Construcción de aplicaciones móviles

```
Programa de Ingeniería de Software

Título: Programación funcional en Kotlin
Duración estimada: 60
Docente: Carlos Andrés Florez
Guía: 03
```

Programación Funcional en Kotlin

Introducción

La programación funcional se basa en varios pilares fundamentales que la diferencian de otros paradigmas como la programación imperativa u orientada a objetos. Este paradigma es la base de Jetpack Compose para crear Apps móviles en Android y es ampliamente soportado en Kotlin. Los pilares clave son:

1. Inmutabilidad

Los datos no cambian una vez que se crean. En lugar de modificar estructuras existentes, se crean nuevas versiones con los cambios aplicados. Esto reduce errores relacionados con efectos secundarios y hace el código más predecible.

Ejemplo básico:

```
// Inmutable - usando val
val lista = listOf(1, 2, 3)
val nuevaLista = lista + 4 // Crea una nueva lista

// Evitar - mutable
var listaMutable = mutableListOf(1, 2, 3)
listaMutable.add(4) // Modifica la lista original
```

Data classes inmutables:

```
data class Usuario(val nombre: String, val edad: Int)
val usuario = Usuario("Juan", 25)
val usuarioActualizado = usuario.copy(edad = 26) // Nueva instancia
```

2. Funciones Puras

Una función es pura si:

- Siempre devuelve el mismo resultado para los mismos argumentos
- No tiene efectos secundarios (no modifica variables externas, bases de datos o el estado global)

Esto facilita la depuración y las pruebas unitarias.

Función pura:

```
fun sumar(a: Int, b: Int): Int = a + b // Siempre devuelve lo mismo
fun calcularArea(radio: Double): Double = Math.PI * radio * radio
```

Función impura (evitar):

```
var contador = 0
fun incrementarContador(): Int {
    contador++ // Efecto secundario
    return contador
}
```

3. Expresiones Lambda

Son funciones anónimas que pueden almacenarse en variables o pasarse como argumentos.

Sintaxis básica:

```
val lambda = { x: Int, y: Int -> x + y }
val resultado = lambda(5, 3) // 8

// Lambda con un solo parámetro (it implícito)
val numeros = listOf(1, 2, 3, 4, 5)
val pares = numeros.filter { it % 2 == 0 }
```

4. Funciones de Orden Superior

Las funciones pueden recibir otras funciones como parámetros o devolver funciones como resultado.

Recibir función como parámetro:

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

val suma = aplicarOperacion(5, 3) { x, y -> x + y }
val multiplicacion = aplicarOperacion(5, 3) { x, y -> x * y }
```

Devolver una función:

```
fun crearMultiplicador(factor: Int): (Int) -> Int {
    return { numero -> numero * factor }
}

val duplicar = crearMultiplicador(2)
val triplicar = crearMultiplicador(3)
println(duplicar(5)) // 10
```

5. Evaluación Perezosa (Lazy Evaluation)

Los valores y expresiones solo se calculan cuando realmente se necesitan. Permite trabajar con estructuras potencialmente infinitas y optimiza el rendimiento.

Lazy property:

```
class ExpensiveResource {
    val data: String by lazy {
        println("Calculando datos...")
        "Datos calculados" // Solo se ejecuta la primera vez
    }
}
```

Sequences para evaluación perezosa:

```
val numeros = generateSequence(1) { it + 1 } // Secuencia infinita
val primerosCinco = numeros.take(5).toList() // Solo calcula los primeros 5

// Vs eager evaluation con listas
val numerosEager = (1..1000000).map { it * 2 }.filter { it > 100 }
val numerosLazy = (1..1000000).asSequence().map { it * 2 }.filter { it > 100 }.take(10).toLis
```

6. Composición de Funciones

En lugar de usar estructuras de control tradicionales, se combinan funciones pequeñas para crear funciones más complejas.

Composición con funciones de extensión:

```
fun String.esPalindromo(): Boolean = this == this.reversed()
fun String.limpiar(): String = this.trim().lowercase()

fun validarPalabra(palabra: String): Boolean {
    return palabra.limpiar().esPalindromo()
}
```

Composición con operadores:

```
val procesarTexto = { texto: String ->
    texto.trim()
        .lowercase()
        .split(" ")
        .filter { it.isNotEmpty() }
        .map { it.capitalize() }
        .joinToString(" ")
}
```

Operaciones funcionales sobre colecciones

Son métodos que aplican los principios de programación funcional para trabajar con colecciones (listas, arrays, etc.) de manera más elegante y eficiente.

Características principales:

- No modifican la colección original (inmutabilidad)
- Devuelven una nueva colección con los cambios aplicados
- Usan funciones lambda como parámetros
- Son funciones puras (mismo input = mismo output)

Las más importantes:

`map` - Transformación

```
val numeros = listOf(1, 2, 3, 4, 5)
val cuadrados = numeros.map { it * it } // [1, 4, 9, 16, 25]
```

`filter` - Filtrado

```
val pares = numeros.filter { it % 2 == 0 } // [2, 4]
```

`reduce` y `fold` - Reducción

```
val suma = numeros.reduce { acc, n -> acc + n } // 15
val sumaConInicial = numeros.fold(10) { acc, n -> acc + n } // 25
```

`flatMap` - Aplanamiento

```
val palabras = listOf("hola mundo", "kotlin funcional")
val todasLasPalabras = palabras.flatMap { it.split(" ") }
// ["hola", "mundo", "kotlin", "funcional"]
```

Funciones de Ámbito en Kotlin

Kotlin incluye varias funciones de alcance: `apply`, `let`, `also`, `with`, `run` que simplifican la escritura de código más conciso, legible y expresivo.

Estas funciones permiten ejecutar bloques de código en un contexto determinado, evitando variables temporales innecesarias, facilitando la inicialización de objetos y promoviendo un estilo de programación más funcional.

Relación con Jetpack Compose

En Jetpack Compose, la programación funcional es fundamental:

```
@Composable
fun Saludo(nombre: String) { // Función pura
    Text(text = "Hola $nombre") // Inmutable
}

@Composable
fun Contador() {
    var count by remember { mutableStateOf(0) } // Estado manejado funcionalmente

    Column {
        Text("Contador: $count")
        Button(onClick = { count++ }) { // Lambda
            Text("Incrementar")
        }
    }
}
```

Actividades Prácticas

1. Para cada uno de los pilares de la programación funcional mencionados, desarrolle dos ejemplos adicionales que ilustren cada concepto.
2. Cree un ejemplo práctico que use las cinco funciones de ámbito (`apply` , `let` , `also` , `with` , `run`) en un solo programa que modele una aplicación de gestión de usuarios.

Recursos Adicionales

- Jetpack Compose Thinking in Composition
- Higher-Order Functions and Lambdas