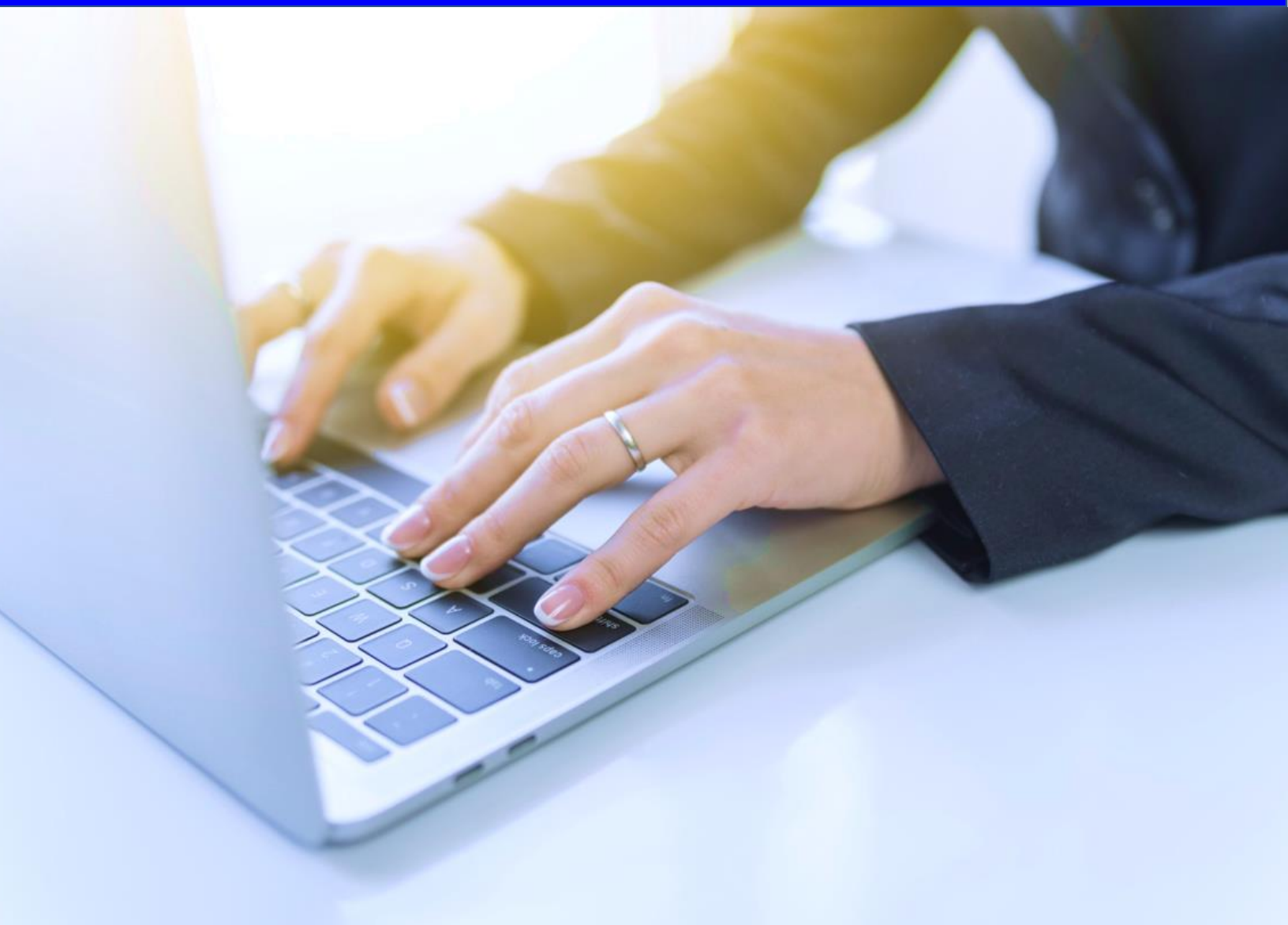
Nombre de la Asignatura:		**PROGRAMACIÓN MOBILE**
Unidad	**Nº I**	**FUNDAMENTOS DE ANDROID**



