Programación Multimedia y Dispositivos Móviles

En esta página encontraréis todo el material que iremos viendo durante el curso.

Esta página servirá básicamente de guía y contendrá ejemplos, enlaces a otros repositorios con proyectos y ejercicios que iremos viendo en clase, o pequeños tutoriales y codelabs para ir aprendiendo diferentes conceptos.

El módulo está dividido en dos grandes bloques, un primer bloque en dónde aprenderéis sobre desarrollo de aplicaciones Android usando Kotlin como lenguaje de programación y Jetpack Compose para el desarrollo de la UI de las mismas.

El segundo bloque trata sobre el desarrollo de videojuegos, para lo cual usaremos el motor de videojuegos Unity y el lenguaje C#.

Desarrollo de apps Android

Desarrollo de videojuegos con Unity

Videos de clases

Canal de YouTube de clases

- Lista de reproducción de los videos de clases sobre Android y/o Kotlin
- Lista de reproducción de los videos de clases sobre Unity y/o C#

Desarrollo de aplicaciones Android con Kotlin y Jetpack Compose

En esta página encontraréis todo el material que iremos viendo durante el curso de la parte de Android.

Página de descarga de Android Studio

Kotlin Playground

Repositorio con respuestas a algunas dudas habituales

Apartados del curso

1. Introducción Kotlin
2. Jetpack Compose
   i. Composable functions
   ii. State management
   iii. Listas y cuadrículas
   iv. Navegación y rutas
   v. Material Design
3. Ciclo de vida de una app
4. Arquitecturas en Android
5. La capa de UI
6. La capa de datos
7. Conexión a internet
8. Persistencia de datos
9. Enunciados de los ejercicios
10. Documentación extra y recursos externos

Codelabs primordiales

1. Mi primera app para Android - Codelabs introductorios
2. Avanzando con Kotlin y el manejo de la UI - Codelabs
3. Más Kotlin y listas de elementos (LazyColumn) - Codelabs
4. Navegación y arquitectura de la app (MVVM) - Codelabs
5. Cómo conectarse a internet - Codelabs
6. Persistencia de datos - Codelabs

El lenguaje Kotlin

Kotlin es un lenguaje de programación moderno, conciso y seguro que se ejecuta en la máquina virtual de Java (JVM) y también se puede compilar a JavaScript o nativo. Fue desarrollado por JetBrains y Google en 2011 y se ha convertido en el lenguaje de programación oficial para el desarrollo de aplicaciones Android.

Características de Kotlin

•**Interoperabilidad con Java**: Kotlin es 100% interoperable con Java, lo que significa que puedes usar todas las bibliotecas de Java en tus proyectos de Kotlin y viceversa.

•**Seguridad nula**: Kotlin tiene un sistema de tipos que elimina la posibilidad de errores de puntero nulo en tiempo de ejecución.

•**Concisión y legibilidad**: Kotlin es un lenguaje conciso y fácil de leer. Puedes escribir menos código y hacer más cosas.

•**Programación funcional**: Kotlin admite programación funcional y orientada a objetos. Puedes escribir funciones de orden superior, funciones lambda y mucho más.

•**Extensiones de funciones**: Kotlin te permite agregar nuevas funciones a las clases existentes sin heredar de ellas.

•**Clases de datos**: Kotlin tiene una sintaxis especial para crear clases de datos que contienen solo datos y no tienen comportamiento.

•**Corrutinas**: Kotlin tiene soporte para corutinas, que te permiten escribir código asincrónico de manera secuencial.

•**Jetpack Compose**: Kotlin es el lenguaje oficial para el desarrollo de aplicaciones Android con Jetpack Compose, un marco de trabajo moderno para la creación de interfaces de usuario.

Apartados

•Sets en Kotlin

Recursos

•Kotlin Playground: Un entorno de programación en línea para probar y aprender Kotlin.
•Apuntes Kotlin: Un repositorio con apuntes y ejemplos de Kotlin.
•Codelabs introductorios de Android: Codelabs introductorios de Android con Jetpack Compose.

Tipos de variables y datos en Kotlin

En Kotlin las variables pueden declararse de dos formas, de forma explícita o de forma implícita. En el caso de las variables explícitas, se debe indicar el tipo de dato que almacenará la variable, mientras que en las variables implícitas, el tipo de dato se infiere automáticamente por el compilador.

SOBRE LA INFERENCIA DE TIPOS

Que el tipo de dato se infiera automáticamente no quiere decir que Kotlin sea un lenguaje de tipado dinámico, ya que una vez que se asigna un tipo de dato a una variable, no se puede cambiar.

Variables explícitas

Para declarar una variable de forma explícita en Kotlin, se debe indicar el tipo de dato que almacenará la variable seguido del nombre de la variable y opcionalmente de su valor inicial.

```
val nombre: String = "Ejemplo"
val edad: Int = 25
```

En el ejemplo anterior, se declaran dos variables de forma explícita, una de tipo `String` llamada `nombre` y otra de tipo `Int` llamada `edad`.

Variables implícitas

Para declarar una variable de forma implícita en Kotlin, se debe utilizar la palabra clave `val` o `var` seguida del nombre de la variable y opcionalmente de su valor inicial. En este caso, el tipo de dato se infiere automáticamente por el compilador.

```
val nombre = "Ejemplo"
val edad = 25
```

En el ejemplo anterior, se declaran dos variables de forma implícita, una de tipo `String` llamada `nombre` y otra de tipo `Int` llamada `edad`.

Tipos de datos primitivos

En Kotlin, los tipos de datos primitivos son los mismos que en Java, pero con algunas diferencias en la forma en que se declaran.

- **Byte**: Almacena números enteros de 8 bits.
- **Short**: Almacena números enteros de 16 bits.
- **Int**: Almacena números enteros de 32 bits.
- **Long**: Almacena números enteros de 64 bits.
- **Float**: Almacena números de punto flotante de 32 bits.
- **Double**: Almacena números de punto flotante de 64 bits.
- **Char**: Almacena caracteres Unicode de 16 bits.
- **Boolean**: Almacena valores booleanos (`true` o `false`).

```
val entero: Int = 10
val flotante: Float = 10.5f
```

```
val caracter: Char = 'A'
val booleano: Boolean = true
```

Tipos de datos compuestos

Además de los tipos de datos primitivos, Kotlin también tiene tipos de datos compuestos que permiten almacenar colecciones de datos.

- **Array**: Almacena una colección de elementos del mismo tipo.
- **List**: Almacena una colección de elementos ordenados.
- **Set**: Almacena una colección de elementos únicos.
- **Map**: Almacena una colección de pares clave-valor.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val nombres = listOf("Juan", "María", "Pedro")
val colores = setOf("Rojo", "Verde", "Azul")
val edades = mapOf("Juan" to 25, "María" to 30, "Pedro" to 35)
```

Conversión de tipos

En Kotlin, la conversión de tipos se realiza de forma segura y explícita utilizando funciones específicas para cada tipo de dato.

```
val numero: Int = 10
val texto: String = numero.toString()
val texto: String = "10"
val numero: Int = texto.toInt()
```

En el primer ejemplo, se convierte un número entero a una cadena de texto utilizando la función `toString()`. En el segundo ejemplo, se convierte una cadena de texto a un número entero utilizando la función `toInt()`.

Sobre las variables mutables e inmutables

En Kotlin, las variables se pueden declarar como `val` (inmutables) o `var` (mutables). Las variables inmutables no pueden cambiar su valor una vez asignado, mientras que las variables mutables pueden cambiar su valor en cualquier momento.

```
val nombre: String = "Ejemplo" // Variable inmutable
var edad: Int = 25 // Variable mutable
```

En el ejemplo anterior, la variable `nombre` es inmutable, por lo que su valor no puede cambiar una vez asignado. La variable `edad`, en cambio, es mutable, por lo que su valor puede cambiar en cualquier momento.

Sobre la mutabilidad y la inmutabilidad

La inmutabilidad es una característica importante en Kotlin, ya que ayuda a prevenir los errores de programación al evitar que los valores de las variables cambien de forma inesperada. Al utilizar variables inmutables, se puede escribir código más seguro y predecible, lo que facilita la depuración y el mantenimiento del código.

Cuándo trabajamos con listas o arrays, podemos modificar los elementos de la lista, pero no podemos cambiar la referencia de la lista.

```
val lista = mutableListOf(1, 2, 3, 4, 5)
lista[0] = 10 // Modifica el elemento en la posición 0
lista = mutableListOf(6, 7, 8, 9, 10) // Error de compilación
```

En el ejemplo anterior, se modifica el elemento en la posición `0` de la lista `lista`, pero no se puede cambiar la referencia de la lista. Si se intenta asignar una nueva lista a la variable `lista`, se produce un error de compilación.

Podemos ver también un ejemplo de lista mutable pero cuyo contenido no puede ser modificado.

```
val lista = listOf(1, 2, 3, 4, 5)
lista[0] = 10 // Error de compilación
```

En el ejemplo anterior, se intenta modificar el elemento en la posición `0` de la lista `lista`, pero como la lista es inmutable, se produce un error de compilación.

Esto nos deja claro que la inmutabilidad no solo se refiere a la variable en sí, sino también a los elementos que contiene la variable.

Podemos entonces tener cuatro situaciones posibles:

- Variable inmutable y elementos inmutables: No se puede cambiar ni la variable ni los elementos.
- Variable inmutable y elementos mutables: No se puede cambiar la variable, pero sí los elementos.
- Variable mutable y elementos inmutables: Se puede cambiar la variable, pero no los elementos.
- Variable mutable y elementos mutables: Se puede cambiar tanto la variable como los elementos.

LAS VARIABLES INMUTABLES Y LA PROGRAMACIÓN FUNCIONAL

Las variables inmutables son una característica fundamental de la programación funcional, ya que permiten escribir código más seguro y predecible al evitar los efectos secundarios y las mutaciones de estado.

Al utilizar variables inmutables, se puede escribir código más conciso, legible y mantenible, lo que facilita la depuración y el mantenimiento del código.

Sobre las variables nulas

En Kotlin, las variables pueden ser nulas si se declara con el operador `?`. Esto permite que una variable pueda contener un valor nulo en lugar de un valor no nulo.

```
val nombre: String? = null
```

En el ejemplo anterior, la variable `nombre` se declara como nula utilizando el operador `?`. Esto significa que la variable `nombre` puede contener un valor nulo en lugar de un valor no nulo.

SOBRE LA SEGURIDAD DE NULOS EN KOTLIN

El manejo de nulos en Kotlin es una de las características más importantes del lenguaje, ya que ayuda a prevenir los errores de referencia nula que son comunes en otros lenguajes de programación.

Más adelante veremos cómo manejar los valores nulos de forma segura en Kotlin.

Sobre las variables declaradas como const

En Kotlin, las variables se pueden declarar como `const` para indicar que su valor es constante en tiempo de compilación. Las variables `const` deben ser de tipo `val` y deben estar en el ámbito de un objeto o de un compañero de clase.

```
const val PI = 3.14159
```

En el ejemplo anterior, se declara una constante `PI` con un valor de `3.14159`. Esta constante es accesible en tiempo de compilación y su valor no puede cambiar en tiempo de ejecución.

Sobre las Strings en Kotlin

En Kotlin, las cadenas de texto se pueden declarar utilizando comillas simples (`'`) o comillas dobles (`"`). Las cadenas de texto declaradas con comillas simples son de tipo `Char`, mientras que las declaradas con comillas dobles son de tipo `String`.

```
val caracter: Char = 'A'
val texto: String = "Ejemplo"
```

En el ejemplo anterior, se declara una variable `caracter` de tipo `Char` con el valor `'A'` y una variable `texto` de tipo `String` con el valor `"Ejemplo"`.

Las Strings en Kotlin se pueden comparar utilizando el operador `==` para comparar el contenido de las cadenas y el operador `===` para comparar las referencias de las cadenas.

```
val texto1 = "Hola"
val texto2 = "Hola"
println(texto1 == texto2) // true
println(texto1 === texto2) // true
```

En el ejemplo anterior, se comparan dos cadenas de texto `texto1` y `texto2` utilizando los operadores `==` y `===`. Ambas comparaciones devuelven `true` ya que las cadenas son iguales en contenido y referencia.

A diferencia de Java, en Kotlin las cadenas de texto son inmutables, lo que significa que una vez que se crea una cadena de texto, no se puede modificar su contenido. Para modificar una cadena de texto en Kotlin, se debe crear una nueva cadena con el contenido modificado.

```
val texto = "Hola"
val nuevoTexto = texto + " Mundo"
```

En el ejemplo anterior, se crea una nueva cadena de texto `nuevoTexto` concatenando la cadena `texto` con la cadena `" Mundo"`. La cadena `texto` no se modifica, sino que se crea una nueva cadena con el contenido modificado.

Sin embargo, si comparamos una string con un caracter, Kotlin no permite la comparación directa, ya que son tipos de datos diferentes.

```
val texto = "H"
val caracter = 'H'
println(texto == caracter) // Error de compilación
```

En el ejemplo anterior, se intenta comparar una cadena de texto `texto` con un caracter `caracter`, lo cual produce un error de compilación ya que los tipos de datos son diferentes.

Expresiones vs sentencias

En Kotlin, las expresiones y las sentencias son dos conceptos fundamentales que se utilizan para definir el flujo de control y la lógica de un programa. Aunque ambos se utilizan para realizar operaciones y tomar decisiones en un programa, existen diferencias importantes entre ellos que es importante comprender.

Expresiones

En Kotlin, una expresión es una combinación de valores, variables, operadores y funciones que se evalúa para producir un resultado. Las expresiones pueden ser tan simples como una constante o tan complejas como una llamada a una función que devuelve un valor.

Las expresiones en Kotlin pueden tener un valor de retorno, lo que significa que pueden ser utilizadas en cualquier lugar donde se espere un valor, como en la inicialización de una variable, en una sentencia de control de flujo o en una llamada a una función.

```
val numero = 10
val resultado = numero * 2
```

En el ejemplo anterior, la expresión `numero * 2` se evalúa para producir un resultado que se asigna a la variable `resultado`. La expresión `numero * 2` es una expresión aritmética que multiplica el valor de la variable `numero` por `2` y devuelve el resultado.

Sentencias

En Kotlin, una sentencia es una instrucción que realiza una acción en un programa. Las sentencias pueden ser tan simples como una asignación de variable o tan complejas como una estructura de control de flujo que toma decisiones basadas en condiciones.

Las sentencias en Kotlin no tienen un valor de retorno, lo que significa que no pueden ser utilizadas en lugares donde se espere un valor, como en la inicialización de una variable o en una llamada a una función.

```
val numero = 10
if (numero > 0) {
    println("El número es positivo")
} else {
    println("El número es negativo")
}
```

En el ejemplo anterior, la sentencia `if (numero > 0) { ... } else { ... }` es una estructura de control de flujo que toma una decisión basada en la condición `numero > 0`. La sentencia `if` evalúa la condición y ejecuta el bloque de código correspondiente si la condición es verdadera.

Diferencias entre expresiones y sentencias

Las principales diferencias entre expresiones y sentencias en Kotlin son las siguientes:

•**Valor de retorno**: Las expresiones tienen un valor de retorno y pueden ser utilizadas en lugares donde se espere un valor, mientras que las sentencias no tienen un valor de retorno y no pueden ser utilizadas en lugares donde se espere un valor.

•**Complejidad**: Las expresiones pueden ser tan simples como una constante o tan complejas como una llamada a una función que devuelve un valor, mientras que las sentencias realizan acciones en un programa y pueden ser tan simples como una asignación de variable o tan complejas como una estructura de control de flujo.

•**Uso**: Las expresiones se utilizan para realizar operaciones y producir resultados, mientras que las sentencias se utilizan para realizar acciones en un programa, como asignar valores a variables, tomar decisiones basadas en condiciones o repetir bloques de código.

LA VENTAJA DE LAS EXPRESIONES EN LA PROGRAMACIÓN FUNCIONAL

En la programación funcional, se fomenta el uso de expresiones en lugar de sentencias, ya que las expresiones son más concisas, legibles y fáciles de entender.

Al utilizar expresiones en lugar de sentencias, se puede escribir menos código y hacer más cosas.

Esto es especialmente útil en Kotlin, que es un lenguaje de programación funcional que fomenta el uso de expresiones para realizar operaciones y producir resultados de forma concisa y eficiente.

Ejemplo de una función que usa sentencias (statement) y expresiones (expression)

Pongamos el ejemplo de una función que dictamina si una persona es mayor de edad o no.

Versión con sentencias

```
fun esMayorDeEdad(edad: Int): Boolean {
    if (edad >= 18) {
        return true
    } else {
        return false
    }
}
```

En este caso, la función `esMayorDeEdad` utiliza sentencias para tomar una decisión basada en la edad de la persona y devolver un valor booleano que indica si la persona es mayor de edad o no.

Versión con expresiones

```
fun esMayorDeEdad(edad: Int): Boolean = edad >= 18
```

En este caso, la función `esMayorDeEdad` utiliza una expresión para evaluar si la edad de la persona es mayor o igual a `18` y devolver un valor booleano que indica si la persona es mayor de edad o no.

Cómo se puede observar, la versión con expresiones es más concisa y legible que la versión con sentencias, ya que utiliza una expresión para realizar la misma operación de forma más eficiente y clara.

Funciones y lambdas en Kotlin

En Kotlin, las funciones son ciudadanos de primera clase, lo que significa que puedes tratarlas como cualquier otro tipo de dato, como un `Int` o un `String`. Esto te permite pasar funciones como argumentos a otras funciones, devolver funciones de otras funciones y almacenar funciones en variables.

Las funciones en Kotlin se definen utilizando la palabra clave `fun`, seguida del nombre de la función, los parámetros de entrada y el tipo de retorno.

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

En el ejemplo anterior, se define una función `suma` que toma dos parámetros de tipo `Int` y devuelve un valor de tipo `Int`. La función suma los dos parámetros de entrada y devuelve el resultado.

Funciones de orden superior

En Kotlin, puedes pasar funciones como argumentos a otras funciones. Estas funciones se conocen como funciones de orden superior y te permiten escribir código más conciso y reutilizable.

```
fun operacion(a: Int, b: Int, funcion: (Int, Int) -> Int): Int {
    return funcion(a, b)
}
fun suma(a: Int, b: Int): Int {
    return a + b
}
fun resta(a: Int, b: Int): Int {
    return a - b
}
val resultadoSuma = operacion(10, 5, ::suma)
val resultadoResta = operacion(10, 5, ::resta)
```

En el ejemplo anterior, se define una función `operacion` que toma dos parámetros de tipo `Int` y una función de orden superior que toma dos parámetros de tipo `Int` y devuelve un valor de tipo `Int`. La función `operacion` aplica la función de orden superior a los dos parámetros de entrada y devuelve el resultado.

Funciones lambda

En Kotlin, puedes definir funciones anónimas conocidas como funciones lambda. Las funciones lambda son funciones sin nombre que puedes pasar como argumentos a otras funciones.

```
val suma = { a: Int, b: Int -> a + b }
val resta = { a: Int, b: Int -> a - b }
val resultadoSuma = suma(10, 5)
val resultadoResta = resta(10, 5)
```

En el ejemplo anterior, se definen dos funciones lambda `suma` y `resta` que toman dos parámetros de tipo `Int` y devuelven un valor de tipo `Int`. Las funciones lambda se asignan a variables y se pueden utilizar como cualquier otra función.

Los parámetros en Kotlin

A diferencia de Java, en Kotlin los parámetros de una función son inmutables por defecto, lo que significa que no se pueden modificar dentro de la función. Si necesitas modificar un parámetro dentro de una función, debes declararlo como una variable `var`.

```
fun duplicar(numero: Int): Int {
    var resultado = numero
    resultado *= 2
    return resultado
}
```

En el ejemplo anterior, se define una función `duplicar` que toma un parámetro de tipo `Int` y devuelve un valor de tipo `Int`. El parámetro `numero` se declara como una variable `var` para poder modificar su valor dentro de la función.

Parámetros con valores por defecto

Los parámetros de una función en Kotlin pueden tener valores por defecto, lo que te permite llamar a la función sin proporcionar todos los argumentos.

