo con try-catch

Conceptos clave a aprender

- Validación robusta
- Result<T>
- find() y removelf()
- Alternar booleanos
- CRUD completo
- Menú interactivo avanzado

Pasos principales

10. Define data class Contacto ← (completado)
11. Define excepciones ← (completado)
12. Implementa **validarEmail()**
13. Implementa **validarTelefono()**
14. Implementa **validarNombre()**
15. Implementa **crearContacto()**
16. Implementa **buscarPorNombre()**
17. Implementa **obtenerFavoritos()**
18. Implementa **obtenerOrdenados()**

19. Implementa **toggleFavorito()**
20. Implementa **eliminarContacto()**
21. Implementa **mostrarContacto()**
22. Implementa **menuInteractivo()**
23. Prueba con ejemplos ← (completado)

Extra

- Editar contacto
- Validación email más estricta
- Búsqueda por teléfono / email
- Exportar a CSV
- Importar de archivo

Errores comunes

- Validar solo 1 campo, deben ser todos
- Usar var en vez de val cuando no corresponde
- Olvidar **?: continue**
- Uso incorrecto de **find()**
- **toggleFavorito()** no actualiza lista

EJERCICIOS OPCIONALES

Ejercicio Opcional 1: Juego de adivinanza de números (Opcional1.kt)

Objetivo: Implementar un juego interactivo donde el usuario adivina un número aleatorio con pistas.

Requisitos:

- Generar número aleatorio entre 1 y 100
- Permitir múltiples intentos
- Dar pistas (mayor/menor)

- Mostrar cantidad de intentos y récord

Funcionalidades requeridas:

- Bucles
- Random
- Variables de estado
- (1..100).random()
- Int.MAX_VALUE para récord

Ejercicio Opcional 2: Aplicación de notas rápidas (Opcional2.kt)

Objetivo: Crear un sistema de notas que permite crear, buscar, modificar y eliminar notas.

Características:

- Crear nota con título y contenido
- Buscar por título o contenido
- Marcar notas como importantes
- Mostrar notas por fecha de creación
- Exportar notas a formato texto

Requisitos técnicos:

- Data class para Nota
- Timestamps de creación
- Búsqueda con filter y map
- Manejo de ficheros básico

Funcionalidades requeridas:

- LocalDateTime.now()
- Exportar con writeText
- StringBuilder para iterar

Ejercicio Opcional 3: Calculadora de IMC con historial (Opcional3.kt)

Objetivo: Crear una calculadora de Índice de Masa Corporal que mantiene historial de mediciones.

Requisitos:

- Calcular IMC: $\text{peso(kg)} / (\text{altura(m)})^2$
- Validar entrada (peso y altura positivos)
- Clasificar según categorías OMS
- Mantener historial de cálculos
- Mostrar tendencias (ganancia/pérdida de peso)

Categorías:

- Bajo peso: $\text{IMC} < 18.5$
- Peso normal: $18.5 - 24.9$
- Sobrepeso: $25.0 - 29.9$
- Obesidad: ≥ 30

Funcionalidades requeridas:

- $\text{peso} / (\text{altura}^2)$
- when con rangos
- `historial.last()`
- Diferencia entre valores