Consideraciones Previas

El contenido que se expone a continuación está ligado a los siguientes objetivos:

- *Comprender la sintaxis y estructura básica de un programa Kotlin.*
- Comprobar algunos conceptos básicos del lenguaje de programación Kotlin
- *Reconocer y utilizar correctamente los diferentes tipos de datos y variables en Kotlin.*
- *Identificar y aplicar los conceptos de operadores, expresiones y sentencias en Kotlin.*
- *Demostrar comprensión de las estructuras de control, como condicionales y bucles, en Kotlin.*
- *Comprender los fundamentos de Kotlin (conceptos de funciones y parámetros, concepto de orientación a objetos incluyendo clases, objeto, herencia y polimorfismo*
- *Ejecutar programas escritos en Kotlin en Android Studio u otra alternativa*
- *Comprender cómo Java y Kotlin pueden usarse en conjunto para desarrollar aplicaciones Android:*
- *Identificar las similitudes y diferencias entre Java y Kotlin.*
- *Comprender cómo funcionan las vistas XML*
- *Comprender cómo funciona Jetpack Compose:*
- *Comparar las diferencias entre Vistas XML y Jetpack Compose*
- *Codificar la navegación entre las distintas Actividades de la aplicación*
- Utilizar Jetpack Compose y Vistas XML en un mismo proyecto

Introducción

Durante el material previo, se abordaron diversos aspectos como la historia de Android, las herramientas necesarias para el desarrollo, los fundamentos básicos de Android y se exploró un ejemplo de una pequeña aplicación utilizando Jetpack Compose. En esta etapa, se continuará profundizando en Kotlin para comprender de manera integral los ejemplos de aplicaciones desarrolladas para Android. Se parte del entendimiento de que los lectores de este material ya poseen conocimientos en el lenguaje de programación Java, por lo que se realizarán comparaciones que agilicen el aprendizaje de Kotlin.

Es esencial adquirir los conocimientos fundamentales de Kotlin, como en la mayoría de los lenguajes de programación esto comprende entender: las herramientas de ejecución, la sintaxis, los tipos de datos, las estructuras condicionales, las estructuras de bucles, los operadores, la escritura de clases, así como los conceptos básicos de herencia y polimorfismo, entre otros aspectos relevantes.

Cabe destacar que este material no tiene como objetivo ser una guía exhaustiva, sino más bien enfocarse en lo fundamental. Por lo tanto, es importante aprender a utilizar la documentación oficial como una fuente de referencia para aclarar dudas y profundizar en el conocimiento de Kotlin. Si se desea explorar más a fondo el lenguaje Kotlin, las APIs y otras cuestiones relacionadas con Android, se recomendarán enlaces oficiales de manera oportuna.

Finalmente, se realizará una breve revisión de Jetpack Compose y las Vistas XML para la generación de pantallas en Android. Es importante comprender que Jetpack Compose funciona exclusivamente con Kotlin, mientras que las vistas XML permiten trabajar tanto con Kotlin como con Java. La forma tradicional de trabajar con Android ha sido a través de vistas XML y, como resultado, existe una amplia documentación en internet, libros, entre otros recursos, sobre esta metodología.

Ideas fuerza

Kotlin se ha convertido en el lenguaje de programación preferido para el desarrollo de aplicaciones en Android. Posee una sintaxis moderna, compatibilidad total con Java, características que mejoran la productividad y la calidad del código, interoperabilidad con bibliotecas existentes y una amplia comunidad que sigue creciendo. Adoptar Kotlin es esencial para aprovechar todas las ventajas del desarrollo nativo para Android.

Al aprender un nuevo lenguaje, en este caso Kotlin, es de suma importancia poder hacer pruebas en pequeño, para conseguir un entendimiento íntegro de lo que se escribe o se lee. En este sentido conocer una amplia gama de opciones de ejecución nos facilita la práctica, la resolución de dudas y aprendizaje de un lenguaje de programación. Con Kotlin tenemos disponible un Playground en línea, Intellij IDEA, CLI y el mismo Android Studio.