```
fun saludar(nombre: String = "Mundo") {
    println("Hola, $nombre!")
}
saludar() // Hola, Mundo!
saludar("Juan") // Hola, Juan!
```

En el ejemplo anterior, se define una función `saludar` que toma un parámetro de tipo `String` con un valor por defecto de `"Mundo"`. Si no se proporciona un argumento al llamar a la función, se utiliza el valor por defecto.

FUNCIONES CON VALORES POR DEFECTO Y SU NO OBLIGATORIEDAD, SU USO EN COMPOSE

Las funciones con valores por defecto son muy útiles en Jetpack Compose, ya que te permiten definir componentes con valores por defecto y llamar a esos componentes sin proporcionar todos los argumentos.

Por ejemplo, puedes definir un botón con un texto por defecto y un color por defecto, y luego llamar a ese botón sin proporcionar el texto o el color si deseas utilizar los valores por defecto.

```
@Composable
fun Boton(texto: String = "Aceptar", color: Color = Color.Blue) {
    Button(onClick = { /* Acción al hacer clic */ }) {
        Text(texto, color = color)
    }
}
Boton() // Botón con texto "Aceptar" y color azul
Boton("Cancelar", Color.Red) // Botón con texto "Cancelar" y color rojo
```

Parámetros de una función lambda

En una función lambda en Kotlin, puedes especificar los tipos de los parámetros de entrada o dejar que el compilador infiera los tipos automáticamente.

```
val suma: (Int, Int) -> Int = { a, b -> a + b }
val resta = { a: Int, b: Int -> a - b }
```

En el ejemplo anterior, se definen dos funciones lambda `suma` y `resta` que toman dos parámetros de tipo `Int` y devuelven un valor de tipo `Int`. En la función lambda `suma`, se especifican los tipos de los parámetros de entrada, mientras que en la función lambda `resta`, se deja que el compilador infiera los tipos automáticamente.

La palabra reservada `it`

En una función lambda en Kotlin, puedes utilizar la palabra reservada `it` para referirte al único parámetro de entrada si la función lambda tiene un solo parámetro.

```
val cuadrado: (Int) -> Int = { it * it }
```

En el ejemplo anterior, se define una función lambda `cuadrado` que toma un parámetro de tipo `Int` y devuelve un valor de tipo `Int`.

SHORT EXPLICATIVO EN YOUTUBE

Enlace al vídeo
La palabra reservada `it` se utiliza para referirse al único parámetro de entrada de la función lambda.

Esto es útil cuando la función lambda tiene un solo parámetro y quieres hacer el código más conciso.

Funciones lambda con múltiples líneas

En una función lambda en Kotlin, puedes utilizar múltiples líneas de código si es necesario.

```
val suma: (Int, Int) -> Int = { a, b ->
    val resultado = a + b
    println("La suma de $a y $b es $resultado")
    resultado
}
```

En el ejemplo anterior, se define una función lambda `suma` que toma dos parámetros de tipo `Int` y devuelve un valor de tipo `Int`. La función lambda realiza la suma de los dos parámetros y muestra un mensaje por consola con el resultado.

Número variable de argumentos

En Kotlin, puedes definir funciones que toman un número variable de argumentos utilizando el operador `vararg`.

```
fun sumar(vararg numeros: Int): Int {
    var suma = 0
```

```
    for (numero in numeros) {
        suma += numero
    }
    return suma
}
val resultado = sumar(1, 2, 3, 4, 5)
```

En el ejemplo anterior, se define una función `sumar` que toma un número variable de argumentos de tipo `Int` utilizando el operador `vararg`. La función suma todos los números pasados como argumentos y devuelve el resultado.

Funciones de extensión (Extension Functions)

En Kotlin, puedes agregar nuevas funciones a las clases existentes sin heredar de ellas.

Estas funciones se conocen como funciones de extensión y te permiten extender la funcionalidad de las clases sin modificar su código fuente.

```
fun String.invertir(): String {
    return this.reversed()
}
val texto = "Hola, mundo!"
val textoInvertido = texto.invertir()
```

En el ejemplo anterior, se define una función de extensión `invertir` para la clase `String` que invierte el contenido de la cadena de texto. La función de extensión se llama como si fuera un método de la clase `String`.

FUNCIONES DE EXTENSIÓN Y FUNCIONES DE ORDEN SUPERIOR

Las funciones de extensión y las funciones de orden superior son dos características poderosas de Kotlin que te permiten escribir código más conciso y reutilizable.

Las funciones de extensión te permiten agregar nuevas funciones a las clases existentes sin heredar de ellas, mientras que las funciones de orden superior te permiten pasar funciones como argumentos a otras funciones.

Al combinar estas dos características, puedes escribir código más flexible y expresivo en Kotlin.

Null safety en Kotlin

En Kotlin, el manejo de nulos es una parte fundamental del lenguaje.

Kotlin está diseñado para evitar los errores de referencia nula que son comunes en otros lenguajes de programación, como Java.

En Kotlin, los tipos de datos pueden ser nulos o no nulos, lo que te permite expresar de forma segura si un valor puede ser nulo o no.

DIFERENCIA CON JAVA

En Java, todos los tipos de datos pueden ser nulos, lo que puede llevar a errores de referencia nula si no se manejan correctamente.

En Kotlin, los tipos de datos no nulos deben ser manejados de forma explícita, lo que ayuda a prevenir los errores de referencia nula.

Tipos de datos nulos

En Kotlin, los tipos de datos pueden ser nulos o no nulos.

- **Tipos de datos no nulos**: Los tipos de datos no nulos no pueden contener valores nulos y deben ser inicializados con un valor no nulo.
- **Tipos de datos nulos**: Los tipos de datos nulos pueden contener valores nulos y deben ser inicializados con un valor nulo o con la función `null`.

```
val nombre: String = "Juan" // Tipo de dato no nulo
val apellido: String? = null // Tipo de dato nulo
```

En el ejemplo anterior, la variable `nombre` es de tipo `String` no nulo, por lo que no puede contener un valor nulo. La variable `apellido`, en cambio, es de tipo `String?` nulo, por lo que puede contener un valor nulo.

Operadores de seguridad de nulos

En Kotlin, puedes utilizar operadores de seguridad de nulos para manejar los valores nulos de forma segura.

- **Operador de llamada segura (`?.`)**: El operador de llamada segura `?.` te permite acceder a las propiedades de un objeto nulo sin lanzar una excepción de referencia nula.
- **Operador de elvis (`?:`)**: El operador de elvis `?:` te permite proporcionar un valor predeterminado en caso de que una expresión sea nula.
- **Operador de no nulo (`!!`)**: El operador de no nulo `!!` te permite forzar la ejecución de una expresión nula, lo que puede lanzar una excepción de referencia nula si la expresión es nula.

```
val nombre: String? = null
val longitud = nombre?.length // null
val longitud = nombre?.length ?: 0 // 0
val longitud = nombre!!.length // Lanza una excepción de referencia nula
```

En el ejemplo anterior, la variable `nombre` es de tipo `String?` nulo, por lo que puede contener un valor nulo.

- En la primera línea, se utiliza el operador de llamada segura `?.` para acceder a la propiedad `length` de la variable `nombre`. Como `nombre` es nulo, la expresión `nombre?.length` devuelve `null`.
- En la segunda línea, se utiliza el operador de elvis `?:` para proporcionar un valor predeterminado de `0` en caso de que la expresión `nombre?.length` sea nula. Como `nombre` es nulo, la expresión `nombre?.length ?: 0` devuelve `0`.
- En la tercera línea, se utiliza el operador de no nulo `!!` para forzar la ejecución de la expresión `nombre!!.length`. Como `nombre` es nulo, la expresión `nombre!!.length` lanza una excepción de referencia nula.

Funciones de extensión de seguridad de nulos

En Kotlin, puedes utilizar funciones de extensión para agregar funcionalidades a los tipos de datos nulos.

```
fun String?.oLongitud(): Int {
    return this?.length ?: 0
}
val nombre: String? = null
val longitud = nombre.oLongitud() // 0
```

En el ejemplo anterior, se define una función de extensión `oLongitud` para el tipo de dato `String?`. La función `oLongitud` devuelve la longitud de la cadena si no es nula, o `0` si es nula.

Conclusiones

El manejo de nulos en Kotlin es una parte fundamental del lenguaje que te permite expresar de forma segura si un valor puede ser nulo o no.

Al utilizar tipos de datos nulos y operadores de seguridad de nulos, puedes prevenir los errores de referencia nula y escribir código más robusto y seguro.

RECURSOS

- Documentación oficial de Kotlin sobre seguridad de nulos
- Kotlin Playground: Un entorno de programación en línea para probar y aprender Kotlin.
- Apuntes Kotlin: Un repositorio con apuntes y ejemplos de Kotlin.

Programación orientada a objetos en Kotlin

La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos", que pueden contener datos en forma de campos (también conocidos como atributos) y código en forma de procedimientos (también conocidos como métodos).

En Kotlin, puedes crear clases y objetos para modelar entidades del mundo real y encapsular datos y comportamientos relacionados. Kotlin es un lenguaje de programación orientado a objetos y admite todas las características tradicionales de la programación orientada a objetos, como la herencia, el polimorfismo, la encapsulación y la abstracción.

Clases y objetos

En Kotlin, puedes definir una clase utilizando la palabra clave `class` seguida del nombre de la clase y el cuerpo de la clase entre llaves `{}`.

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

En el ejemplo anterior, se define una clase `Persona` con dos propiedades `nombre` y `edad`. Las propiedades de la clase se inicializan con valores predeterminados.

Para crear un objeto de una clase en Kotlin, puedes utilizar la palabra clave `val` o `var` seguida del nombre del objeto, el operador de asignación `=` y la invocación del constructor de la clase.

```
val persona = Persona()
persona.nombre = "Juan"
persona.edad = 30
```

En el ejemplo anterior, se crea un objeto `persona` de la clase `Persona` y se inicializan las propiedades `nombre` y `edad` del objeto.

Propiedades y métodos

En Kotlin, puedes definir propiedades y métodos en una clase utilizando la palabra clave `var` o `val` seguida del nombre de la propiedad o el método y el tipo de dato de la propiedad o el método.

```
class Persona {
    var nombre: String = ""
    var edad: Int = 0
    fun saludar() {
        println("Hola, soy $nombre")
    }
}
```

En el ejemplo anterior, se define una clase `Persona` con dos propiedades `nombre` y `edad`, y un método `saludar` que imprime un mensaje de saludo con el nombre de la persona.

Para acceder a las propiedades y métodos de un objeto en Kotlin, puedes utilizar el operador de acceso `.` seguido del nombre de la propiedad o el método.

```
val persona = Persona()
persona.nombre = "Juan"
persona.edad = 30
persona.saludar()
```

En el ejemplo anterior, se crea un objeto `persona` de la clase `Persona` y se inicializan las propiedades `nombre` y `edad` del objeto. Luego, se llama al método `saludar` en el objeto `persona` para imprimir un mensaje de saludo con el nombre de la persona.

DIFERENCIAS CON JAVA

A diferencia de Java, en Kotlin, las propiedades y los métodos de una clase son públicos por defecto, lo que significa que se pueden acceder desde cualquier parte del código.

Encapsulación

En Kotlin, puedes encapsular propiedades y métodos en una clase utilizando los modificadores de acceso `public`, `protected`, `private` y `internal`.

- `public`: Las propiedades y métodos públicos son accesibles desde cualquier parte del código.
- `protected`: Las propiedades y métodos protegidos son accesibles desde la clase actual y las clases derivadas.
- `private`: Las propiedades y métodos privados son accesibles solo desde la clase actual.
- `internal`: Las propiedades y métodos internos son accesibles desde el módulo actual.

```
class Persona {
    var nombre: String = ""
        private set
    var edad: Int = 0
        private set
    fun saludar() {
        println("Hola, soy $nombre")
    }
}
```

En el ejemplo anterior, se define una clase `Persona` con dos propiedades `nombre` y `edad` que se han encapsulado con el modificador de acceso `private`. Esto significa que las propiedades `nombre` y `edad` solo se pueden acceder y modificar dentro de la clase `Persona`.

Para acceder a las propiedades de una clase en Kotlin, puedes utilizar los métodos de acceso `get` y `set` para obtener y establecer el valor de una propiedad.

```
val persona = Persona()
persona.nombre = "Juan"
persona.edad = 30
persona.saludar()
```

En el ejemplo anterior, se crea un objeto `persona` de la clase `Persona` y se inicializan las propiedades `nombre` y `edad` del objeto.

Sin embargo, al intentar modificar las propiedades `nombre` y `edad` desde fuera de la clase `Persona`, se produce un error de compilación debido a que las propiedades están encapsuladas con el modificador de acceso `private`.

Setters y Getters personalizados

En Kotlin, puedes definir setters y getters personalizados para las propiedades de una clase utilizando la palabra clave `set` y `get` seguida de la lógica personalizada para establecer y obtener el valor de la propiedad.

```
class Persona {
    var nombre: String = ""
        set(value) {
            field = value.capitalize()
        }
    var edad: Int = 0
        set(value) {
            field = if (value >= 0) value else 0
        }
    fun saludar() {
        println("Hola, soy $nombre")
    }
}
```

En el ejemplo anterior, se define una clase `Persona` con dos propiedades `nombre` y `edad` que tienen setters personalizados. El setter de la propiedad `nombre` capitaliza el valor de la propiedad, y el setter de la propiedad `edad` establece el valor de la propiedad en `0` si es menor que `0`.

Para acceder a las propiedades de una clase en Kotlin, puedes utilizar los métodos de acceso `get` y `set` para obtener y establecer el valor de una propiedad.

```
val persona = Persona()
persona.nombre = "juan"
persona.edad = -10
persona.saludar()
```

En el ejemplo anterior, se crea un objeto `persona` de la clase `Persona` y se inicializan las propiedades `nombre` y `edad` del objeto.

Constructores

En Kotlin, puedes definir un constructor primario utilizando la palabra clave `constructor` seguida de los parámetros del constructor.

Sin embargo, en Kotlin, puedes omitir la palabra clave `constructor` y definir los parámetros del constructor directamente en la declaración de la clase.

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

```
    var edad: Int = edad
}
```

En el ejemplo anterior, se define una clase `Persona` con un constructor primario que toma dos parámetros `nombre` y `edad`. Las propiedades de la clase se inicializan con los valores de los parámetros del constructor.

Para crear un objeto de una clase con un constructor primario en Kotlin, puedes utilizar la palabra clave `val` o `var` seguida del nombre del objeto, el operador de asignación `=` y la invocación del constructor de la clase con los argumentos del constructor.

```
val persona = Persona("Juan", 30)
```

En el ejemplo anterior, se crea un objeto `persona` de la clase `Persona` utilizando el constructor primario y se inicializan las propiedades `nombre` y `edad` del objeto.

Constructores secundarios

En Kotlin, puedes definir constructores secundarios utilizando la palabra clave `constructor` seguida de los parámetros del constructor.

```
class Persona {
    var nombre: String = ""
    var edad: Int = 0
    constructor(nombre: String, edad: Int) {
        this.nombre = nombre
        this.edad = edad
    }
}
```

En el ejemplo anterior, se define una clase `Persona` con un constructor secundario que toma dos parámetros `nombre` y `edad`. Las propiedades de la clase se inicializan con los valores de los parámetros del constructor secundario.

Para crear un objeto de una clase con un constructor secundario en Kotlin, puedes utilizar la palabra clave `val` o `var` seguida del nombre del objeto, el operador de asignación `=` y la invocación del constructor secundario de la clase con los argumentos del constructor.

```
val persona = Persona("Juan", 30)
```

En el ejemplo anterior, se crea un objeto `persona` de la clase `Persona` utilizando el constructor secundario y se inicializan las propiedades `nombre` y `edad` del objeto.

Herencia

En Kotlin, puedes crear una clase que herede de otra clase utilizando la palabra clave `:` seguida del nombre de la clase base.

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

```
class Empleado : Persona() {
    var salario: Double = 0.0
}
```

En el ejemplo anterior, se define una clase `Persona` con dos propiedades `nombre` y `edad`, y una clase `Empleado` que hereda de la clase `Persona` y agrega una propiedad `salario`.

Para crear un objeto de una clase derivada en Kotlin, puedes utilizar la palabra clave `val` o `var` seguida del nombre del objeto, el operador de asignación `=` y la invocación del constructor de la clase derivada.

```
val empleado = Empleado()
empleado.nombre = "Juan"
empleado.edad = 30
empleado.salario = 1000.0
```

En el ejemplo anterior, se crea un objeto `empleado` de la clase `Empleado` y se inicializan las propiedades `nombre`, `edad` y `salario` del objeto.

Polimorfismo

En Kotlin, puedes utilizar el polimorfismo para tratar un objeto de una clase derivada como un objeto de la clase base.

```
open class Persona {
    open fun saludar() {
        println("Hola, soy una persona")
    }
}
class Empleado : Persona() {
    override fun saludar() {
        println("Hola, soy un empleado")
    }
}
```

En el ejemplo anterior, se define una clase `Persona` con un método `saludar` y una clase `Empleado` que hereda de la clase `Persona` y sobrescribe el método `saludar`.

LA PALABRA RESERVADA OPEN

La palabra reservada `open` se utiliza para marcar una clase o un miembro de una clase como "sobreseíble", lo que significa que puede ser heredado y sobrescrito por clases derivadas.
Para utilizar el polimorfismo en Kotlin, puedes crear un objeto de la clase derivada y asignarlo a una variable de la clase base.

```
val persona: Persona = Empleado()
persona.saludar()
```

En el ejemplo anterior, se crea un objeto `empleado` de la clase `Empleado` y se asigna a una variable `persona` de la clase `Persona`. Al llamar al método `saludar` en la variable `persona`, se ejecuta la implementación del método `saludar` de la clase `Empleado`.

Abstracción

En Kotlin, puedes utilizar la abstracción para definir una clase base con métodos abstractos que deben ser implementados por las clases derivadas.

```
abstract class Persona {
    abstract fun saludar()
}
class Empleado : Persona() {
    override fun saludar() {
        println("Hola, soy un empleado")
    }
}
```

En el ejemplo anterior, se define una clase `Persona` con un método abstracto `saludar` y una clase `Empleado` que hereda de la clase `Persona` e implementa el método `saludar`.

Para utilizar la abstracción en Kotlin, puedes crear un objeto de la clase derivada y asignarlo a una variable de la clase base.

```
val persona: Persona = Empleado()
persona.saludar()
```

En el ejemplo anterior, se crea un objeto `empleado` de la clase `Empleado` y se asigna a una variable `persona` de la clase `Persona`. Al llamar al método `saludar` en la variable `persona`, se ejecuta la implementación del método `saludar` de la clase `Empleado`.

Objetos anónimos en Kotlin

En Kotlin, puedes crear objetos anónimos utilizando la palabra clave `object`.

Los objetos anónimos son instancias de una clase anónima que no tienen un nombre y se utilizan para definir una clase de forma concisa y reutilizable.

```
val persona = object {
    val nombre = "Juan"
    val edad = 25
}
println(persona.nombre) // Juan
println(persona.edad) // 25
```

En el ejemplo anterior, se crea un objeto anónimo que tiene dos propiedades `nombre` y `edad`.

El objeto anónimo se asigna a la variable `persona` y se puede acceder a sus propiedades utilizando la notación de punto.

Los objetos anónimos son útiles cuando necesitas crear una instancia de una clase de forma rápida y concisa sin tener que definir una clase con nombre.

Uso de objetos anónimos

Los objetos anónimos se utilizan en Kotlin para:

- Crear instancias de una clase de forma rápida y concisa.
- Definir clases de forma reutilizable sin tener que definir una clase con nombre.
- Implementar interfaces y clases abstractas de forma anónima.

Los objetos anónimos son una característica poderosa de Kotlin que te permite escribir código de forma más concisa y eficiente.

Implementación de interfaces con objetos anónimos

En Kotlin, puedes implementar interfaces de forma anónima utilizando objetos anónimos. Esto te permite definir una clase que implementa una interfaz sin tener que definir una clase con nombre.

```
interface Saludable {
    fun saludar()
}
val persona = object : Saludable {
    override fun saludar() {
        println("Hola, soy una persona saludable!")
    }
}
persona.saludar() // Hola, soy una persona saludable!
```

En el ejemplo anterior, se define una interfaz `Saludable` con un método `saludar`.

Se crea un objeto anónimo que implementa la interfaz `Saludable` y se asigna a la variable `persona`.

El objeto anónimo define la implementación del método `saludar` y se puede llamar al método utilizando la notación de punto.

Implementación del patrón Singleton con objetos anónimos

En Kotlin, puedes implementar el patrón Singleton utilizando objetos anónimos. El patrón Singleton garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia.