Los fundamentos de Kotlin, al igual que otros lenguajes de programación, se basan en conceptos clave como variables, funciones, estructuras condicionales y de bucle, clases, herencia, tipos de datos y anotaciones. Estos componentes mínimos son esenciales para construir software sólido y eficiente. Al dominar estos fundamentos, los desarrolladores adquieren las bases necesarias para comprender y aplicar conceptos más avanzados, y tienen la capacidad de crear aplicaciones robustas y escalables en Kotlin.

Jetpack Compose es una biblioteca moderna para la creación de interfaces de usuario para Android. Está diseñada exclusivamente para Kotlin, lo que permite a los desarrolladores aprovechar al máximo las características del lenguaje y su integración nativa con las APIs de Android.

El desarrollo de interfaces gráficas en Android usando Vistas XML ha sido la forma tradicional de desarrollo durante largos años, algunas de las ventajas de trabajar con Vistas XML son: la posibilidad de trabajar con Java o Kotlin, y que existe una amplia documentación a través de internet, libros, y otros formatos.

Los Intents en Android proporcionan un mecanismo versátil y poderoso para navegar entre Actividades, permitiendo que las aplicaciones se comuniquen y transfieran datos de manera eficiente, brindando a los usuarios una experiencia fluida y coherente en la navegación entre pantallas

Índice

Desarrollo

Introducción a Kotlin

Kotlin es un lenguaje moderno y conciso que se ha vuelto muy popular en la comunidad de desarrollo de Android debido a su sintaxis clara y su capacidad para eliminar el código redundante. Fue diseñado para ser compatible con Java, lo que significa que puedes utilizarlo junto con tu código Java existente (Jetbrains, 2023).

El desarrollo de aplicaciones móviles en Android ha sido orientado hacia Kotlin desde la conferencia de Google I/O en 2019 (Jetbrains, 2023).

Al utilizar Kotlin para el desarrollo de Android, puedes obtener los siguientes beneficios (Jetbrains, 2023):

- **Menos código con una mayor legibilidad**. Dedica menos tiempo a escribir tu código y a entender el código de los demás.
- **Menos errores comunes**. Las aplicaciones desarrolladas con Kotlin tienen un 20% menos de probabilidades de sufrir fallos según los datos internos de Google.
- **Compatibilidad de Kotlin con las bibliotecas Jetpack**. Jetpack Compose es la herramienta moderna recomendada por Android para crear interfaces de usuario nativas en Kotlin. Las extensiones KTX añaden características del lenguaje Kotlin, como corrutinas, funciones de extensión, lambdas y parámetros con nombre, a las bibliotecas de Android existentes.
- **Compatibilidad con el desarrollo multiplataforma**. Kotlin Multiplatform permite el desarrollo no solo en Android, sino también en aplicaciones iOS, backend y web. Algunas bibliotecas de Jetpack ya son multiplataforma. Compose Multiplatform, el marco de interfaz de usuario declarativo de JetBrains basado en Kotlin y Jetpack Compose, permite compartir interfaces de usuario en múltiples plataformas, como iOS, Android, escritorio y web.
- **Lenguaje y entorno maduros**. Desde su creación en 2011, Kotlin ha evolucionado constantemente, no solo como lenguaje, sino como un ecosistema completo con herramientas sólidas. Ahora está integrado de forma transparente en Android Studio y es utilizado activamente por muchas empresas para desarrollar aplicaciones Android.
- **Interoperabilidad con Java**. Puedes utilizar Kotlin junto con el lenguaje de programación Java en tus aplicaciones sin necesidad de migrar todo tu código a Kotlin.
- **Fácil aprendizaje**. Kotlin es muy fácil de aprender, especialmente para los desarrolladores de Java.

¿Cuáles fueron los factores clave que impulsaron la decisión de convertir a Kotlin en el lenguaje oficial para el desarrollo de aplicaciones Android, reemplazando a Java?

Cómo ejecutar código Kotlin

Cuando se comienza el aprendizaje de un lenguaje, es bueno conocer distintas alternativas para su ejecución y pruebas en pequeño. En este punto podemos deducir que Android Studio permite la ejecución de código Kotlin, pero el contexto de un proyecto de Android no es el mejor ambiente para partir el aprendizaje. Por la razón anterior, se documentan varias opciones disponibles para la ejecución de código Kotlin:

- **Usando el Playground de Kotlin**: Puedes utilizar el Playground de Kotlin, una herramienta en línea que te permite escribir y probar código Kotlin de manera rápida y sencilla en tu navegador web. Puedes acceder al Playground en el siguiente enlace: https://play.kotlinlang.org/ . Es una excelente opción para probar ideas rápidamente sin necesidad de configurar un entorno de desarrollo completo.
- **Utilizando IntelliJ IDEA**: IntelliJ IDEA es un entorno de desarrollo integrado (IDE) muy popular para Kotlin (y otros lenguajes de programación). Proporciona una excelente experiencia de desarrollo para Kotlin y te permite ejecutar y depurar tu código Kotlin de manera eficiente. Puedes descargar IntelliJ IDEA desde el siguiente enlace: https://www.jetbrains.com/idea/
- **Usando Android Studio**: Dentro de Android Studio puedes ejecutar código Kotlin a través de algún proyecto o practicar ingresando a Tools / Kotlin / Kotlin REPL.
- **Descargando e instalando el compilador de Kotlin**: Si prefieres una configuración más personalizada, puedes descargar el compilador de Kotlin directamente desde el repositorio oficial en GitHub. En la página de versiones de Kotlin en GitHub, puedes encontrar los binarios del compilador para diferentes plataformas y sistemas operativos. Una vez que hayas descargado e instalado el compilador, podrás ejecutar código Kotlin directamente desde la línea de comandos o mediante scripts. Puedes acceder al repositorio de Kotlin en GitHub en el siguiente enlace: https://github.com/JetBrains/kotlin/releases

Estas son algunas opciones populares para ejecutar código en Kotlin, antes de continuar trabajando con Android es importante comprender los fundamentos del lenguaje, por lo que es relevante que escoja una herramienta para probar código de manera rápida y sencilla. En la plataforma se cuenta con un video que profundiza más sobre el uso de estas herramientas.

Kotlin Básico

La comprensión de los elementos fundamentales en Kotlin es esencial para trabajar con él, al igual que sucede en la mayoría de los lenguajes de programación. Es importante familiarizarse con la sintaxis básica, como la forma de comentar el código, declarar variables, conocer los tipos de datos disponibles, así como aprender a escribir clases y funciones. Estos conocimientos básicos son primordiales para poder escribir y comprender programas básicos en el contexto del desarrollo para Android.

Cabe destacar que este material tiene como objetivo resaltar los aspectos más importantes de Kotlin para comenzar en el desarrollo para Android, sin pretender ser una guía exhaustiva. En este sentido, se recomienda siempre tener la capacidad de leer y buscar en la documentación oficial de Kotlin (https://kotlinlang.org/docs/home.html) y en la documentación oficial de Android (https://developer.android.com/docs). La consulta de estas fuentes confiables y actualizadas es fundamental para obtener información detallada y precisa sobre el lenguaje Kotlin y su integración con Android.

Kotlin y Java comparten muchas similitudes, lo que facilita el proceso de aprendizaje y migración de Java a Kotlin. Sin embargo, para un aprendizaje ágil y efectivo de Kotlin, es importante tener en cuenta las diferencias clave entre ambos lenguajes. Una de las diferencias más destacadas es que, a diferencia de Java, Kotlin no requiere el uso de punto y coma (;) al final de cada declaración.