```
object Configuracion {
    val servidor = "localhost"
    val puerto = 8080
}
println(Configuracion.servidor) // localhost
println(Configuracion.puerto) // 8080
```

En el ejemplo anterior, se define un objeto anónimo `Configuracion` que tiene dos propiedades `servidor` y `puerto`.

El objeto anónimo se utiliza para almacenar la configuración de la aplicación y garantiza que solo haya una instancia de la configuración en toda la aplicación.

CARACTERÍSTICAS DE LOS OBJETOS ANÓNIMOS

Los objetos anónimos en Kotlin tienen las siguientes características:

- No tienen un nombre y se crean utilizando la palabra clave `object`.
- Pueden tener propiedades, métodos y constructores.
- Se utilizan para definir clases de forma concisa y reutilizable.
- Se pueden utilizar para implementar interfaces y clases abstractas de forma anónima.
- Se pueden utilizar para implementar el patrón Singleton y almacenar configuraciones globales.

SOBRE EL PATRÓN SINGLETON

El patrón Singleton es un patrón de diseño que garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia.

En Kotlin, puedes implementar el patrón Singleton utilizando objetos anónimos, que garantizan que solo haya una instancia de la clase en toda la aplicación.

Usos comunes del patrón Singleton incluyen la creación de objetos de configuración, conexiones a bases de datos y puntos de acceso a servicios globales.

Data classes en Kotlin

En Kotlin, puedes crear clases de datos utilizando la palabra clave `data`. Las clases de datos son una forma concisa de definir clases que contienen datos y no tienen lógica adicional.

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

En el ejemplo anterior, se define una clase de datos `Persona` que tiene dos propiedades `nombre` y `edad`. La clase de datos se crea utilizando la palabra clave `data` seguida del nombre de la clase y las propiedades de la clase.

SHORT EXPLICATIVO EN YOUTUBE

Enlace al vídeo
VÍDEO EXPLICATIVO EN YOUTUBE

Tutorial LibreOffice Writer - 24/40 Insertar hiperenlaces.

Propiedades de las clases de datos

Las clases de datos en Kotlin tienen las siguientes propiedades:

- **Propiedades de solo lectura**: Las propiedades de una clase de datos son de solo lectura y no se pueden modificar una vez que se han inicializado.
- **Métodos `toString()`, `equals()` y `hashCode()`**: Las clases de datos en Kotlin generan automáticamente los métodos `toString()`, `equals()` y `hashCode()` a partir de las propiedades de la clase.
- **Métodos `componentN()`**: Las clases de datos en Kotlin generan automáticamente métodos `componentN()` que te permiten desestructurar objetos de la clase en variables individuales.
- **Método `copy()`**: Las clases de datos en Kotlin generan automáticamente un método `copy()` que te permite crear copias de objetos de la clase con propiedades modificadas.

Creación de objetos de clases de datos

Para crear un objeto de una clase de datos en Kotlin, puedes utilizar la palabra clave `data` seguida del nombre de la clase y los valores de las propiedades.

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

En el ejemplo anterior, se crea un objeto de la clase de datos `Persona` con el nombre `Juan` y la edad `25`.

Desestructuración de objetos de clases de datos

En Kotlin, puedes desestructurar objetos de clases de datos en variables individuales utilizando la notación de desestructuración.

```
val (nombre, edad) = persona
```

En el ejemplo anterior, se desestructura el objeto `persona` en las variables `nombre` y `edad`.

Copia de objetos de clases de datos

En Kotlin, puedes crear copias de objetos de clases de datos utilizando el método `copy()`.

```
val persona2 = persona.copy(edad = 30)
```

En el ejemplo anterior, se crea una copia del objeto `persona` con la edad modificada a `30`.

Enum classes en Kotlin

En Kotlin, puedes crear enumeraciones utilizando la palabra clave `enum class`.

Las enumeraciones son una forma de definir un conjunto fijo de constantes que representan valores específicos. Cada constante en una enumeración tiene un nombre y un valor asociado.

```
enum class Color {
    ROJO, VERDE, AZUL
}
```

En el ejemplo anterior, se define una enumeración `Color` con tres constantes `ROJO`, `VERDE` y `AZUL`.

Propiedades de las enumeraciones

Las enumeraciones en Kotlin tienen las siguientes propiedades:

- **Constantes**: Las enumeraciones contienen un conjunto fijo de constantes que representan valores específicos.
- **Propiedades y métodos**: Las enumeraciones pueden tener propiedades y métodos asociados con cada constante.
- **Constructores**: Las enumeraciones pueden tener constructores personalizados para inicializar las constantes con valores específicos.
- **Implementación de interfaces**: Las enumeraciones pueden implementar interfaces y heredar de clases.

Acceso a las constantes de una enumeración

Puedes acceder a las constantes de una enumeración utilizando la notación de punto.

```
val color = Color.ROJO
```

En el ejemplo anterior, se accede a la constante `ROJO` de la enumeración `Color` y se asigna a la variable `color`.

Propiedades y métodos de una enumeración

Puedes definir propiedades y métodos en una enumeración para agregar funcionalidades a las constantes.

```
enum class Color(val rgb: Int) {
    ROJO(0xFF0000),
    VERDE(0x00FF00),
    AZUL(0x0000FF);
    fun nombreEnMayusculas(): String {
        return name.toUpperCase()
    }
}
```

En el ejemplo anterior, se define una enumeración `Color` con una propiedad `rgb` y un método `nombreEnMayusculas` que devuelve el nombre de la constante en mayúsculas.

Implementación de interfaces con enumeraciones

Las enumeraciones en Kotlin pueden implementar interfaces para agregar funcionalidades a las constantes.

```
interface Describible {
    fun describir(): String
}
enum class DiaSemana : Describible {
    LUNES {
        override fun describir(): String {
            return "Primer día de la semana"
        }
    },
    MARTES {
        override fun describir(): String {
            return "Segundo día de la semana"
        }
    },
    MIERCOLES {
        override fun describir(): String {
            return "Tercer día de la semana"
        }
    }
}
```

En el ejemplo anterior, se define una interfaz `Describible` con un método `describir`.

Se crea una enumeración `DiaSemana` que implementa la interfaz `Describible` y define la implementación del método `describir` para cada constante.

Uso de enumeraciones en Kotlin

Las enumeraciones son útiles en Kotlin para:

- Definir un conjunto fijo de constantes que representan valores específicos.
- Agregar propiedades y métodos a las constantes para agregar funcionalidades.
- Implementar interfaces y heredar de clases para agregar funcionalidades a las constantes.

Genéricos en Kotlin

En Kotlin, puedes utilizar genéricos para crear clases, funciones e interfaces que trabajen con tipos de datos de forma genérica. Los genéricos te permiten escribir código que es reutilizable y flexible, ya que puedes definir clases y funciones que trabajen con cualquier tipo de datos.

VIDEO EXPLICATIVO EN YOUTUBE

Genéricos en Kotlin

Clases genéricas

En Kotlin, puedes crear clases genéricas utilizando la palabra clave `class` seguida de los parámetros de tipo entre corchetes `<>`. Los parámetros de tipo se utilizan para definir los tipos de datos que la clase puede manejar.

```
class Caja<T>(val contenido: T) {
    fun obtenerContenido(): T {
        return contenido
    }
}
val cajaEntero = Caja(10)
val cajaCadena = Caja("Hola")
val entero: Int = cajaEntero.obtenerContenido()
val cadena: String = cajaCadena.obtenerContenido()
```

En el ejemplo anterior, se define una clase genérica `Caja` que tiene un parámetro de tipo `T`. La clase `Caja` tiene una propiedad `contenido` de tipo `T` y un método `obtenerContenido` que devuelve el contenido de la caja.

Se crean dos instancias de la clase `Caja` con tipos de datos diferentes: una con un entero y otra con una cadena. Se obtiene el contenido de cada caja y se asigna a variables de tipo `Int` y `String`.

Funciones genéricas

En Kotlin, puedes crear funciones genéricas utilizando la palabra clave `fun` seguida de los parámetros de tipo entre corchetes `<>`. Los parámetros de tipo se utilizan para definir los tipos de datos que la función puede manejar.

```
fun <T> imprimirElemento(elemento: T) {
    println(elemento)
}
imprimirElemento(10)
imprimirElemento("Hola")
```

En el ejemplo anterior, se define una función genérica `imprimirElemento` que toma un parámetro de tipo `T` y lo imprime en la consola. La función `imprimirElemento` se llama dos veces con un entero y una cadena como argumentos.

Interfaces genéricas

En Kotlin, puedes crear interfaces genéricas utilizando la palabra clave `interface` seguida de los parámetros de tipo entre corchetes `<>`. Los parámetros de tipo se utilizan para definir los tipos de datos que la interfaz puede manejar.

```
interface Contenedor<T> {
    fun obtenerContenido(): T
}
class Caja<T>(val contenido: T) : Contenedor<T> {
    override fun obtenerContenido(): T {
        return contenido
    }
}
val cajaEntero = Caja(10)
val entero: Int = cajaEntero.obtenerContenido()
```

En el ejemplo anterior, se define una interfaz genérica `Contenedor` que tiene un parámetro de tipo `T`. La interfaz `Contenedor` define un método `obtenerContenido` que devuelve el contenido del contenedor.

Se crea una clase `Caja` que implementa la interfaz `Contenedor` con un tipo de dato `T`. Se crea una instancia de la clase `Caja` con un entero y se obtiene el contenido de la caja.

Variance en genéricos

En Kotlin, puedes utilizar la anotación `in` y `out` para especificar la variabilidad de los parámetros de tipo en clases y funciones genéricas.

- `in`: Indica que el parámetro de tipo solo se utiliza en posiciones de entrada (como parámetros de métodos).
- `out`: Indica que el parámetro de tipo solo se utiliza en posiciones de salida (como valores de retorno de métodos).

```
interface Contenedor<out T> {
    fun obtenerContenido(): T
}
class Caja<in T>(val contenido: T) {
    fun ponerContenido(nuevoContenido: T) {
        // ...
    }
}
```

En el ejemplo anterior, se define una interfaz `Contenedor` con un parámetro de tipo `T` que solo se utiliza en posiciones de salida. La clase `Caja` tiene un parámetro de tipo `T` que solo se utiliza en posiciones de entrada.

Restricciones en genéricos

En Kotlin, puedes utilizar restricciones para limitar los tipos de datos que se pueden utilizar en clases y funciones genéricas. Puedes utilizar restricciones para garantizar que los tipos de datos cumplan ciertos requisitos.

```
fun <T : Number> sumar(a: T, b: T): T {
    return a + b
}
val resultadoEntero = sumar(1, 2)
val resultadoFlotante = sumar(1.5, 2.5)
```

En el ejemplo anterior, se define una función `sumar` que toma dos parámetros de tipo `T` que deben ser subtipos de `Number`. La función `sumar` devuelve la suma de los dos parámetros.

Se llama a la función `sumar` con un entero y un flotante como argumentos, y se asigna el resultado a variables de tipo `Int` y `Float`.

Genéricos en clases y funciones

En Kotlin, los genéricos te permiten escribir código que es reutilizable y flexible, ya que puedes definir clases y funciones que trabajen con cualquier tipo de datos. Puedes utilizar genéricos en clases, funciones e interfaces para crear código genérico y flexible.

```
class Caja<T>(val contenido: T) {
    fun obtenerContenido(): T {
        return contenido
    }
}
fun <T> imprimirElemento(elemento: T) {
    println(elemento)
}
interface Contenedor<T> {
    fun obtenerContenido(): T
}
fun <T : Number> sumar(a: T, b: T): T {
    return a + b
}
```

En el ejemplo anterior, se muestran ejemplos de clases genéricas, funciones genéricas e interfaces genéricas en Kotlin. Puedes utilizar genéricos para escribir código que sea reutilizable y flexible, ya que puedes definir clases y funciones que trabajen con cualquier tipo de datos.

Recursos adicionales

- Documentación oficial de Kotlin sobre genéricos
- Tutorial de Kotlin sobre genéricos
- Ejemplos de genéricos en Kotlin

Clases selladas en Kotlin

Las clases selladas son un tipo especial de clase en Kotlin que se utilizan para representar un conjunto finito de subclases.

Las clases selladas son útiles cuando tienes un conjunto limitado de subclases y quieres asegurarte de que todas las subclases se manejan de forma segura en un bloque `when`.

Declaración de clases selladas

En Kotlin, puedes declarar una clase sellada utilizando la palabra clave `sealed` antes de la palabra clave `class`.

```
sealed class Resultado {
    data class Exito(val mensaje: String) : Resultado()
    data class Error(val mensaje: String) : Resultado()
}
```

En el ejemplo anterior, se declara una clase sellada `Resultado` con dos subclases: `Exito` y `Error`. Ambas subclases tienen una propiedad `mensaje` de tipo `String`.

Uso de clases selladas

Las clases selladas se utilizan principalmente en expresiones `when` para manejar de forma segura todas las subclases.

```
fun procesarResultado(resultado: Resultado) {
    when (resultado) {
        is Resultado.Exito -> println("Exito: ${resultado.mensaje}")
        is Resultado.Error -> println("Error: ${resultado.mensaje}")
    }
}
```

En el ejemplo anterior, se define una función `procesarResultado` que toma un parámetro de tipo `Resultado`. Dentro de la expresión `when`, se manejan de forma segura las subclases `Exito` y `Error` de la clase sellada `Resultado`.

Las clases selladas son una forma segura y concisa de manejar un conjunto finito de subclases en Kotlin.

Ventajas de las clases selladas

- **Seguridad**: Las clases selladas garantizan que todas las subclases se manejen de forma segura en un bloque `when`.
- **Concisión**: Las clases selladas permiten definir un conjunto finito de subclases de forma concisa y legible.
- **Extensibilidad**: Las clases selladas pueden tener subclases anidadas, lo que permite una mayor extensibilidad y modularidad en el código.

Las clases selladas son una característica poderosa de Kotlin que te permite representar de forma segura un conjunto finito de subclases y manejarlas de forma concisa y legible en tu código.

Scope functions en Kotlin

En Kotlin, los *scope functions* son funciones que te permiten ejecutar un bloque de código en el contexto de un objeto. Estas funciones te permiten acceder a las propiedades y métodos del objeto de forma más concisa y legible.

Las *scope functions* en Kotlin son las siguientes:

- `let`
- `run`
- `with`
- `apply`
- `also`

VÍDEO EXPLICATIVO EN YOUTUBE

Kotlin Scope Functions - let, run, apply y also

`let`

La función `let` te permite ejecutar un bloque de código en el contexto de un objeto y devolver el resultado de la última expresión del bloque.

La función `let` se utiliza para realizar operaciones en un objeto y devolver un resultado.

```
val resultado = persona?.let {
    println("Nombre: ${it.nombre}")
    println("Edad: ${it.edad}")
    it.edad + 1
}
```

En el ejemplo anterior, se utiliza la función `let` para acceder a las propiedades `nombre` y `edad` del objeto `persona` y devolver la edad incrementada en 1.

`run`

A diferencia de `let`, la función `run` te permite acceder a las propiedades y métodos del objeto sin necesidad de utilizar `it`.

La función `run` se utiliza para realizar operaciones en un objeto y devolver un resultado.

```
val resultado = persona?.run {
    println("Nombre: $nombre")
    println("Edad: $edad")
    edad + 1
}
```

En el ejemplo anterior, se utiliza la función `run` para acceder a las propiedades `nombre` y `edad` del objeto `persona` y devolver la edad incrementada en 1.

with

La función `with` es similar a `run`, pero se utiliza con un objeto como argumento en lugar de un receptor.

La función `with` se utiliza para realizar operaciones en un objeto y devolver un resultado.

```
val resultado = with(persona) {
    println("Nombre: $nombre")
    println("Edad: $edad")
    edad + 1
}
```

En el ejemplo anterior, se utiliza la función `with` para acceder a las propiedades `nombre` y `edad` del objeto `persona` y devolver la edad incrementada en 1.

apply

A diferencia de las funciones anteriores, la función `apply` se utiliza para realizar operaciones en un objeto y devolver el objeto modificado.

La función `apply` se utiliza para realizar operaciones en un objeto y devolver el objeto modificado.

```
val resultado = persona?.apply {
    println("Nombre: $nombre")
    println("Edad: $edad")
    edad += 1
}
```

En el ejemplo anterior, se utiliza la función `apply` para acceder a las propiedades `nombre` y `edad` del objeto `persona` y modificar la edad incrementándola en 1.

also

La función `also` es similar a `apply`, pero se utiliza con un objeto como argumento en lugar de un receptor.

La función `also` se utiliza para realizar operaciones en un objeto y devolver el objeto original.

```
val resultado = persona?.also {
    println("Nombre: ${it.nombre}")
    println("Edad: ${it.edad}")
}
```

En el ejemplo anterior, se utiliza la función `also` para acceder a las propiedades `nombre` y `edad` del objeto `persona` y devolver el objeto original.

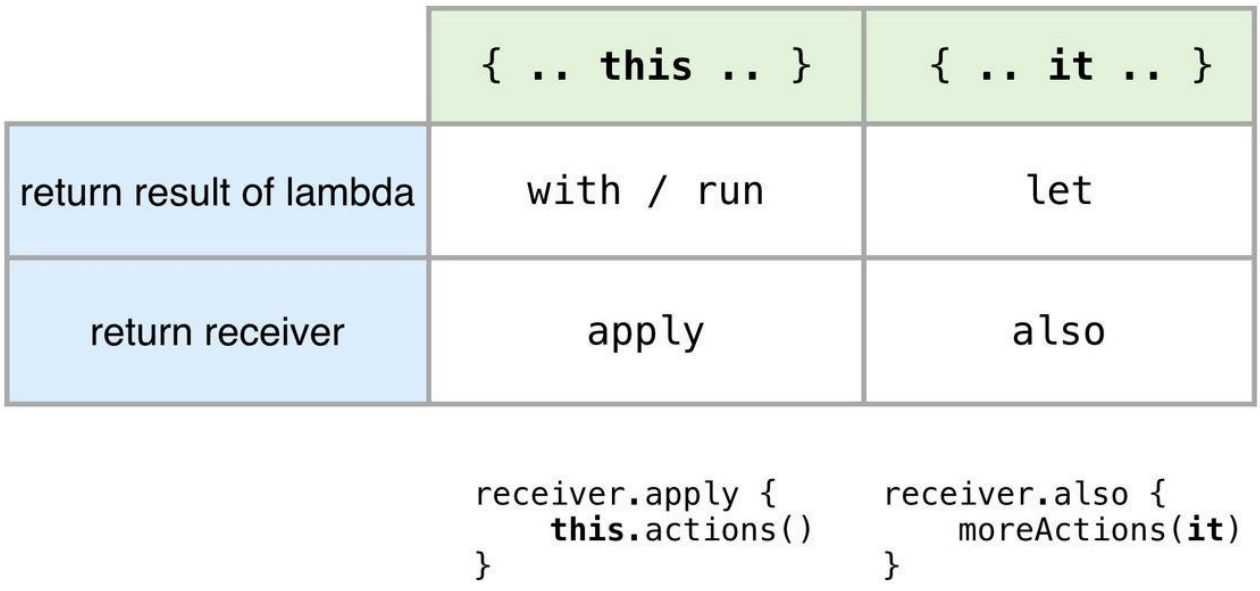
Resumen

En resumen, las *scope functions* en Kotlin te permiten ejecutar un bloque de código en el contexto de un objeto y realizar operaciones en el objeto de forma más concisa y legible. Cada función tiene un propósito específico y se utiliza en diferentes situaciones, dependiendo de tus necesidades.

Podríamos decir que las diferencias generales son las siguientes:

- `let` se utiliza para realizar operaciones en un objeto y devolver un resultado.
- `run` se utiliza para acceder a las propiedades y métodos del objeto sin necesidad de utilizar `it`.
- `with` es similar a `run`, pero se utiliza con un objeto como argumento en lugar de un receptor.
- `apply` se utiliza para realizar operaciones en un objeto y devolver el objeto modificado.
- `also` es similar a `apply`, pero se utiliza con un objeto como argumento en lugar de un receptor.

En la siguiente imagen se muestra un resumen visual de las diferencias entre las *scope functions*:



Arrays en Kotlin

En Kotlin, puedes crear arrays utilizando la función `arrayOf()`. Los arrays en Kotlin son inmutables por defecto, lo que significa que no puedes modificar su tamaño una vez que se han creado.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
```

En el ejemplo anterior, se crea un array `numeros` con los valores `1, 2, 3, 4, 5`. Los arrays en Kotlin se crean utilizando la función `arrayOf()` seguida de los valores del array entre paréntesis.

Acceso a elementos de un array

Puedes acceder a elementos individuales de un array utilizando el operador de indexación `[]` seguido del índice del elemento que deseas acceder.

```
val primerNumero = numeros[0]
val segundoNumero = numeros[1]
```

En el ejemplo anterior, se accede al primer y segundo elemento del array `numeros` utilizando los índices `0` y `1` respectivamente.

Modificación de elementos de un array

Para modificar un elemento de un array, puedes utilizar el operador de indexación `[]` seguido del índice del elemento que deseas modificar.

```
numeros[0] = 10
```

En el ejemplo anterior, se modifica el primer elemento del array `numeros` asignándole el valor `10`.

Iteración sobre un array

Puedes iterar sobre los elementos de un array utilizando un bucle `for` o la función `forEach()`.

```
for (numero in numeros) {
    println(numero)
}
numeros.forEach { numero ->
    println(numero)
}
```

En el ejemplo anterior, se itera sobre los elementos del array `numeros` utilizando un bucle `for` y la función `forEach()`.

Arrays de tipos primitivos

En Kotlin, puedes utilizar arrays de tipos primitivos como `IntArray`, `DoubleArray`, `BooleanArray`, etc. para mejorar el rendimiento de tu código.

```
val enteros = intArrayOf(1, 2, 3, 4, 5)
val dobles = doubleArrayOf(1.0, 2.0, 3.0, 4.0, 5.0)
val booleanos = booleanArrayOf(true, false, true, false)
```

En el ejemplo anterior, se crean arrays de tipos primitivos `IntArray`, `DoubleArray` y `BooleanArray` con valores iniciales.

Operaciones con arrays

Los arrays en Kotlin proporcionan una serie de operaciones útiles, como `size` para obtener el tamaño del array, `contains` para comprobar si un elemento está presente en el array, `indexOf` para obtener el índice de un elemento en el array, etc.

```
val tamaño = numeros.size
val contieneTres = numeros.contains(3)
val indiceDeCuatro = numeros.indexOf(4)
```

En el ejemplo anterior, se obtiene el tamaño del array `numeros`, se comprueba si el array contiene el número `3` y se obtiene el índice del número `4` en el array.

Listas en Kotlin

En Kotlin, puedes crear listas utilizando la función `listOf()`. Las listas en Kotlin son inmutables por defecto, lo que significa que no puedes modificar su tamaño una vez que se han creado.

```
val numeros = listOf(1, 2, 3, 4, 5)
```

En el ejemplo anterior, se crea una lista `numeros` con los valores `1, 2, 3, 4, 5`. Las listas en Kotlin se crean utilizando la función `listOf()` seguida de los valores de la lista entre paréntesis.

Acceso a elementos de una lista

Puedes acceder a elementos individuales de una lista utilizando el operador de indexación `[]` seguido del índice del elemento que deseas acceder.

```
val primerNumero = numeros[0]
val segundoNumero = numeros[1]
```

En el ejemplo anterior, se accede al primer y segundo elemento de la lista `numeros` utilizando los índices `0` y `1` respectivamente.

Iteración sobre una lista

Puedes iterar sobre los elementos de una lista utilizando un bucle `for` o la función `forEach()`.

```
for (numero in numeros) {
    println(numero)
}
numeros.forEach { numero ->
    println(numero)
}
```

En el ejemplo anterior, se itera sobre los elementos de la lista `numeros` utilizando un bucle `for` y la función `forEach()`.

Listas de tipos primitivos

En Kotlin, puedes utilizar listas de tipos primitivos como `IntList`, `DoubleList`, `BooleanList`, etc. para mejorar el rendimiento de tu código.

```
val enteros = intListOf(1, 2, 3, 4, 5)
val dobles = doubleListOf(1.0, 2.0, 3.0, 4.0, 5.0)
val booleanos = booleanListOf(true, false, true, false)
```

En el ejemplo anterior, se crean listas de tipos primitivos `IntList`, `DoubleList` y `BooleanList` con valores iniciales.

Modificación de elementos de una lista

Las listas en Kotlin son inmutables por defecto, por lo que no puedes modificar los elementos de una lista una vez que se ha creado. Si necesitas una lista mutable, puedes utilizar la función `mutableListOf()`.

```
val numeros = mutableListOf(1, 2, 3, 4, 5)
numeros[0] = 10
```

En el ejemplo anterior, se crea una lista mutable `numeros` con los valores `1, 2, 3, 4, 5`. Se modifica el primer elemento de la lista asignándole el valor `10`.

Operaciones comunes con listas

Kotlin proporciona una serie de funciones de extensión para realizar operaciones comunes con listas, como `map()`, `filter()`, `reduce()`, `sortedBy()`, etc.

```
val cuadrados = numeros.map { numero -> numero * numero }
val pares = numeros.filter { numero -> numero % 2 == 0 }
val suma = numeros.reduce { acc, numero -> acc + numero }
val ordenados = numeros.sortedBy { numero -> numero }
```

En el ejemplo anterior, se utilizan las funciones de extensión `map()`, `filter()`, `reduce()` y `sortedBy()` para realizar operaciones comunes con la lista `numeros`.

Estás funciones hacen lo siguiente:

- `map()`: Aplica una función a cada elemento de la lista y devuelve una nueva lista con los resultados.
- `filter()`: Filtra los elementos de la lista que cumplen una condición dada y devuelve una nueva lista con los elementos filtrados.
- `reduce()`: Combina los elementos de la lista en un solo valor utilizando una función dada.
- `sortedBy()`: Ordena los elementos de la lista en función de una clave dada y devuelve una nueva lista ordenada.

Otros ejemplos de operaciones comunes con listas son `sum()`, `max()`, `min()`, `average()`, `distinct()`, etc.

```
val suma = numeros.sum()
val maximo = numeros.max()
val minimo = numeros.min()
val promedio = numeros.average()
val unicos = numeros.distinct()
```

En el ejemplo anterior, se utilizan las funciones de extensión `sum()`, `max()`, `min()`, `average()` y `distinct()` para realizar operaciones comunes con la lista `numeros`.

Su funcionalidad es la siguiente:

- `sum()`: Calcula la suma de los elementos de la lista.
- `max()`: Devuelve el valor máximo de la lista.
- `min()`: Devuelve el valor mínimo de la lista.
- `average()`: Calcula el promedio de los elementos de la lista.
- `distinct()`: Devuelve una nueva lista con los elementos únicos de la lista original.

Funciones de extensión para listas

En Kotlin, puedes crear tus propias funciones de extensión para realizar operaciones específicas con listas.

```
fun List<Int>.duplicar(): List<Int> {
    return this.map { it * 2 }
}
val duplicados = numeros.duplicar()
```

En el ejemplo anterior, se define una función de extensión `duplicar()` que duplica cada elemento de una lista de enteros. La función de extensión se llama con la lista `numeros` y devuelve una nueva lista con los elementos duplicados.

Maps en Kotlin

En Kotlin, un `Map` es una colección de pares clave-valor donde cada clave está asociada a un valor. Los `Map` en Kotlin son inmutables por defecto, lo que significa que no puedes modificar su tamaño una vez que se han creado.

```
val numeros = mapOf("uno" to 1, "dos" to 2, "tres" to 3)
```

En el ejemplo anterior, se crea un `Map` `numeros` con los pares clave-valor `"uno" -> 1`, `"dos" -> 2` y `"tres" -> 3`. Los `Map` en Kotlin se crean utilizando la función `mapOf()` seguida de los pares clave-valor entre paréntesis.

Acceso a elementos de un Map

Puedes acceder a elementos individuales de un `Map` utilizando la clave del elemento que deseas acceder.

```
val numeroUno = numeros["uno"]
val numeroDos = numeros["dos"]
```

En el ejemplo anterior, se accede a los valores asociados a las claves `"uno"` y `"dos"` del `Map` `numeros`.

Iteración sobre un Map

Puedes iterar sobre los pares clave-valor de un `Map` utilizando un bucle `for` o la función `forEach()`.

```
for ((clave, valor) in numeros) {
    println("Clave: $clave, Valor: $valor")
}
numeros.forEach { (clave, valor) ->
    println("Clave: $clave, Valor: $valor")
}
```

En el ejemplo anterior, se itera sobre los pares clave-valor del `Map` `numeros` utilizando un bucle `for` y la función `forEach()`.

Maps de tipos primitivos

En Kotlin, puedes utilizar `Map` de tipos primitivos como `IntMap`, `DoubleMap`, `BooleanMap`, etc. para mejorar el rendimiento de tu código.

```
val enteros = intMapOf(1 to "uno", 2 to "dos", 3 to "tres")
val dobles = doubleMapOf(1.0 to "uno", 2.0 to "dos", 3.0 to "tres")
val booleanos = booleanMapOf(true to "verdadero", false to "falso")
```

En el ejemplo anterior, se crean `Map` de tipos primitivos `IntMap`, `DoubleMap` y `BooleanMap` con pares clave-valor iniciales.

Modificación de elementos de un Map

Los `Map` en Kotlin son inmutables por defecto, por lo que no puedes modificar los elementos de un `Map` una vez que se ha creado. Si necesitas un `Map` mutable, puedes utilizar la función `mutableMapOf()`.

```
val numeros = mutableMapOf("uno" to 1, "dos" to 2, "tres" to 3)
numeros["uno"] = 10
```

En el ejemplo anterior, se crea un `Map` mutable `numeros` con los pares clave-valor `"uno" -> 1`, `"dos" -> 2` y `"tres" -> 3`. Se modifica el valor asociado a la clave `"uno"` asignándole el valor `10`.

Operaciones con Maps

Los `Map` en Kotlin proporcionan una serie de operaciones útiles, como `get()`, `containsKey()`, `containsValue()`, `keys`, `values`, `filterKeys()`, `filterValues()`, etc.

```
val valorUno = numeros.get("uno")
val contieneDos = numeros.containsKey("dos")
val contieneCinco = numeros.containsValue(5)
val claves = numeros.keys
val valores = numeros.values
val numerosPares = numeros.filterKeys { clave -> clave.length % 2 == 0 }
val numerosImpares = numeros.filterValues { valor -> valor % 2 != 0 }
```

En el ejemplo anterior, se utilizan las operaciones `get()`, `containsKey()`, `containsValue()`, `keys`, `values`, `filterKeys()` y `filterValues()` para realizar operaciones comunes con el `Map` `numeros`.

Estas operaciones sirven para lo siguiente:

- `get()`: Obtiene el valor asociado a una clave.
- `containsKey()`: Comprueba si una clave está presente en el `Map`.
- `containsValue()`: Comprueba si un valor está presente en el `Map`.
- `keys`: Obtiene las claves del `Map`.
- `values`: Obtiene los valores del `Map`.
- `filterKeys()`: Filtra los pares clave-valor del `Map` por las claves.
- `filterValues()`: Filtra los pares clave-valor del `Map` por los valores.

Los `Map` en Kotlin son una forma eficiente de almacenar y acceder a datos asociados a claves. Puedes utilizar `Map` para representar relaciones entre objetos y realizar operaciones comunes con ellos de forma sencilla y eficiente.

Para más información sobre los `Map` en Kotlin, puedes consultar la documentación oficial de Kotlin sobre Maps.`;

Sets en Kotlin

En Kotlin, un `Set` es una colección de elementos únicos, lo que significa que no puede contener elementos duplicados. Los `Set` en Kotlin son inmutables por defecto, lo que significa que no puedes modificar su tamaño una vez que se han creado.

```
val numeros = setOf(1, 2, 3, 4, 5)
```

En el ejemplo anterior, se crea un `Set` `numeros` con los valores `1, 2, 3, 4, 5`. Los `Set` en Kotlin se crean utilizando la función `setOf()` seguida de los valores del `Set` entre paréntesis.

Acceso a elementos de un Set

Puedes acceder a elementos individuales de un `Set` utilizando la función `contains()` para comprobar si un elemento está presente en el `Set`.

```
val contieneTres = numeros.contains(3)
val contieneSeis = numeros.contains(6)
```

En el ejemplo anterior, se comprueba si el `Set` `numeros` contiene los elementos `3` y `6` utilizando la función `contains()`.

Iteración sobre un Set

Puedes iterar sobre los elementos de un `Set` utilizando un bucle `for` o la función `forEach()`.

```
for (numero in numeros) {
    println(numero)
}
numeros.forEach { numero ->
    println(numero)
}
```

En el ejemplo anterior, se itera sobre los elementos del `Set` `numeros` utilizando un bucle `for` y la función `forEach()`.

Sets de tipos primitivos

En Kotlin, puedes utilizar `Set` de tipos primitivos como `IntSet`, `DoubleSet`, `BooleanSet`, etc. para mejorar el rendimiento de tu código.

```
val enteros = intSetOf(1, 2, 3, 4, 5)
val dobles = doubleSetOf(1.0, 2.0, 3.0, 4.0, 5.0)
val booleanos = booleanSetOf(true, false, true, false)
```

En el ejemplo anterior, se crean `Set` de tipos primitivos `IntSet`, `DoubleSet` y `BooleanSet` con valores iniciales.

Modificación de elementos de un Set

Los `Set` en Kotlin son inmutables por defecto, por lo que no puedes modificar los elementos de un `Set` una vez que se ha creado. Si necesitas un `Set` mutable, puedes utilizar la función `mutableSetOf()`.

```
val numeros = mutableSetOf(1, 2, 3, 4, 5)
numeros.add(6)
```

En el ejemplo anterior, se crea un `Set` mutable `numeros` con los valores `1, 2, 3, 4, 5` y se añade el valor `6` al `Set` utilizando la función `add()`.

Operaciones con Sets

Los `Set` en Kotlin proporcionan una serie de operaciones útiles, como `size` para obtener el tamaño del `Set`, `contains` para comprobar si un elemento está presente en el `Set`, `union` para unir dos `Set`, `intersect` para obtener la intersección de dos `Set`, etc.

```
val numerosPares = setOf(2, 4, 6, 8, 10)
val numerosImpares = setOf(1, 3, 5, 7, 9)
val union = numeros.union(numerosPares)
val interseccion = numeros.intersect(numerosImpares)
val diferencia = numeros.subtract(numerosPares)
val diferenciaSimetrica = numeros.symmetricDifference(numerosPares)
val contieneTodos = numeros.containsAll(numerosPares)
val contieneAlguno = numeros.containsAny(numerosPares)
val esSubconjunto = numeros.isSubset(numerosPares)
val esSuperconjunto = numeros.isSuperset(numerosPares)
val esDisjunto = numeros.isDisjoint(numerosPares)
val esVacio = numeros.isEmpty()
```

En el ejemplo anterior, se utilizan las operaciones `union()` e `intersect()` para realizar operaciones comunes con los `Set` `numeros`, `numerosPares` y `numerosImpares`.

Estas operaciones sirven para lo siguiente:

- `union()`: Devuelve un `Set` que contiene todos los elementos de los dos `Set`.
- `intersect()`: Devuelve un `Set` que contiene los elementos comunes de los dos `Set`.
- `subtract()`: Devuelve un `Set` que contiene los elementos del primer `Set` que no están en el segundo `Set`.
- `symmetricDifference()`: Devuelve un `Set` que contiene los elementos que están en uno de los `Set` pero no en ambos.
- `containsAll()`: Comprueba si un `Set` contiene todos los elementos de otro `Set`.
- `containsAny()`: Comprueba si un `Set` contiene al menos un elemento de otro `Set`.
- `isSubset()`: Comprueba si un `Set` es un subconjunto de otro `Set`.
- `isSuperset()`: Comprueba si un `Set` es un superconjunto de otro `Set`.

- `isDisjoint()`: Comprueba si dos `Set` son disjuntos, es decir, si no tienen elementos en común.
- `isEmpty()`: Comprueba si un `Set` está vacío.

Los `Set` en Kotlin son una forma eficiente de almacenar y manipular colecciones de elementos únicos. Puedes utilizar las operaciones proporcionadas por los `Set` para realizar operaciones comunes, como unir, intersecar, restar y comparar `Set` entre sí.

Si necesitas un `Set` mutable, puedes utilizar la función `mutableSetOf()` para crear un `Set` que puedas modificar. Los `Set` en Kotlin son una herramienta poderosa que te permite trabajar con colecciones de elementos únicos de forma eficiente y concisa.

Jetpack Compose

Jetpack Compose es un marco de trabajo moderno para la creación de interfaces de usuario en aplicaciones Android. Con Compose, puedes crear interfaces de usuario de manera declarativa, lo que significa que puedes definir cómo se ve tu aplicación en función del estado de la misma.

Características de Jetpack Compose

•**Declarativo**: Con Compose, defines la interfaz de usuario de tu aplicación de manera declarativa, lo que significa que puedes describir cómo se ve tu aplicación en función del estado de la misma. Esto hace que sea más fácil de entender y mantener tu código.

•**Composable functions**: En Compose, las interfaces de usuario se crean a partir de funciones componibles, que son funciones que devuelven un árbol de elementos de la interfaz de usuario. Puedes componer estas funciones para crear interfaces de usuario complejas y reutilizables.

•**State management**: Compose tiene un sistema de manejo de estado integrado que te permite gestionar el estado de tu aplicación de manera sencilla y eficiente. Puedes definir y observar el estado de tu aplicación de forma reactiva.

•**Material Design**: Compose incluye un conjunto de widgets y estilos basados en Material Design, el lenguaje de diseño de Google para aplicaciones Android. Puedes utilizar estos widgets y estilos para crear interfaces de usuario modernas y atractivas.

•**Preview en tiempo real**: Compose incluye una función de vista previa en tiempo real que te permite ver cómo se verá tu interfaz de usuario mientras escribes código. Esto hace que sea más fácil iterar y probar tu diseño.

Apartados

•Composable functions: Aprende a crear funciones componibles en Jetpack Compose y a componerlas para crear interfaces de usuario complejas.

•State management: Descubre cómo gestionar el estado de tu aplicación en Jetpack Compose y cómo hacer que tu interfaz de usuario sea reactiva.

•Listas y cuadrículas: Aprende a mostrar listas y cuadrículas de elementos en Jetpack Compose y a personalizar su apariencia.

•Navegación y rutas: Descubre cómo implementar la navegación entre pantallas en Jetpack Compose y cómo definir rutas para tu aplicación.

•Material Design: Aprende a implementar los principios de Material Design en Jetpack Compose y a personalizar los estilos de tus componentes.

Recursos

•Documentación oficial de Jetpack Compose: La documentación oficial de Jetpack Compose, que incluye guías, tutoriales y ejemplos para aprender a usar Compose.

•Ejemplos básicos de Compose: Un repositorio con ejemplos básicos de Jetpack Compose para que puedas aprender a crear interfaces de usuario con Compose.
•Codelabs introductorios de Android: Codelabs introductorios de Android con Jetpack Compose para que puedas aprender a crear aplicaciones Android con Compose.

Funciones componibles en Jetpack Compose

En Jetpack Compose, las interfaces de usuario se crean a partir de funciones componibles, que son funciones que devuelven un árbol de elementos de la interfaz de usuario. Puedes componer estas funciones para crear interfaces de usuario complejas y reutilizables.

Crear una función componible

Para crear una función componible en Jetpack Compose, utiliza la anotación `@Composable` antes de la definición de la función. Una función componible puede tener parámetros y devolver un árbol de elementos de la interfaz de usuario utilizando las funciones de composición proporcionadas por Compose.

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

En el ejemplo anterior, se define una función componible `Greeting` que toma un parámetro `name` de tipo `String` y devuelve un elemento de texto `Text` que muestra un saludo personalizado.

Componer funciones componibles

Puedes componer funciones componibles para crear interfaces de usuario más complejas. Utiliza las funciones de composición proporcionadas por Compose, como `Column`, `Row`, `Box`, `Spacer`, etc., para organizar y diseñar los elementos de la interfaz de usuario.

```
@Composable
fun GreetingList(names: List<String>) {
    Column {
        names.forEach { name ->
            Greeting(name = name)
        }
    }
}
```

En el ejemplo anterior, se define una función componible `GreetingList` que toma una lista de nombres y muestra un saludo personalizado para cada nombre utilizando la función componible `Greeting`.