Comentarios

Los comentarios se utilizan para agregar información descriptiva al código sin afectar su ejecución. Los comentarios son especialmente útiles para hacer que el código sea más comprensible para otros desarrolladores y para proporcionar aclaraciones sobre el propósito de ciertas secciones o líneas de código. En Kotlin al igual que en Java se disponen de comentario en línea y multilínea (Jetbrains, 2023).

```
// Este es un comentario en línea

/* Este es un comentario en bloque
   de múltiples líneas. */
```

Código 1 Ejemplo de comentario en línea y multilínea en Kotlin
Fuente: Elaboración propia

Variables

En Kotlin, las variables son utilizadas para almacenar y manipular datos durante la ejecución de un programa. Permiten guardar valores de diferentes tipos, como números, texto u objetos. En Kotlin, existen dos tipos de variables: las mutables, declaradas con la palabra clave "var", y las no mutables, declaradas con la palabra clave "val" (Jetbrains, 2023).

Las variables mutables (var) pueden cambiar su valor a lo largo del programa, lo que significa que se les puede asignar un nuevo valor en cualquier momento. Por otro lado, las variables no mutables (val) se utilizan para almacenar valores que no cambiarán una vez asignados, es decir, son inmutables.

La distinción entre variables mutables y no mutables en Kotlin es importante, ya que promueve la inmutabilidad cuando sea posible, lo que puede llevar a un código más seguro y menos propenso a errores. Al utilizar variables no mutables (val), se garantiza que su valor no cambiará accidentalmente, lo que facilita la comprensión del código y el razonamiento sobre su comportamiento.

```
// VAL funciona como una constante
val a = "hola"
a = "hola2"
// error: val cannot be reassigned

// VAR es mutable
var b = "hola"
b = "hola2" // funciona
```

Código 2 Uso de val y var en Kotlin
Fuente: Elaboración propia

Tipos de Datos

En Kotlin, se pueden encontrar tipos de datos básicos similares a los de Java, como **Int, Double, Boolean y String**. Sin embargo, Kotlin introduce algunas mejoras significativas. Por ejemplo, se puede declarar una variable como **nullable** utilizando el **operador "?"**, lo que permite expresar explícitamente la posibilidad de que una variable pueda contener un valor nulo. Además, Kotlin ofrece tipos de datos nulos seguros, lo que ayuda a prevenir los errores comunes de **NullPointerException** que a menudo se encuentran en Java (Jetbrains, 2023).

```
var mensaje1:String? = null // permite nulos
var mensaje2:String = null // error: mensaje2 no permite nulos
```

Código 3 Uso del operador ? para hacer que una variable permita nulos en Kotlin
Fuente: Elaboración propia

Otra ventaja de Kotlin es su capacidad para **inferir** automáticamente los tipos de datos. Esto significa que, en muchos casos, los desarrolladores no necesitan especificar explícitamente el tipo de datos de una variable, ya que el compilador puede deducirlo en función del valor asignado.

```
// variable nombre1 explícitamente configurado para que sea String
var nombre1:String = "Juan"
// Kotlin infiere que es un String, no es necesario hacerlo explícito
var nombre2 = "Juan"
```

Código 4 Inferencia de tipos en Kotlin
Fuente: Elaboración propia

Números Enteros

```
val edad1:Long  = 35 // 64 bits | -2**63 - 2**63-1
val edad2:Int   = 35 // 32 bits | -2**31 - 2**31-1
val edad3:Short = 35 // 16 bits | -2**15 - 2**15-1
val edad4:Byte  = 35 // 08 bits | -2**7 - 2**7-1
```

Código 5 Tipos de datos para números enteros en Kotlin y rangos de valores soportados
Fuente: Elaboración propia

Números de Punto Flotante

```
val peso:Double = 70.5  // MIN: 4.9E-324 | MAX: 1.7976931348623157E308
val peso2:Float = 70.5f // MIN: 1.4E-45  | MAX: 3.4028235E38
```

Código 6 Tipos de Datos Numéricos de Punto Flotante y rangos de valores soportados
Fuente: Elaboración propia