Actualización de funciones componibles

Las funciones componibles en Jetpack Compose son reactivas, lo que significa que se vuelven a ejecutar automáticamente cuando cambian los datos de entrada. Esto permite que la interfaz de usuario se actualice de forma dinámica en respuesta a los cambios en los datos.

```
@Composable
fun Counter(count: Int) {
    Text(text = "Count: $count")
}
```

En el ejemplo anterior, se define una función componible `Counter` que muestra el recuento actual. Cuando cambia el recuento, la función componible se vuelve a ejecutar automáticamente para reflejar el nuevo valor.

Columnas y filas en Jetpack Compose+

Jetpack Compose proporciona las funciones `Column` y `Row` para organizar elementos de la interfaz de usuario en columnas y filas respectivamente. Puedes anidar columnas y filas para crear diseños más complejos y reutilizables.

```
@Composable
fun GreetingList(names: List<String>) {
    Column {
        names.forEach { name ->
            Greeting(name = name)
            Spacer(modifier = Modifier.height(8.dp))
        }
    }
}
```

En el ejemplo anterior, se utiliza la función `Column` para organizar los saludos en una lista vertical. Se añade un `Spacer` entre cada saludo para separarlos visualmente.

Modificadores en Jetpack Compose

Jetpack Compose utiliza modificadores para aplicar estilos y comportamientos a los elementos de la interfaz de usuario. Puedes utilizar modificadores para cambiar el tamaño, la posición, el color, la forma, etc., de los elementos de la interfaz de usuario.

```
@Composable
fun Greeting(name: String) {
    Text(
        text = "Hello, $name!",
        modifier = Modifier
            .padding(16.dp)
            .background(Color.Blue)
            .clickable { /* Acción al hacer clic */ }
    )
}
```

En el ejemplo anterior, se utiliza el modificador `padding` para añadir un relleno alrededor del texto, el modificador `background` para cambiar el color de fondo del texto, y el modificador `clickable` para añadir una acción al hacer clic en el texto.

Recursos en Jetpack Compose

Jetpack Compose utiliza el sistema de recursos de Android para gestionar los recursos de la interfaz de usuario, como cadenas, colores, dimensiones, etc. Puedes acceder a los recursos utilizando la función `stringResource()`, `colorResource()`, `dimenResource()`, etc.

```
@Composable
fun Greeting() {
```

```
    Text(
        text = stringResource(id = R.string.hello),
        color = colorResource(id = R.color.primary),
        fontSize = dimenResource(id = R.dimen.text_size)
    )
}
```

En el ejemplo anterior, se utiliza la función `stringResource()` para obtener una cadena de recursos, la función `colorResource()` para obtener un color de recursos, y la función `dimenResource()` para obtener una dimensión de recursos.

Temas en Jetpack Compose

Jetpack Compose utiliza el sistema de temas de Android para aplicar estilos coherentes a la interfaz de usuario. Puedes definir un tema personalizado utilizando la función `provideAppTheme()` y aplicarlo a tu aplicación utilizando el modificador `MaterialTheme`.

```
@Composable
fun MyApp() {
    MaterialTheme(
        colors = lightColors(),
        typography = Typography,
        shapes = Shapes
    ) {
        Greeting()
    }
}
```

En el ejemplo anterior, se define un tema personalizado con colores, tipografía y formas personalizadas, y se aplica a la aplicación utilizando el modificador `MaterialTheme`.

Ejemplos de funciones componibles

Ejemplo de función con Row

```
@Composable
fun Greeting(name: String) {
    Row {
        Text(text = "Hello, $name!")
        Spacer(modifier = Modifier.width(8.dp))
        Icon(Icons.Default.Favorite, contentDescription = null)
    }
}
```

En el ejemplo anterior, se utiliza la función `Row` para organizar el texto y el icono en una fila horizontal.

Ejemplo de función con Box

```
@Composable
fun Greeting(name: String) {
    Box(
        modifier = Modifier
            .background(Color.Blue)
```

```
            .padding(16.dp)
    ) {
        Text(text = "Hello, $name!", color = Color.White)
    }
}
```

En el ejemplo anterior, se utiliza la función `Box` para colocar el texto en un cuadro azul con un relleno de 16 dp.

Ejemplo de función con Column

```
@Composable
fun GreetingList(names: List<String>) {
    Column {
        names.forEach { name ->
            Greeting(name = name)
            Divider(color = Color.Gray, thickness = 1.dp)
        }
    }
}
```

En el ejemplo anterior, se utiliza la función `Column` para organizar los saludos en una lista vertical con una línea divisoria entre cada saludo.

Cómo añadir imágenes usando Image y painterResource

```
@Composable
fun Greeting(name: String) {
    Row {
        Image(
            painter = painterResource(id = R.drawable.ic_launcher_foreground),
            contentDescription = null,
            modifier = Modifier.size(48.dp)
        )
        Spacer(modifier = Modifier.width(8.dp))
        Text(text = "Hello, $name!")
    }
}
```

En el ejemplo anterior, se utiliza la función `Image` para añadir una imagen a la interfaz de usuario utilizando un recurso de imagen.

Previews en Jetpack Compose

Jetpack Compose proporciona una función `@Preview` que te permite previsualizar tus funciones componibles en tiempo real en Android Studio. Puedes definir previsualizaciones para tus funciones componibles y ver cómo se ven en diferentes configuraciones y estados.

```
@Preview
@Composable
fun GreetingPreview() {
    Greeting(name = "World")
}
```

En el ejemplo anterior, se define una previsualización `GreetingPreview` para la función componible `Greeting` con el nombre "World". Puedes ver la previsualización en Android Studio y ajustarla según sea necesario.

Opciones de previsualización

Jetpack Compose proporciona varias opciones de previsualización que te permiten personalizar la apariencia de tus previsualizaciones. Puedes definir diferentes configuraciones, tamaños, orientaciones, temas, etc., para tus previsualizaciones y ver cómo se ven en diferentes contextos.

```
@Preview(
    showBackground = true,
    name = "Greeting Preview",
    uiMode = Configuration.UI_MODE_NIGHT_YES,
    widthDp = 320,
    heightDp = 240
)
@Composable
fun GreetingPreview() {
    Greeting(name = "World")
}
```

Las opciones de previsualización son las siguientes:

- `showBackground`: Muestra un fondo en la previsualización.
- `name`: Nombre de la previsualización.
- `uiMode`: Modo de interfaz de usuario (claro, oscuro, etc.).
- `widthDp`: Ancho de la previsualización en dp.
- `heightDp`: Alto de la previsualización en dp.

Opciones de alineación en Jetpack Compose

Jetpack Compose proporciona opciones de alineación que te permiten alinear los elementos de la interfaz de usuario de forma horizontal y vertical. Puedes utilizar las opciones de alineación para controlar la posición de los elementos en la pantalla y crear diseños más precisos y coherentes.

```
@Composable
fun Greeting(name: String) {
    Text(
        text = "Hello, $name!",
        modifier = Modifier.align(Alignment.CenterHorizontally)
    )
}
```

En el ejemplo anterior, se utiliza el modificador `align` para alinear el texto horizontalmente en el centro de la pantalla.

Opciones de alineación horizontal

Las opciones de alineación horizontal en Jetpack Compose son las siguientes:

- `start`: Alinea el elemento al principio del eje horizontal.

- `centerHorizontally`: Alinea el elemento en el centro del eje horizontal.
- `end`: Alinea el elemento al final del eje horizontal.

Opciones de alineación vertical

Las opciones de alineación vertical en Jetpack Compose son las siguientes:

- `top`: Alinea el elemento en la parte superior del eje vertical.
- `centerVertically`: Alinea el elemento en el centro del eje vertical.
- `bottom`: Alinea el elemento en la parte inferior del eje vertical.

Opciones de alineación personalizadas

Además de las opciones de alineación predefinidas, Jetpack Compose te permite crear opciones de alineación personalizadas utilizando la función `Alignment`.

```
val CustomAlignment = Alignment(0.25f, 0.75f)
```

En el ejemplo anterior, se define una opción de alineación personalizada `CustomAlignment` con un desplazamiento horizontal del 25% y un desplazamiento vertical del 75%.

Uso de opciones de alineación en Column y Row

Puedes utilizar las opciones de alineación en las funciones `Column` y `Row` para alinear los elementos de la interfaz de usuario de forma horizontal y vertical.

```
@Composable
fun Greeting(name: String) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.SpaceEvenly
    ) {
        Text(text = "Hello, $name!")
        Icon(Icons.Default.Favorite, contentDescription = null)
    }
}
```

En el ejemplo anterior, se utiliza la opción de alineación vertical `Alignment.CenterVertically` y la disposición horizontal `Arrangement.SpaceEvenly` para alinear el texto y el icono en una fila horizontal.

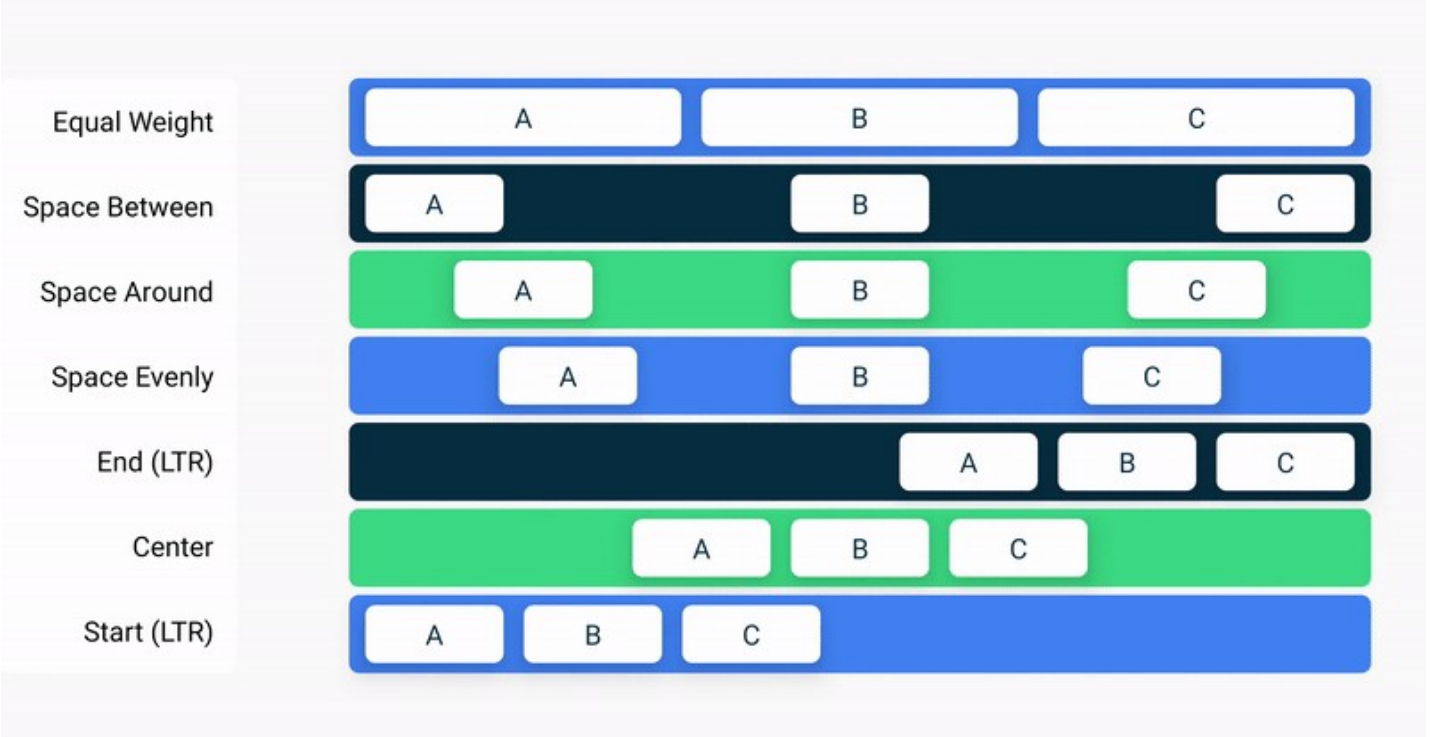
Las diferentes opciones de alineación y disposición te permiten crear diseños flexibles y personalizados en Jetpack Compose.

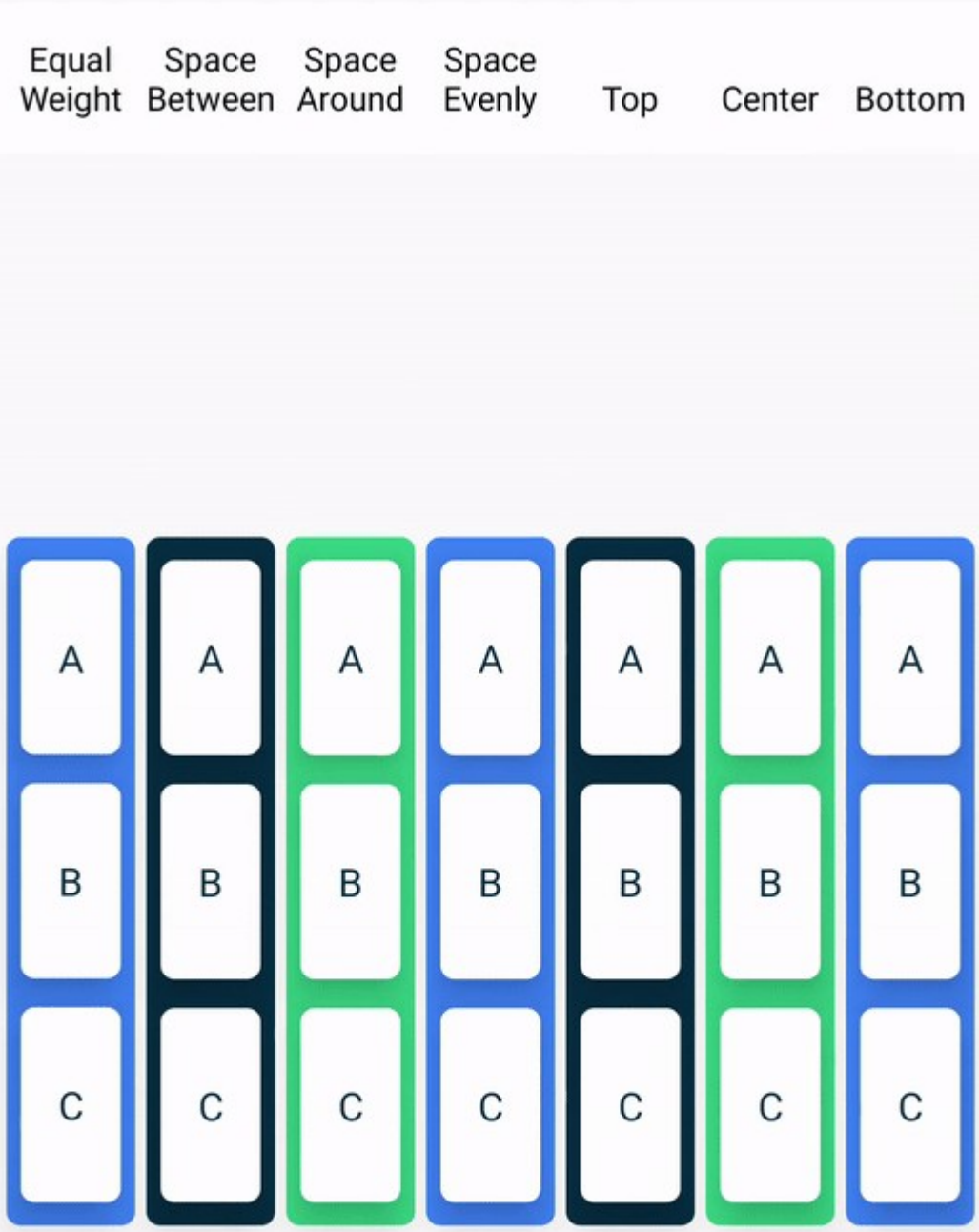
Para el Arrangement existen las siguientes opciones:

- `SpaceAround`: Distribuye el espacio entre los elementos de forma uniforme, con espacio adicional alrededor de los elementos.
- `SpaceBetween`: Distribuye el espacio entre los elementos de forma uniforme, sin espacio adicional alrededor de los elementos.

- `SpaceEvenly`: Distribuye el espacio entre los elementos de forma uniforme, con espacio adicional alrededor de los elementos y en los extremos.
- `Center`: Centra los elementos en el espacio disponible.
- `Start`: Coloca los elementos al principio del espacio disponible.
- `End`: Coloca los elementos al final del espacio disponible.

En las siguientes imágenes animadas se muestran ejemplos de alineación horizontal y vertical en Jetpack Compose:




Equal Weight
Space Between
Space Around
Space Evenly
Top
Center
Bottom
A
B
C

Más ejemplos de modificaciones

Modificador de tamaño

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.size(48.dp))
}
```

En el ejemplo anterior, se utiliza el modificador `size` para cambiar el tamaño del texto a 48 dp.

Modificador de altura y anchura

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier =
Modifier.width(100.dp).height(50.dp))
}
```

En el ejemplo anterior, se utilizan los modificadores `width` y `height` para cambiar la anchura del texto a 100 dp y la altura a 50 dp.

Modificador de tipografía

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", style = TextStyle(fontWeight =
FontWeight.Bold))
}
```

En el ejemplo anterior, se utiliza el modificador `style` para cambiar el peso de la fuente del texto a negrita.

Modificador de alineación

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier =
Modifier.align(Alignment.CenterHorizontally))
}
```

En el ejemplo anterior, se utiliza el modificador `align` para alinear el texto horizontalmente en el centro.

Modificador de margen

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.padding(16.dp))
}
```

En el ejemplo anterior, se utiliza el modificador `padding` para añadir un margen de 16 dp alrededor del texto.

Modificador de color de fondo

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.background(Color.Blue))
}
```

En el ejemplo anterior, se utiliza el modificador `background` para cambiar el color de fondo del texto a azul.

Modificador de borde

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.border(1.dp, Color.Black))
}
```

En el ejemplo anterior, se utiliza el modificador `border` para añadir un borde de 1 dp de grosor alrededor del texto.

Modificador de clic

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.clickable { /* Acción al
hacer clic */ })
}
```

En el ejemplo anterior, se utiliza el modificador `clickable` para añadir una acción al hacer clic en el texto.

Modificador de forma

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier =
Modifier.clip(RoundedCornerShape(4.dp)))
}
```

En el ejemplo anterior, se utiliza el modificador `clip` para aplicar una forma redondeada con un radio de 4 dp alrededor del texto.

Modificador de rotación

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.rotate(45f))
}
```

En el ejemplo anterior, se utiliza el modificador `rotate` para rotar el texto 45 grados.

Modificador de escala

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.scale(1.5f))
```

```
}
```

En el ejemplo anterior, se utiliza el modificador `scale` para escalar el texto a 1.5 veces su tamaño original.

Modificador de desplazamiento

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.offset(x = 16.dp, y =
16.dp))
}
```

En el ejemplo anterior, se utiliza el modificador `offset` para desplazar el texto 16 dp hacia la derecha y 16 dp hacia abajo.

Modificador de sombra

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.shadow(4.dp, shape =
CircleShape))
}
```

En el ejemplo anterior, se utiliza el modificador `shadow` para añadir una sombra de 4 dp alrededor del texto con una forma circular.

Modificador de desenfoque

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!", modifier = Modifier.blur(4.dp))
}
```

En el ejemplo anterior, se utiliza el modificador `blur` para aplicar un efecto de desenfoque al texto con un radio de 4 dp.

Otros ejemplos de funciones componibles de interés

El uso de Spacer

```
@Composable
fun Greeting(name: String) {
    Row {
        Text(text = "Hello, $name!")
        Spacer(modifier = Modifier.width(8.dp))
        Icon(Icons.Default.Favorite, contentDescription = null)
    }
}
```

En el ejemplo anterior, se utiliza la función `Spacer` para añadir un espacio entre el texto y el icono en una fila horizontal.

Gestión de estado en Jetpack Compose

Jetpack Compose es un marco de trabajo moderno para la creación de interfaces de usuario en aplicaciones Android. Con Compose, puedes crear interfaces de usuario de manera declarativa, lo que significa que puedes definir cómo se ve tu aplicación en función del estado de la misma.

Estado en Jetpack Compose

En Jetpack Compose, el estado es una parte fundamental de la arquitectura de tu aplicación.

El estado representa la información que puede cambiar a lo largo del tiempo y que afecta a la apariencia y el comportamiento de tu interfaz de usuario.

Puedes definir y observar el estado de tu aplicación de forma reactiva en Compose.

Definición de estado

Puedes definir el estado de tu aplicación utilizando la función `mutableStateOf()` de Compose.

```
val contador = mutableStateOf(0)
```

En el ejemplo anterior, se define un estado `contador` con un valor inicial de `0`.

Observación de estado

Puedes observar el estado de tu aplicación utilizando la función `observeAsState()` de Compose.

```
val contadorState = contador.observeAsState()
val contador = contadorState.value
```

En el ejemplo anterior, se observa el estado `contador` y se obtiene su valor actual.

Actualización de estado

Puedes actualizar el estado de tu aplicación utilizando la función `value` de Compose.

```
contador.value++
```

En el ejemplo anterior, se incrementa en uno el valor del estado `contador`.

¡IMPORTANTE!

El estado en Compose es inmutable, por lo que debes utilizar la función `value` para actualizar el estado.

Para que haya recomposición, la actualización del estado debe realizarse dentro de un evento de un componente `@Composable`.

Ejemplo completo

```
@Composable
fun Contador() {
    val contador = mutableStateOf(0)
    val contadorState = contador.observeAsState()

    Button(onClick = { contador.value++ }) {
        Text(text = "Contador: ${contadorState.value}")
    }
}
```

En el ejemplo anterior, se define un componente `Contador` que muestra un botón y un texto con el valor del estado `contador`. Al hacer clic en el botón, se incrementa en uno el valor del estado `contador`.

Tipos de estado

En Jetpack Compose, puedes utilizar diferentes tipos de estado para gestionar la información de tu aplicación de forma reactiva.

- `mutableStateOf()`: Crea un estado mutable que puede cambiar a lo largo del tiempo.
- `remember`: Crea un estado que se mantiene entre recomposiciones.
- `derivedStateOf()`: Crea un estado derivado a partir de otros estados.

Uso de remember

La función `remember` de Compose te permite crear un estado que se mantiene entre recomposiciones.

```
@Composable
fun Contador() {
    val contador = remember { mutableStateOf(0) }
    val contadorState = contador.observeAsState()

    Button(onClick = { contador.value++ }) {
        Text(text = "Contador: ${contadorState.value}")
    }
}
```

En el ejemplo anterior, se utiliza la función `remember` para crear un estado `contador` que se mantiene entre recomposiciones.

Uso de rememberSaveable

La función `rememberSaveable` de Compose te permite crear un estado que se mantiene entre configuraciones.

```
@Composable
fun Contador() {
    val contador = rememberSaveable { mutableStateOf(0) }
    val contadorState = contador.observeAsState()
```

```
    Button(onClick = { contador.value++ }) {
        Text(text = "Contador: ${contadorState.value}")
    }
}
```

En el ejemplo anterior, se utiliza la función `rememberSaveable` para crear un estado `contador` que se mantiene entre configuraciones.

REMEMBER VS REMEMBERSAVEABLE

La diferencia entre `remember` y `rememberSaveable` es que `rememberSaveable` guarda el estado en el `Bundle` de la actividad para que se pueda restaurar después de una recreación de la actividad.

Esto es útil para guardar el estado de la aplicación cuando la actividad se destruye y se vuelve a crear, por ejemplo, al girar la pantalla.

Uso de derivedStateOf

La función `derivedStateOf` de Compose te permite crear un estado derivado a partir de otros estados.

```
@Composable
fun Contador() {
    val contador = mutableStateOf(0)
    val doble = derivedStateOf { contador.value * 2 }

    Button(onClick = { contador.value++ }) {
        Text(text = "Contador: ${contador.value}, Doble: ${doble.value}")
    }
}
```

En el ejemplo anterior, se utiliza la función `derivedStateOf` para crear un estado `doble` que es el doble del estado `contador`.

Flows en Kotlin

En Kotlin, un `Flow` es una secuencia de valores que se emiten de forma asíncrona y reactiva. Los `Flow` te permiten trabajar con datos de forma reactiva y gestionar la concurrencia de forma sencilla.

Puedes crear un `Flow` utilizando la función `flowOf()` de Kotlin.

```
val numeros = flowOf(1, 2, 3, 4, 5)
```

En el ejemplo anterior, se crea un `Flow` `numeros` con los valores `1, 2, 3, 4, 5`.

Observación de Flows

Puedes observar un `Flow` utilizando la función `collect()` de Kotlin.

```
numeros.collect { numero ->
```

```
    println(numero)
}
```

En el ejemplo anterior, se observa el `Flow` `numeros` y se imprime cada valor que se emite.

Transformación de Flows

Puedes transformar un `Flow` utilizando operadores como `map`, `filter`, `flatMap`, etc.

```
val cuadrados = numeros.map { numero -> numero * numero }
val pares = numeros.filter { numero -> numero % 2 == 0 }
```

En el ejemplo anterior, se utilizan los operadores `map` y `filter` para transformar el `Flow` `numeros`.

Flows en Jetpack Compose

En Jetpack Compose, puedes utilizar Flows para gestionar la información de tu aplicación de forma reactiva.

Puedes convertir un `Flow` en un estado observable utilizando la función `collectAsState()` de Compose.

```
val numeros = flowOf(1, 2, 3, 4, 5)
val numerosState = numeros.collectAsState()
```

En el ejemplo anterior, se convierte el `Flow` `numeros` en un estado observable `numerosState`.

Elevación del estado

En Jetpack Compose, puedes elevar el estado de un componente para compartirlo con otros componentes.

Esto te permite gestionar el estado de tu aplicación de forma centralizada y compartirlo entre diferentes partes de tu interfaz de usuario.

```
@Composable
fun Contador() {
    val contador = remember { mutableStateOf(0) }
    ContadorBoton(contador)
    ContadorTexto(contador)
}
@Composable
fun ContadorBoton(contador: MutableState<Int>) {
    Button(onClick = { contador.value++ }) {
        Text(text = "Incrementar")
    }
}
@Composable
fun ContadorTexto(contador: MutableState<Int>) {
    Text(text = "Contador: ${contador.value}")
}
```

En el ejemplo anterior, se eleva el estado `contador` del componente `Contador` para compartirlo con los componentes `ContadorBoton` y `ContadorTexto`.

De esta forma, el estado `contador` se gestiona de forma centralizada en el componente `Contador` y se comparte con los componentes hijos. Y ambos se recomponen cuando el estado cambia.

Esto también facilita la reutilización de los componentes y la separación de las preocupaciones en tu aplicación. Además de facilitar la prueba y el mantenimiento del código.

ELEVACIÓN DEL ESTADO VS. INYECCIÓN DE DEPENDENCIAS

La elevación del estado es una técnica común en Jetpack Compose para compartir el estado entre componentes.

Otra técnica común es la inyección de dependencias, que consiste en pasar el estado como argumento a los componentes que lo necesitan.

Ambas técnicas tienen sus ventajas y desventajas, y la elección entre ellas depende del diseño y la arquitectura de tu aplicación.

Conclusión

La gestión de estado es una parte fundamental de la arquitectura de tu aplicación en Jetpack Compose.

Con Compose, puedes definir, observar y actualizar el estado de tu aplicación de forma reactiva y declarativa.

Los Flows te permiten trabajar con datos de forma reactiva y gestionar la concurrencia de forma sencilla en Kotlin.

Ejemplo de gestión del estado básica en una app de Contador

```
@Composable
fun Contador() {
    val contador = remember { mutableStateOf(0) }
    val contadorState = contador.observeAsState()

    Button(onClick = { contador.value++ }) {
        Text(text = "Contador: ${contadorState.value}")
    }
}
```

En el ejemplo anterior, se define un componente `Contador` que muestra un botón y un texto con el valor del estado `contador`. Al hacer clic en el botón, se incrementa en uno el valor del estado `contador`.

Recursos

- Documentación oficial de Jetpack Compose: La documentación oficial de Jetpack Compose, que incluye guías, tutoriales y ejemplos para aprender a usar Compose.

•Ejemplos básicos de Compose: Un repositorio con ejemplos básicos de Jetpack Compose para que puedas aprender a crear interfaces de usuario con Compose.
•Avanzando con Kotlin y el manejo de la UI - Codelabs
•Más Kotlin y listas de elementos (LazyColumn) – Codelabs

Listas en Compose

En Jetpack Compose, puedes mostrar listas de elementos utilizando los componentes `LazyColumn` y `LazyRow`. Estos componentes te permiten mostrar una lista de elementos de forma eficiente y reactiva.

LazyColumn

El componente `LazyColumn` te permite mostrar una lista de elementos de forma eficiente y reactiva. Puedes utilizar `LazyColumn` para mostrar una lista de elementos verticales que se cargan de forma perezosa a medida que el usuario se desplaza por la lista.

```
@Composable
fun ListaVertical() {
    LazyColumn {
        items(100) { index ->
            Text(text = "Elemento $index")
        }
    }
}
```

En el ejemplo anterior, se define un componente `ListaVertical` que muestra una lista de 100 elementos verticales utilizando `LazyColumn`. El método `items()` de `LazyColumn` se utiliza para generar los elementos de la lista en función de un rango de índices.

LazyRow

El componente `LazyRow` te permite mostrar una lista de elementos de forma eficiente y reactiva. Puedes utilizar `LazyRow` para mostrar una lista de elementos horizontales que se cargan de forma perezosa a medida que el usuario se desplaza por la lista.

```
@Composable
fun ListaHorizontal() {
    LazyRow {
        items(100) { index ->
            Text(text = "Elemento $index")
        }
    }
}
```

En el ejemplo anterior, se define un componente `ListaHorizontal` que muestra una lista de 100 elementos horizontales utilizando `LazyRow`. El método `items()` de `LazyRow` se utiliza para generar los elementos de la lista en función de un rango de índices.

Scroll infinito

Puedes implementar el scroll infinito en las listas de Compose utilizando el método `items()` de `LazyColumn` o `LazyRow` y pasando una lista infinita como argumento. Esto te permite cargar nuevos elementos a medida que el usuario se desplaza por la lista.

```
@Composable
```

```
fun ScrollInfinito() {
    val elementos = remember { mutableStateListOf<String>() }
    LazyColumn {
        items(elementos) { elemento ->
            Text(text = elemento)
        }
    }
}
```

En el ejemplo anterior, se define un componente `ScrollInfinito` que muestra una lista de elementos utilizando `LazyColumn`. Se utiliza un estado mutable `mutableStateListOf` para almacenar los elementos de la lista, y se pasa esta lista como argumento al método `items()` de `LazyColumn`. Esto permite cargar nuevos elementos a medida que el usuario se desplaza por la lista.

El manejo del scroll infinito en Compose es similar al manejo del scroll infinito en otras bibliotecas de UI, como RecyclerView en Android. Puedes utilizar técnicas como la paginación y la carga perezosa para cargar nuevos elementos a medida que el usuario se desplaza por la lista.

El estado mutable `mutableStateListOf` se utiliza para almacenar los elementos de la lista y notificar a Compose cuando se actualizan los elementos. Esto permite que Compose vuelva a renderizar la lista con los nuevos elementos.

Elementos personalizados

Puedes utilizar elementos personalizados en las listas de Compose para mostrar elementos más complejos. Para ello, puedes utilizar el método `item()` de `LazyColumn` o `LazyRow` y pasar un componente personalizado como argumento.

```
@Composable
fun ListaPersonalizada() {
    LazyColumn {
        items(100) { index ->
            ElementoPersonalizado(index)
        }
    }
}
@Composable
fun ElementoPersonalizado(index: Int) {
    Text(text = "Elemento $index")
}
```

En el ejemplo anterior, se define un componente `ListaPersonalizada` que muestra una lista de 100 elementos utilizando `LazyColumn`. El método `items()` de `LazyColumn` se utiliza para generar los elementos de la lista en función de un rango de índices, y se pasa un componente personalizado `ElementoPersonalizado` como argumento.

Escuchadores de eventos

Puedes añadir escuchadores de eventos a los elementos de la lista para responder a las interacciones del usuario. Por ejemplo, puedes añadir un escuchador de clics a un elemento de la lista para realizar una acción cuando el usuario haga clic en él.

```
@Composable
fun ListaConClics() {
    LazyColumn {
        items(100) { index ->
            Text(
                text = "Elemento $index",
                modifier = Modifier.clickable { /* Acción al hacer clic */ }
            )
        }
    }
}
```

En el ejemplo anterior, se define un componente `ListaConClics` que muestra una lista de 100 elementos utilizando `LazyColumn`. Se añade un escuchador de clics al elemento de la lista utilizando el modificador `clickable`.

Separadores

Puedes añadir separadores entre los elementos de la lista utilizando el método `item()` de `LazyColumn` o `LazyRow` y pasando un componente separador como argumento.

```
@Composable
fun ListaConSeparadores() {
    LazyColumn {
        items(100) { index ->
            Column {
                ElementoPersonalizado(index)
                Divider()
            }
        }
    }
}
```

En el ejemplo anterior, se define un componente `ListaConSeparadores` que muestra una lista de 100 elementos utilizando `LazyColumn`. Se añade un separador `Divider()` entre cada elemento de la lista.

Filtrado y ordenación

Puedes filtrar y ordenar los elementos de la lista utilizando funciones de extensión como `filter()` y `sortedBy()`. Estas funciones te permiten realizar operaciones comunes con las listas de forma sencilla y eficiente.

```
@Composable
fun ListaFiltrada() {
    val numeros = (0..100).toList()
    LazyColumn {
        items(numeros.filter { it % 2 == 0 }) { numero ->
            Text(text = "Número $numero")
        }
    }
}
```

En el ejemplo anterior, se define un componente `ListaFiltrada` que muestra una lista de números pares del 0 al 100 utilizando `LazyColumn`. Se filtran los números pares utilizando la función de extensión `filter()`.

Porqué usar LazyColumn y LazyRow en lugar de Column y Row

`LazyColumn` y `LazyRow` son componentes optimizados para mostrar listas de elementos de forma eficiente y reactiva. A diferencia de `Column` y `Row`, que renderizan todos los elementos de la lista de forma inmediata, `LazyColumn` y `LazyRow` renderizan solo los elementos visibles en la pantalla y los elementos que están cerca de la zona visible.

Esto hace que `LazyColumn` y `LazyRow` sean más eficientes en términos de rendimiento y consumo de recursos, especialmente cuando se trabaja con listas grandes o infinitas.

Por lo tanto, es recomendable utilizar `LazyColumn` y `LazyRow` para mostrar listas de elementos en Jetpack Compose, ya que proporcionan una experiencia de usuario más fluida y eficiente.

Cuadrículsa o Grids

En Jetpack Compose, puedes mostrar listas de elementos en forma de cuadrículas utilizando el componente `LazyVerticalGrid` o `LazyHorizontalGrid`. Estos componentes te permiten mostrar una cuadrícula de elementos de forma eficiente y reactiva.

```
@Composable
fun CuadriculaVertical() {
    LazyVerticalGrid(cells = GridCells.Fixed(3)) {
        items(100) { index ->
            Text(text = "Elemento $index")
        }
    }
}
```

En el ejemplo anterior, se define un componente `CuadriculaVertical` que muestra una cuadrícula de 100 elementos en 3 columnas utilizando `LazyVerticalGrid`. El método `items()` de `LazyVerticalGrid` se utiliza para generar los elementos de la cuadrícula en función de un rango de índices.

```
@Composable
fun CuadriculaHorizontal() {
    LazyHorizontalGrid(cells = GridCells.Fixed(3)) {
        items(100) { index ->
            Text(text = "Elemento $index")
        }
    }
}
```

En el ejemplo anterior, se define un componente `CuadriculaHorizontal` que muestra una cuadrícula de 100 elementos en 3 filas utilizando `LazyHorizontalGrid`. El

método `items()` de `LazyHorizontalGrid` se utiliza para generar los elementos de la cuadrícula en función de un rango de índices.

Navegación y rutas en Jetpack Compose

Jetpack Compose es un marco de trabajo moderno para la creación de interfaces de usuario en aplicaciones Android. Con Compose, puedes crear interfaces de usuario de manera declarativa, lo que significa que puedes definir cómo se ve tu aplicación en función del estado de la misma.

Configuración de la navegación

Para administrar la navegación en Jetpack Compose, necesitas agregar la dependencia de `navigation-compose` a tu archivo `build.gradle` y definir las rutas y destinos de tu aplicación en un archivo de configuración de navegación.

build.gradle.kts

```
dependencies {
    val nav_version = "2.8.5"
    implementation("androidx.navigation:navigation-compose:$nav_version")
}
```

Navegación en Jetpack Compose

En Jetpack Compose, puedes implementar la navegación entre pantallas utilizando el componente `NavHost`. `NavHost` es un contenedor que muestra las pantallas de tu aplicación en función de las rutas definidas.

```
@Composable
fun MyApp() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "pantalla1") {
        composable("pantalla1") {
            Pantalla1()
        }
        composable("pantalla2") {
            Pantalla2()
        }
    }
}
```

En el ejemplo anterior, se define un componente `MyApp` que utiliza `NavHost` para implementar la navegación entre dos pantallas: `Pantalla1` y `Pantalla2`. `NavHost` toma un `NavController` como argumento y define las rutas de las pantallas utilizando el método `composable`.

Definición de rutas

Puedes definir las rutas de tu aplicación utilizando el método `composable` de `NavHost`. Cada ruta tiene un nombre único que se utiliza para identificar la pantalla correspondiente.

```
NavHost(navController = navController, startDestination = "pantalla1") {
    composable("pantalla1") {
        Pantalla1()
    }
```

```
    composable("pantalla2") {
        Pantalla2()
    }
}
```

En el ejemplo anterior, se definen dos rutas: `pantalla1` y `pantalla2`, que corresponden a las pantallas `Pantalla1` y `Pantalla2`, respectivamente.

Navegación entre pantallas

Puedes navegar entre pantallas en Jetpack Compose utilizando el `NavController` y el método `navigate`.

```
Button(onClick = { navController.navigate("pantalla2") }) {
    Text(text = "Ir a Pantalla 2")
}
```

En el ejemplo anterior, se define un botón que, al hacer clic en él, navega a la pantalla `Pantalla2` utilizando el método `navigate` del `NavController`.

Paso de datos entre pantallas

Puedes pasar datos entre pantallas en Jetpack Compose utilizando el método `navigate` y el argumento `arguments`.

```
Button(onClick = {
    navController.navigate("pantalla2") {
        launchSingleTop = true
        popUpTo("pantalla1") { inclusive = true }
        arguments = bundleOf("dato" to "Hola, mundo!")
    }
}) {
    Text(text = "Ir a Pantalla 2")
}
```

En el ejemplo anterior, se define un botón que, al hacer clic en él, navega a la pantalla `Pantalla2` y pasa el dato `"Hola, mundo!"` como argumento.

El argumento `arguments` se utiliza para pasar datos entre pantallas. Puedes utilizar `bundleOf` para crear un `Bundle` con los datos que deseas pasar.

Recuperación de datos en la pantalla de destino

Puedes recuperar los datos pasados como argumentos en la pantalla de destino utilizando el método `currentBackStackEntryAsState` y el argumento `arguments`.

```
val navBackStackEntry by navController.currentBackStackEntryAsState()
val dato = navBackStackEntry?.arguments?.getString("dato")
```

En el ejemplo anterior, se recupera el dato pasado como argumento en la pantalla `Pantalla2` utilizando el método `currentBackStackEntryAsState` y el argumento `arguments`.

Navegación entre pantallas usando lambdas

En Jetpack Compose, puedes utilizar lambdas para definir la navegación entre pantallas de forma más concisa.

La definición de estas lambdas puede hacerse en el método `composable` de `NavHost`. Esto nos ayuda a mantener un código más limpio y legible.

```
NavHost(navController = navController, startDestination = "pantalla1") {
    composable("pantalla1") {
        Pantalla1 {
            navController.navigate("pantalla2")
        }
    }
    composable("pantalla2") {
        Pantalla2 {
            navController.navigate("pantalla1")
        }
    }
}
```

En el ejemplo anterior, se define la navegación entre las pantallas `Pantalla1` y `Pantalla2` utilizando lambdas en el método `composable` de `NavHost`.

De esta forma, al definir la navegación en el propio componente, se mantiene un código más limpio y legible.

Por ejemplo, el composable `Pantalla1` recibe una lambda que navega a la pantalla `Pantalla2`, y el composable `Pantalla2` recibe una lambda que navega a la pantalla `Pantalla1`.

```
@Composable
fun Pantalla1(onNavigate: () -> Unit) {
    Button(onClick = onNavigate) {
        Text(text = "Ir a Pantalla 2")
    }
}
@Composable
fun Pantalla2(onNavigate: () -> Unit) {
    Button(onClick = onNavigate) {
        Text(text = "Ir a Pantalla 1")
    }
}
```

En los componibles `Pantalla1` y `Pantalla2`, se define un botón que, al hacer clic en él, ejecuta la lambda recibida como argumento, que navega a la pantalla correspondiente.

Para la definición de los nombres de las pantallas

Es importante definir nombres significativos para las pantallas de tu aplicación. Los nombres de las pantallas deben reflejar claramente el propósito y la funcionalidad de cada pantalla.

Por ejemplo, si tienes una pantalla de inicio y una pantalla de perfil, puedes nombrarlas `pantallaInicio` y `pantallaPerfil`, respectivamente.

```
NavHost(navController = navController, startDestination = "pantallaInicio") {
    composable("pantallaInicio") {
        PantallaInicio()
    }
    composable("pantallaPerfil") {
        PantallaPerfil()
    }
}
```

Estos nombres pueden definirse en un archivo de constantes o en un objeto de compañero para mantener un código más organizado y legible. Existen varias opciones para definir los nombres de las pantallas, como objetos de compañero, constantes o enumeraciones.

- Objeto anónimo
- Constantes
- Enumeraciones
- Clases selladas

```
object Rutas {
    const val PANTALLA_INICIO = "pantallaInicio"
    const val PANTALLA_PERFIL = "pantallaPerfil"
}
```

En el ejemplo anterior, se definen los nombres de las pantallas `PANTALLA_INICIO` y `PANTALLA_PERFIL` utilizando un objeto anónimo, constantes, enumeraciones y clases selladas.

Recursos

•Documentación oficial de Jetpack Compose Navigation: La documentación oficial de Jetpack Compose Navigation, que incluye guías, tutoriales y ejemplos para aprender a usar la navegación en Compose.

•Navegación y arquitectura de la app (MVVM) - Codelabs

•Ejercicio navegación - Enunciado - Solución

Material Design en Jetpack Compose

Material Design es un sistema de diseño desarrollado por Google que ayuda a los desarrolladores a crear interfaces de usuario atractivas y coherentes en aplicaciones Android. Con Jetpack Compose, puedes implementar los principios de Material Design de forma sencilla y eficiente.

Temas y estilos

Jetpack Compose incluye un conjunto de widgets y estilos basados en Material Design que puedes utilizar para crear interfaces de usuario modernas y atractivas. Puedes personalizar los colores, tipografías y formas de tus componentes para adaptarlos a la identidad visual de tu aplicación.

```
@Composable
fun MiApp() {
    MaterialTheme {
        Column {
            Text(text = "Hola, mundo!", style = MaterialTheme.typography.title-
large)
            Button(onClick = { /* Acción */ }) {
                Text(text = "Haz clic aquí")
            }
        }
    }
}
```

En el ejemplo anterior, se define un componente `MiApp` que utiliza `MaterialTheme` para aplicar los estilos de Material Design a los componentes de la interfaz de usuario. Se utiliza `MaterialTheme.typography.title-large` para aplicar un estilo de tipografía grande al texto y se utiliza `Button` para mostrar un botón con el texto "Haz clic aquí".

Widgets de Material Design

Jetpack Compose incluye una amplia variedad de widgets de Material Design que puedes utilizar para crear interfaces de usuario complejas y atractivas. Algunos de los widgets más comunes son:

- `Button`: Un botón interactivo que el usuario puede pulsar para realizar una acción.
- `TextField`: Un campo de texto que el usuario puede utilizar para introducir texto.
- `Checkbox`: Una casilla de verificación que el usuario puede marcar o desmarcar.
- `RadioButton`: Un botón de opción que el usuario puede seleccionar entre varias opciones.
- `Switch`: Un interruptor que el usuario puede activar o desactivar.

Puedes utilizar estos widgets y muchos más para crear interfaces de usuario modernas y atractivas en tus aplicaciones Android con Jetpack Compose.

Personalización de estilos

Puedes personalizar los estilos de Material Design en Jetpack Compose para adaptarlos a la identidad visual de tu aplicación. Puedes definir tus propios temas y estilos utilizando el componente `MaterialTheme` y los objetos `Typography`, `Colors` y `Shapes`.

```
val MiTema = lightColors(
    primary = Color(0xFF6200EE),
    primaryVariant = Color(0xFF3700B3),
    secondary = Color(0xFF03DAC6)
)
@Composable
fun MiApp() {
    MaterialTheme(colors = MiTema) {
        Column {
            Text(text = "Hola, mundo!", style = MaterialTheme.typography.title-
large)
            Button(onClick = { /* Acción */ }) {
                Text(text = "Haz clic aquí")
            }
        }
    }
}
```

En el ejemplo anterior, se define un tema personalizado `MiTema` con colores personalizados y se utiliza `MaterialTheme` para aplicar este tema a los componentes de la interfaz de usuario. Se utiliza `MaterialTheme.typography.title-large` para aplicar un estilo de tipografía grande al texto y se utiliza `Button` para mostrar un botón con el texto "Haz clic aquí".

Recursos

- Documentación oficial de Material Design: La documentación oficial de Material Design, que incluye guías, tutoriales y ejemplos para aprender a diseñar interfaces de usuario con Material Design.
- Documentación oficial de Jetpack Compose: La documentación oficial de Jetpack Compose, que incluye guías, tutoriales y ejemplos para aprender a crear interfaces de usuario con Compose.

Ciclo de vida de una aplicación Android

El ciclo de vida de una aplicación Android es el conjunto de estados por los que pasa una aplicación desde que se inicia hasta que se detiene. Comprender el ciclo de vida de una aplicación es fundamental para desarrollar aplicaciones robustas y eficientes en Android.

Componentes del ciclo de vida

En Android, los componentes del ciclo de vida de una aplicación son actividades, fragmentos, servicios y receptores de difusión. Cada uno de estos componentes tiene su propio ciclo de vida y se comporta de manera diferente en función de los eventos que se producen en la aplicación.

Vamos a centrarnos en las Actividades o `Activity`, que son los componentes principales de una aplicación Android y representan una única pantalla con la que el usuario puede interactuar.

Estados de una actividad

Las actividades en Android pueden estar en uno de los siguientes estados:

- **Created**: La actividad ha sido creada pero aún no es visible para el usuario.
- **Started**: La actividad es visible para el usuario pero no tiene el foco.
- **Resumed**: La actividad está en primer plano y tiene el foco.
- **Paused**: La actividad ha perdido el foco pero aún es visible para el usuario.
- **Stopped**: La actividad ya no es visible para el usuario.
- **Destroyed**: La actividad ha sido destruida y liberada de la memoria.

Métodos del ciclo de vida

En Android, cada actividad tiene una serie de métodos que se invocan en función de su estado en el ciclo de vida. Algunos de los métodos más comunes son:

- `onCreate()`: Se llama cuando la actividad se crea por primera vez.
- `onStart()`: Se llama cuando la actividad se hace visible para el usuario.
- `onResume()`: Se llama cuando la actividad obtiene el foco y se convierte en activa.
- `onPause()`: Se llama cuando la actividad pierde el foco pero aún es visible.
- `onStop()`: Se llama cuando la actividad ya no es visible para el usuario.
- `onDestroy()`: Se llama cuando la actividad es destruida y liberada de la memoria.

Ejemplo de ciclo de vida de una actividad

A continuación, se muestra un ejemplo del ciclo de vida de una actividad en Android:

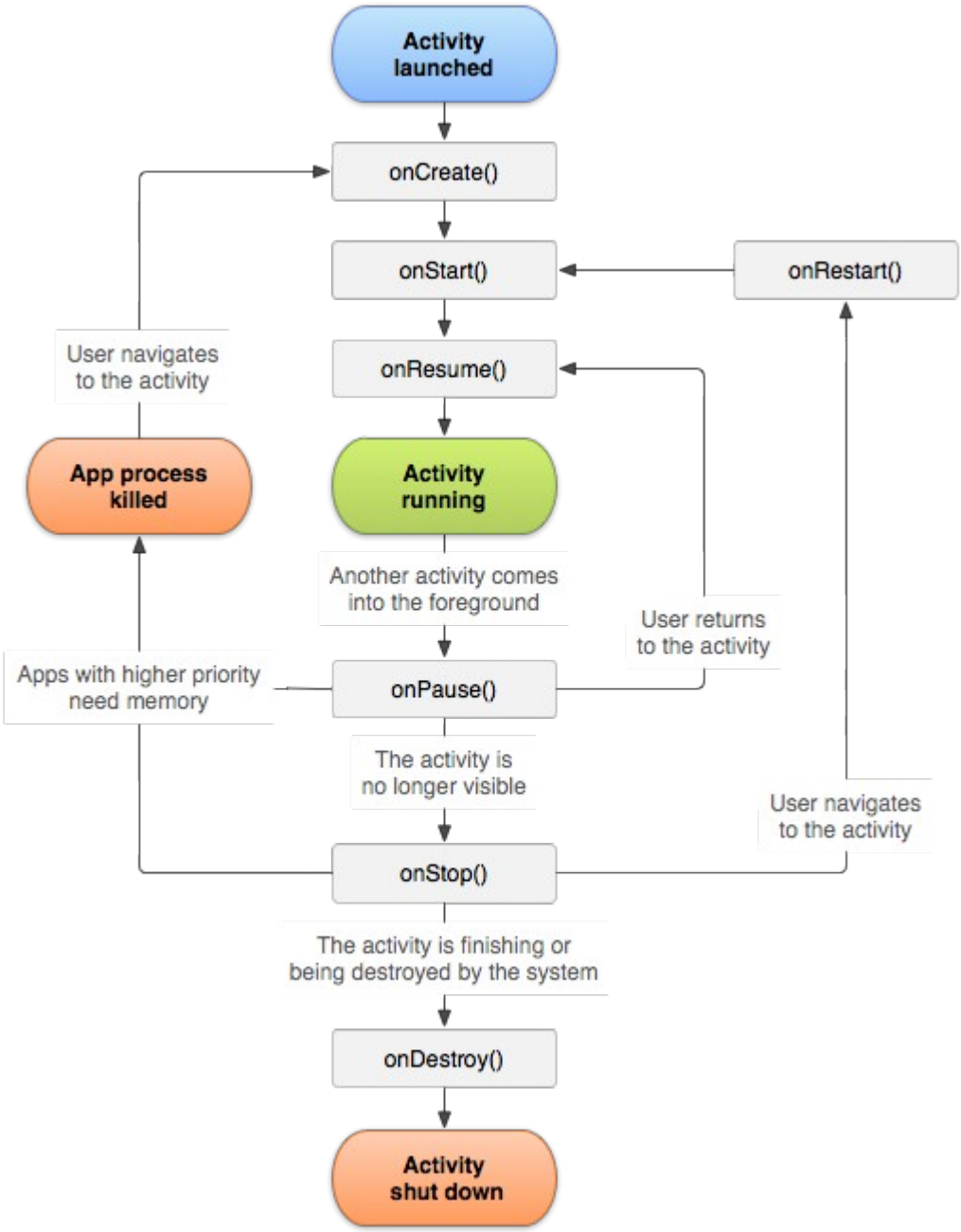
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        Log.d("MainActivity", "onCreate")
    }
    override fun onStart() {
```

```
        super.onStart()
        Log.d("MainActivity", "onStart")
    }
    override fun onResume() {
        super.onResume()
        Log.d("MainActivity", "onResume")
    }
    override fun onPause() {
        super.onPause()
        Log.d("MainActivity", "onPause")
    }
    override fun onStop() {
        super.onStop()
        Log.d("MainActivity", "onStop")
    }
    override fun onDestroy() {
        super.onDestroy()
        Log.d("MainActivity", "onDestroy")
    }
}
```

En el ejemplo anterior, se define una actividad `MainActivity` con los métodos del ciclo de vida más comunes. Cada método imprime un mensaje de registro en la consola para indicar en qué estado se encuentra la actividad.

Imagen del ciclo de vida de una actividad

A continuación, se muestra una imagen que representa el ciclo de vida de una actividad en Android:



Cómo podemos observar, el ciclo de vida de una actividad en Android consta de varios estados y transiciones entre ellos.

Es importante comprender estos estados para gestionar correctamente el ciclo de vida de una aplicación Android.

Sobre el contexto o `Context`

El contexto o `Context` es un objeto que proporciona información sobre el entorno en el que se ejecuta una aplicación Android.

El contexto se utiliza para acceder a recursos de la aplicación, iniciar actividades, crear servicios y mucho más.

En Android, el contexto se puede obtener a través de la clase `Context` o de una subclase de `Context`, como `Activity` o `Service`.

Obtener el contexto en Android

En Android, puedes obtener el contexto de varias formas, dependiendo de la clase en la que te encuentres. Algunas de las formas más comunes de obtener el contexto son:

- `this`: En una actividad, puedes utilizar `this` para hacer referencia al contexto de la actividad.
- `getApplicationContext()`: En una actividad o un servicio, puedes utilizar `getApplicationContext()` para obtener el contexto de la aplicación.
- `getContext()`: En una vista, puedes utilizar `getContext()` para obtener el contexto de la vista.

CONTEXTO EN COMPOSE

Para obtener el contexto en un entorno de Compose, puedes utilizar el modificador `LocalContext.current`.

Usos comunes del contexto

Algunos de los usos comunes del contexto en Android son:

- Acceder a recursos de la aplicación, como cadenas, imágenes y estilos.
- Iniciar actividades y servicios.
- Crear notificaciones y alarmas.
- Acceder a la base de datos de la aplicación.

El contexto es fundamental para el desarrollo de aplicaciones Android y te permite interactuar con el sistema operativo y los servicios de la plataforma.

Toasts en Android

Los toasts son mensajes breves que se muestran en la pantalla durante un corto período de tiempo. En Android, puedes utilizar la clase `Toast` para mostrar toasts en tu aplicación.

A continuación, se muestra un ejemplo de cómo mostrar un toast en Android:

```
Toast.makeText(this, "Hola, mundo!", Toast.LENGTH_SHORT).show()
```

En el ejemplo anterior, se utiliza `Toast.makeText()` para crear un toast con el mensaje "Hola, mundo!" y `Toast.LENGTH_SHORT` para especificar la duración del toast.

Los toasts son útiles para mostrar mensajes informativos o de confirmación al usuario de forma rápida y sencilla.

Snackbar en Android

Los snackbar son mensajes breves que se muestran en la parte inferior de la pantalla y permiten al usuario realizar una acción o deshacer una acción.

En Android, puedes utilizar la clase `Snackbar` de la biblioteca de diseño de materiales para mostrar snackbar en tu aplicación.

A continuación, se muestra un ejemplo de cómo mostrar un snackbar en Android:

```
Snackbar.make(view, "Mensaje de snackbar", Snackbar.LENGTH_SHORT)
    .setAction("Deshacer") {
        // Acción al deshacer
    }
    .show()
```

En el ejemplo anterior, se utiliza `Snackbar.make()` para crear un snackbar con el mensaje "Mensaje de snackbar" y `Snackbar.LENGTH_SHORT` para especificar la duración del snackbar. Se utiliza `setAction()` para añadir un botón de deshacer al snackbar y `show()` para mostrar el snackbar en la pantalla.

Los snackbar son útiles para mostrar mensajes informativos o de confirmación al usuario y permitirle realizar acciones adicionales.

Diálogos en Android

Los diálogos son ventanas emergentes que se utilizan para mostrar información, solicitar confirmación al usuario o realizar una acción.

En Android, puedes utilizar la clase `AlertDialog` para crear diálogos en tu aplicación.

A continuación, se muestra un ejemplo de cómo mostrar un diálogo en Android:

```
val builder = AlertDialog.Builder(this)
builder.setTitle("Título del diálogo")
builder.setMessage("Mensaje del diálogo")
builder.setPositiveButton("Aceptar") { dialog, which ->
    // Acción al hacer clic en Aceptar
}
builder.setNegativeButton("Cancelar") { dialog, which ->
    // Acción al hacer clic en Cancelar
}
builder.show()
```

En compose, puedes utilizar el componente `AlertDialog` para crear diálogos de forma declarativa.

Los diálogos son útiles para interactuar con el usuario y solicitar su entrada o confirmación de forma clara y concisa.

Arquitectura en Android

En Android, la arquitectura de una aplicación se refiere a la estructura y organización de sus componentes, como las actividades, fragmentos, servicios y otros elementos que la componen. Una buena arquitectura es fundamental para crear aplicaciones robustas, escalables y fáciles de mantener.

En este documento, veremos los principios básicos de la arquitectura en Android y cómo puedes aplicarlos en tus propias aplicaciones. También veremos algunas de las arquitecturas más comunes en Android, como MVC, MVP, MVVM y Clean Architecture, y cómo puedes elegir la mejor arquitectura para tu proyecto.

Principios básicos de la arquitectura en Android

Al diseñar la arquitectura de una aplicación Android, es importante tener en cuenta los siguientes principios básicos:

- **Separación de responsabilidades**: Divide tu aplicación en capas o componentes que tengan responsabilidades claras y bien definidas. Por ejemplo, separa la lógica de presentación de la lógica de negocio y de acceso a datos.
- **Escalabilidad**: Diseña tu arquitectura de manera que puedas agregar nuevas funcionalidades o componentes sin tener que reescribir todo el código.
- **Mantenibilidad**: Haz que tu código sea fácil de entender, modificar y depurar. Utiliza patrones de diseño y buenas prácticas de programación para mantener tu código limpio y organizado.
- **Pruebas unitarias**: Diseña tu arquitectura de manera que puedas escribir pruebas unitarias para cada componente de tu aplicación. Las pruebas unitarias te permiten validar el comportamiento de tu código de forma automatizada y garantizar su calidad.
- **Reactividad**: Diseña tu arquitectura de manera que puedas responder de forma rápida y eficiente a los eventos y cambios en tu aplicación. Utiliza patrones de diseño reactivos para crear interfaces de usuario dinámicas y reactivas.

Arquitecturas comunes en Android

En Android, existen varias arquitecturas comunes que puedes utilizar para diseñar tu aplicación. Algunas de las arquitecturas más populares son las siguientes:

- **MVC (Modelo-Vista-Controlador)**: En el patrón MVC, la vista es responsable de mostrar la interfaz de usuario, el controlador es responsable de manejar las interacciones del usuario y el modelo es responsable de la lógica de negocio y acceso a datos.
- **MVP (Modelo-Vista-Presentador)**: En el patrón MVP, la vista es responsable de mostrar la interfaz de usuario, el presentador es responsable de manejar las interacciones del usuario y el modelo es responsable de la lógica de negocio y acceso a datos. El presentador actúa como un intermediario entre la vista y el modelo.

•**MVVM (Modelo-Vista-ViewModel)**: En el patrón MVVM, la vista es responsable de mostrar la interfaz de usuario, el ViewModel es responsable de manejar la lógica de presentación y el modelo es responsable de la lógica de negocio y acceso a datos. El ViewModel actúa como un intermediario entre la vista y el modelo.

•**Clean Architecture**: La Clean Architecture es una arquitectura en capas que separa la aplicación en capas de dominio, aplicación e infraestructura. Cada capa tiene responsabilidades claras y bien definidas, lo que facilita la escalabilidad y mantenibilidad de la aplicación.

Elección de la arquitectura adecuada

A la hora de elegir la arquitectura para tu aplicación Android, es importante tener en cuenta varios factores, como el tamaño y complejidad de la aplicación, el equipo de desarrollo, los requisitos de rendimiento y escalabilidad, y la experiencia previa con las arquitecturas disponibles.

Algunos consejos para elegir la arquitectura adecuada son los siguientes:

•**Evalúa tus necesidades**: Analiza los requisitos de tu aplicación y elige la arquitectura que mejor se adapte a ellos. Por ejemplo, si tu aplicación es pequeña y sencilla, puedes optar por una arquitectura MVC o MVP. Si tu aplicación es grande y compleja, puedes optar por una arquitectura MVVM o Clean Architecture.

•**Prueba diferentes arquitecturas**: Experimenta con diferentes arquitecturas y evalúa sus ventajas y desventajas en función de tus necesidades. No tengas miedo de probar nuevas arquitecturas y adaptarlas a tus necesidades específicas.

•**Consulta la comunidad**: Busca en la comunidad de desarrolladores de Android y consulta ejemplos, tutoriales y buenas prácticas sobre arquitectura. La comunidad de desarrolladores es una gran fuente de información y te puede ayudar a elegir la arquitectura adecuada para tu proyecto.

La arquitectura que vamos a utilizar en este curso es MVVM

En este curso, vamos a utilizar la arquitectura MVVM (Modelo-Vista-ViewModel) para diseñar nuestras aplicaciones Android. La arquitectura MVVM es una arquitectura moderna y escalable que separa la lógica de presentación de la lógica de negocio y acceso a datos.

En la arquitectura MVVM, la vista es responsable de mostrar la interfaz de usuario, el ViewModel es responsable de manejar la lógica de presentación y el modelo es responsable de la lógica de negocio y acceso a datos. El ViewModel actúa como un intermediario entre la vista y el modelo, lo que facilita la separación de responsabilidades y la reutilización de código.

La arquitectura MVVM es una arquitectura popular en Android y es compatible con las bibliotecas y herramientas de Jetpack, como LiveData, ViewModel y Room. Utilizaremos estas bibliotecas y herramientas en nuestro curso para crear aplicaciones Android modernas y eficientes.

Recursos

- Documentación oficial de Android: La documentación oficial de Android, que incluye guías, tutoriales y ejemplos para aprender a desarrollar aplicaciones Android.
- Navegación y arquitectura de la app (MVVM) - Codelabs
- Ejercicio Lista de la compra - Enunciado - Solución
- App de Carta Alta - Enunciado - Solución
- Examen diciembre - Enunciado - Solución

Capa de UI

En esta sección vamos a ver cómo implementar la capa de UI en una arquitectura MVVM (Model-View-ViewModel) en una aplicación Android con Jetpack Compose.

¿Qué es MVVM?

MVVM es un patrón de arquitectura de software que se utiliza para separar la lógica de presentación de la lógica de negocio en una aplicación. En MVVM, la capa de UI se divide en tres componentes principales:

- **Model**: Representa los datos y la lógica de negocio de la aplicación.
- **View**: Representa la interfaz de usuario de la aplicación.
- **ViewModel**: Actúa como un intermediario entre el Model y la View. Se encarga de manejar la lógica de presentación y de exponer los datos necesarios para que la View pueda mostrarlos.

MVVM es un patrón muy utilizado en el desarrollo de aplicaciones Android, ya que facilita la separación de responsabilidades y la reutilización de código.

Implementación de la capa de UI en MVVM

Para implementar la capa de UI en una arquitectura MVVM en una aplicación Android con Jetpack Compose, puedes seguir los siguientes pasos:

1. **Definir el Model**: Define las clases y estructuras de datos que representan los datos y la lógica de negocio de tu aplicación. Por ejemplo, puedes definir una clase `User` que represente un usuario de la aplicación.
2. **Definir el ViewModel**: Define una clase que extienda `ViewModel` y que contenga la lógica de presentación de tu aplicación. Por ejemplo, puedes definir un ViewModel que contenga la lógica para cargar los datos de un usuario.
3. **Definir la View**: Define la interfaz de usuario de tu aplicación utilizando Jetpack Compose. Por ejemplo, puedes definir una función componible que muestre los datos de un usuario en la pantalla.
4. **Conectar el ViewModel con la View**: Conecta el ViewModel con la View para que la View pueda mostrar los datos del ViewModel. Puedes utilizar `rememberViewModel` para crear una instancia del ViewModel en la View y `viewModel` para acceder a los datos del ViewModel.

Ejemplo de implementación

Recursos

- Documentación oficial de Jetpack Compose: La documentación oficial de Jetpack Compose, que incluye guías, tutoriales y ejemplos para aprender a usar Compose.

- Navegación y arquitectura de la app (MVVM) - Codelabs
- Ejercicio Lista de la compra - Enunciado - Solución
- App de Carta Alta - Enunciado - Solución
- Examen diciembre - Enunciado - Solución

Capa de datos en Android

En esta sección vamos a ver cómo implementar la capa de datos en una aplicación Android con Jetpack Compose y Room. La capa de datos es una parte fundamental de la arquitectura de una aplicación, ya que se encarga de gestionar el acceso a los datos y de proporcionar una interfaz para interactuar con ellos.

La capa de datos en una aplicación Android

La capa de datos en una aplicación Android se encarga de gestionar el acceso a los datos de la aplicación y de proporcionar una interfaz para interactuar con ellos. La capa de datos se divide en tres componentes principales:

- **Modelo de datos**: Representa los datos de la aplicación y define su estructura y comportamiento. Por ejemplo, puedes definir una clase `User` que represente un usuario de la aplicación.
- **Data Access Object (DAO)**: Define las operaciones de acceso a los datos, como la inserción, actualización y eliminación de datos. Por ejemplo, puedes definir un DAO que contenga métodos para insertar, actualizar y eliminar usuarios.
- **Base de datos**: Representa la base de datos de la aplicación y contiene la lógica para crear y acceder a las tablas de la base de datos. Por ejemplo, puedes definir una clase `AppDatabase` que extienda `RoomDatabase` y contenga métodos para acceder a las tablas de la base de datos.

La capa de datos es importante para garantizar la integridad y consistencia de los datos de la aplicación, así como para proporcionar una interfaz coherente y segura para interactuar con ellos.

Implementación de la capa de datos en Jetpack Compose

Para implementar la capa de datos en una aplicación Android con Jetpack Compose, puedes seguir los siguientes pasos:

1. **Definir el modelo de datos**: Define las clases y estructuras de datos que representan los datos de la aplicación. Por ejemplo, puedes definir una clase `User` que represente un usuario de la aplicación.
2. **Definir el DAO**: Define una interfaz que contenga las operaciones de acceso a los datos y anótala con `@Dao`. Por ejemplo, puedes definir un DAO que contenga métodos para insertar, actualizar y eliminar usuarios.
3. **Definir la base de datos**: Define una clase que extienda `RoomDatabase` y anótala con `@Database`. En esta clase, define los métodos para acceder a las tablas de la base de datos y crea una instancia de la base de datos.
4. **Conectar la capa de datos con la capa de repositorio**: Conecta la capa de datos con la capa de repositorio de tu aplicación para que puedas acceder a los datos de forma sencilla y

reactiva. Puedes utilizar `remember` para crear una instancia de la base de datos en la capa de datos y `viewModel` para acceder a los datos de la base de datos.

Ejemplo de implementación

Recursos

•Documentación oficial de Jetpack Compose: La documentación oficial de Jetpack Compose, que incluye guías, tutoriales y ejemplos para aprender a usar Compose.

Conexión a internet

Para conectarse a internet en una aplicación Android con Jetpack Compose, puedes utilizar las siguientes opciones:

•**Retrofit**: Retrofit es una biblioteca de cliente HTTP para Android y Java que facilita la conexión a servicios web RESTful. Puedes utilizar Retrofit para realizar peticiones HTTP a un servidor y obtener los datos necesarios para tu aplicación.

•**Ktor**: Ktor es un framework de cliente y servidor web en Kotlin que te permite crear aplicaciones web

•**Volley**: Volley es una biblioteca de red que facilita la conexión a servicios web en Android. Puedes utilizar Volley para realizar peticiones HTTP y gestionar las respuestas de forma sencilla.

Vamos a centrarnos en la primera opción, Retrofit, que es una de las bibliotecas más utilizadas para conectarse a servicios web en Android.

¿Qué es Retrofit?

Retrofit es una biblioteca de cliente HTTP para Android y Java que facilita la conexión a servicios web RESTful. Retrofit te permite definir una interfaz de servicio web con anotaciones que describen las operaciones disponibles en el servicio, como las peticiones GET, POST, PUT y DELETE. Retrofit se encarga de convertir las respuestas del servidor en objetos Java/Kotlin y de gestionar la comunicación con el servidor de forma eficiente.

Implementación de Retrofit en Jetpack Compose

Para implementar Retrofit en una aplicación Android con Jetpack Compose, puedes seguir los siguientes pasos:

1.**Definir la interfaz de servicio web**: Define una interfaz que contenga las operaciones disponibles en el servicio web y anótala con las anotaciones de Retrofit, como `@GET`, `@POST`, `@PUT` y `@DELETE`. Por ejemplo, puedes definir una interfaz `ApiService` que contenga métodos para obtener y enviar datos al servidor.

2.**Crear una instancia de Retrofit**: Crea una instancia de Retrofit utilizando el constructor de `Retrofit.Builder` y configura la URL base del servicio web y el convertidor de JSON. Por ejemplo, puedes crear una instancia de Retrofit que se conecte a un servidor en `https://api.example.com` y utilice el convertidor de JSON de Gson.

3.**Crear una instancia del servicio web**: Crea una instancia del servicio web a partir de la interfaz de servicio web y la instancia de Retrofit. Por ejemplo, puedes crear una instancia del servicio web a partir de la interfaz `ApiService` y la instancia de Retrofit.

4.**Realizar peticiones HTTP**: Utiliza la instancia del servicio web para realizar peticiones HTTP al servidor y obtener los datos necesarios para tu aplicación. Por ejemplo, puedes

utilizar el método `getUsers()` de la interfaz `ApiService` para obtener una lista de usuarios del servidor.

Ejemplo de implementación

Recursos

Persistencia de datos en Android

En esta sección vamos a ver cómo implementar la persistencia de datos en una aplicación Android con Jetpack Compose y Room. La persistencia de datos es una parte fundamental de la arquitectura de una aplicación, ya que permite almacenar y recuperar datos de forma segura y eficiente.

Para ello, vamos a utilizar Room, una biblioteca de persistencia de datos que forma parte de Jetpack y que facilita el acceso a la base de datos y la gestión de los datos de la aplicación. Veremos cómo definir entidades, DAOs y la base de datos en Room, y cómo conectar Room con la capa de datos de la aplicación para acceder a los datos de forma sencilla y reactiva.

También veremos como almacenar datos de preferencias con `DataStore` y como trabajar con archivos en Android.

¿Qué es Room?

Room es una biblioteca de persistencia de datos que forma parte de Jetpack, el conjunto de bibliotecas y herramientas recomendadas por Google para el desarrollo de aplicaciones Android. Room proporciona una capa de abstracción sobre SQLite, la base de datos relacional integrada en Android, y facilita el acceso a la base de datos y la gestión de los datos de la aplicación.

Room se compone de tres componentes principales:

- **Database**: Representa la base de datos de la aplicación y contiene la lógica para crear y acceder a las tablas de la base de datos.
- **Entity**: Representa una tabla de la base de datos y contiene la definición de las columnas y los tipos de datos de la tabla.
- **DAO (Data Access Object)**: Define las operaciones de acceso a la base de datos, como la inserción, actualización y eliminación de datos.

Implementación de Room en Jetpack Compose

Para implementar Room en una aplicación Android con Jetpack Compose, puedes seguir los siguientes pasos:

1. **Definir la entidad**: Define una clase que represente la tabla de la base de datos y anótala con `@Entity`. Por ejemplo, puedes definir una clase `User` que represente un usuario de la aplicación.
2. **Definir el DAO**: Define una interfaz que contenga las operaciones de acceso a la base de datos y anótala con `@Dao`. Por ejemplo, puedes definir un DAO que contenga métodos para insertar, actualizar y eliminar usuarios.
3. **Definir la base de datos**: Define una clase que extienda `RoomDatabase` y anótala con `@Database`. En esta clase, define los métodos para acceder a las tablas de la base de datos y crea una instancia de la base de datos.
4. **Conectar Room con la capa de datos**: Conecta Room con la capa de datos de tu aplicación para que puedas acceder a la base de datos y gestionar los datos de la aplicación.

Puedes utilizar `remember` para crear una instancia de la base de datos en la capa de datos y `viewModel` para acceder a los datos de la base de datos.

Ejemplo de implementación

Recursos

Enunciados para ejercicios de ejemplo (Android)

En esta página encontraréis una serie de enunciados para ejercicios de ejemplo que podéis realizar para practicar y aprender más sobre el desarrollo de aplicaciones Android con Kotlin y Jetpack Compose.

Índice

App de lista de la compra o to-do list

Elaborar una app sencilla de una sola pantalla en la que tendremos una disposición de un TextField y un botón para añadir elementos a la lista con el nombre que el usuario introduzca en el TextField.

La lista debe estar gestionada con un LazyColumn y debe incluirse un paquete data con un datasource con datos de ejemplo que la lista debe almacenar de primeras para poder probar la aplicación.

Hacer varias versiones de la app pasando por los siguientes pasos:

1. Gestionando el estado de forma sencilla, los elementos de la lista tan solo tendrán un nombre.
2. Añadir un botón en cada elemento para que al pulsarlo este se borre.
3. Añadir un CheckBox a cada elemento para marcar el item como realizado o no realizado. Este estado debe manejarse correctamente.
4. Elevar el estado de la app para manejarlo mediante un ViewModel y su correspondiente StateFlow.
5. Añadir una barra superior a la app a través de un Scafold, en la barra superior debe aparecer el título de la app y un botón para elminar todos los elementos de la lista marcados como realizados de golpe.

Solución en Github

App de contadores

Elaborar una app sencilla de una sola pantalla en la que manejaremos el estado de uno o varios contadores.

Debemos elaborar varios ejemplos de la misma app pasando por los siguientes:

1.Una app con un único contador que se muestre en pantalla y al que se le añada 1 cada vez que se pulse un botón. Debe existir también un botón para reiniciar el contador a 0.
2.En esta versión tendremos dos contadores y un contador general, de forma que los dos primeros funcionarán de forma independiente y el general aumentará cada vez que cualquiera de los otros aumente.
3.Modificar la versión anterior añadiendo un TextField que solo admita números enteros positivos por cada contador, para que si el usuario lo desea, se pueda modificar la cantidad que se aumenta con cada pulsación. Esto debe afectar al contador y al contador general.
4.Crear una pantalla principal con botones que mediante navegación te llevan a cada uno de los 3 ejemplos anteriores y configurarlos con una barra superior que incluya una flecha para volver a la pantalla principal.
5.Incluir una nueva versión en la que se eleva el estado de los contadores para manejarlo mediante un ViewModel y su correspondiente StateFlow.

Solución en Github

App de pruebas de navegación

Elaborar una app con 4 pantallas diferentes y gestionar su navegación tal y como hemos visto en ejemplos anteriores de forma que sea posible navegar así:

•Desde la pantalla principal se debe poder ir mediante un botón a la pantalla 2 y la 3.
•Desde la pantalla 2 se debe poder volver a la 1 y moverse a la 3.
•Desde la pantalla 3 se debe poder volver a la 1 o a la 2 dependiendo de cuál se venga. Se debe poder también ir a la 4 de forma simple o pasándole uno o dos parámetros textuales que se visualizarán en ella.
•Desde la pantalla 4 se debe poder volver a la 3 o a la 1 directamente y borrando la pila de pantallas. Además, si se reciben parámetros deben mostrarse, en caso contrario debe mostrarse un mensaje de que no se han recibido.

Solución en Github

App de calculadora

Elaborar la interfaz y el control del estado para una aplicación sencilla de pantalla única que cuente con una calculadora.

Se pueden añadir tantas posibles operaciones matemáticas como se quieran.

•Operaciones básicas: suma, resta, multiplicación, división, división entera, resto de la división entera.

•Otras operaciones: potencia, logaritmo neperiano, logaritmo para base elegida, raíz cuadrada, raíz para cualquier radical.

•Operaciones trigonométricas: seno, coseno, tangente, arcoseno, arcocoseno, arcotangente.

Y todas las que se os ocurran y queráis implementar.

App de TicTacToe

Elaborar una app de una sola pantalla en la que se pueda jugar por turnos al tres en raya. Debe controlarse el estado mediante ViewModel y StateFlow.

Se pueden utilizar elementos clickables para poner las X o las O y se pueden usar iconos o imágenes para mostrarlas, es indiferente.

Se debe mostrar en cada momento si es el turno del jugador X o de las O. Se debe mostrar al finalizar el juego quién ha ganado o si se ha empatado y dar una opción para volver a jugar. Esto debe hacerse en un cuadro de diálogo.

Este ejemplo puede basarse en siguiente Codelab cuyo código final está disponible en este repositorio.

App de Trivial

Basándonos en el mismo ejemplo que el ejercicio anterior:

Este ejemplo puede basarse en siguiente Codelab cuyo código final está disponible en este repositorio.

Elaborar un pequeño juego de Trivial, las preguntas deben estar contenidas en un datasource en el paquete data, se debe crear un modelo de datos acorde para almacenar cada pregunta.

La aplicación debe tener una pantalla principal dónde se muestre la puntuación máxima en porcentaje de aciertos (0% por defecto al iniciar) y se deje al usuario elegir la cantidad de pregunas que quiere responder (mínimo de 5 y máximo de 20).

Una vez elegida la cantidad de preguntas debe comenzar el juego en una pantalla secundaria, a partir de esta pantalla se irán respondiendo preguntas hasta llegar a las establecidas y se calculará el porcentaje de acierto que, si es superior al record se debe actualizar.

Se pueden elaborar variantes de esta aplicación añadiendo cosas como tiempos límite de respuesta, cambiando a un número de preguntas fijo pero con un máximo de errores permitido o cualquier variante que e os ocurra.

App genérica de un modelo de datos a elegir

Elaborar una app con navegación y control del estado mediante ViewModels y StateFlow. Se debe elaborar en base a un modelo de datos sencillo elegido por cada uno/a.

La app debe de contar con una pantalla principal que contenga una barra superior con el título de la app y en su MainContent debe haber una lista de elementos visuales basados en el modelo de datos elegido.

Cada elemento de la lista debe poder eliminarse.

Debe crearse una segunda pantalla con un formulario desde el cuál poder añadir nuevos elementos a la lista.

Debe ser posible también editar los elementos existentes, dependiendo del modelo de datos tendrá más o menos sentido poder editar uno o varios campos del mismo. Se puede reutilizar la pantalla de añadir para editar usando navegación y dependiendo de si esta pantalla recibe o no un /id.

Deben precargarse datos en la lista desde un datasource contenido en el paquete data del proyecto.

A este proyecto se le pueden añadir tantas features como se quieran, desde búsqueda de elementos, hasta ordenación en base a diferentes propiedades del modelo de datos elegido u otras como selección de varios elementos para su eliminación o cambios de estado en grupo.

También se pueden añadir otras pantallas para gestionar datos secundarios o preferencias.

App generadora de alarmas

Generar una app en la que podamos añadir una alarma o aviso para una fecha y hora determinada, la app debe mostrar en la pantalla cuánto falta para que llegue la alarma.

Esta app puede ampliarse con niveles de importancia o tematizándola.

Puede ampliarse también para configurar notificaciones push cuándo suene una alarma o aviso.

Documentación y recursos externos

Páginas de documentación interesantes

- Página oficial de Android Developers
- Listas y cuadrículas con Compose
- Cuadrículas diferidas (LazyGrids)
- Guía de arquitectura de apps
- Descripción general de ViewModel
- ViewModel Scope
- La capa de datos
- Corrutinas en Kotlin
- Corrutinas y dispatchers
- Página oficial de Retrofit
- Bases de datos con Room
- Cómo guardar datos en una base de datos local usando Room
- Cómo definir datos con entidades de Room
- Cómo acceder a los datos con DAO de Room
- La clase Database de Room
- Depurar la DB con el inspector de bases de datos de Android Studio
- Documentación general de Room
- Cómo probar los Flows de Kotlin en Android
- Inyección de dependencias en Android
- Más Codelabs de interés
- Ciclo de vida de una actividad
- Datos de preferencias con DataStore - Codelab

Recursos generales externos

- Kotlin y Java en una misma app - fork de avidaldo
- Vistazo rápido al ciclo de vida de una actividad
- Refactoring GURU - refactorización, patrones de diseño, principios SOLID
- develou.com
- Repositorio de ejemplos de Compose
- Guía de Kotlin
- Antonio Leiva en Youtube
- Mouredev en Youtube
- Aristidev en Youtube
- DevKiper en Youtube
- Video sobre Flows en Kotlin