Otros Tipos de Datos

```
val nombre:String = "Santiago"
val multiLine:String = """
Lorem ipsum
dolor asit atme
consequeum
"""
val letraC:Char = 'c' //
val esDomingo = true // true or false
```

Código 7 Tipos de dato String, Char y Boolean
Fuente: Elaboración propia

Operadores

En Kotlin, hay una amplia gama de operadores para manipular y transformar datos. A diferencia de Java, Kotlin introduce algunos operadores adicionales que facilitan la escritura de código más conciso y legible. Algunos de estos operadores útiles incluyen el operador **"elvis" (?:)** para manejar nulos de manera segura, el operador de **rango (..)** para crear rangos numéricos o alfanuméricos de forma rápida, y el operador de comprobación de **tipo seguro (is)** para verificar la compatibilidad de tipos en tiempo de ejecución. Estos operadores adicionales en Kotlin no solo simplifican el código, sino que también brindan una mayor claridad y eficiencia en la expresión de lógica y manipulación de datos.

Acá un resumen de los operadores más utilizados:

Categoría	Operador	Descripción
Aritméticos	+	Suma
	-	Resta
	*	Multiplicación
	/	División
	%	Módulo (resto de la división)
Asignación	=	Asignación
	+=	Asignación de suma
	-=	Asignación de resta
	*=	Asignación de multiplicación
	/=	Asignación de división
	%=	Asignación de módulo
Comparación	==	Igual a
	!=	Distinto de
	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
Lógicos	&&	AND lógico
	||	OR lógico
	!	NOT lógico
Incremento/Decremento	++	Incremento
	--	Decremento
Operadores de Nulabilidad	?:	Operador Elvis
	?.	Llamada Segura (safe call) a un método o propiedad de un objeto que podría ser nulo
Rango	..	Para crear rangos numéricos o alfanuméricos
Tipo	is	Verifica el tipo

Tabla 1 Operadores más utilizados en Kotlin
Fuente: Elaboración propia

Sintaxis para números grandes

Para mejorar la legibilidad, puedes escribir utilizando guiones bajos en Kotlin:

```
val oneMillion  = 1_000_000  // integer
val colorHex    = 0xFF_00_CC // ej: new java.awt.Color(colorHex)
val dos         = 0b00000000_00000010 // 2 en bytes
```

Código 8 Ejemplo de código Kotlin usando _ para mejorar la legibilidad de números
Fuente: Elaboración propia

String Templates

Los String templates en Kotlin son una característica que permite la combinación de texto estático con expresiones o variables dentro de una cadena de caracteres. Esto simplifica la concatenación de valores en una cadena y hace que el código sea más legible y conciso.

```
val nombre = "Juan"
val mensaje = """
Querido $nombre,
Nos comunicamos con ud. por el motivo ...
El nombre con largo ${nombre.length}
"""
```

Código 9 Ejemplo del uso de String Templates en Kotlin
Fuente: Elaboración propia

Definición e importación de paquetes

Al igual que en Java estas especificaciones deben estar en la parte de arriba.

```
package cl.stgoneira.learning.android

import java.time.*
```

Código 10 Definición e importación de paquetes en Kotlin
Fuente: Elaboración propia

Punto de Entrada al programa

Al igual que en Java se necesita una función llamada main.

```
// puedes ser un main sin parámetros
fun main() {
    println("Hello world!")
}
// ---------------------------------
// o uno con el parámetro args
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Código 11 Punto de entrada al programa Kotlin
Fuente: Elaboración propia

Estructuras Condicionales

Las estructuras condicionales en Kotlin permiten tomar decisiones en función de ciertas condiciones. La estructura más común es el "if-else", donde se evalúa una expresión booleana y se ejecuta un bloque de código si la condición es verdadera, y otro bloque de código si es falsa. Kotlin también ofrece la estructura "when", que es una forma más expresiva y versátil de evaluar múltiples casos y tomar decisiones basadas en ellos. La estructura "when" permite evaluar diferentes condiciones y ejecutar el bloque de código correspondiente al primer caso que coincide. Además, Kotlin proporciona operadores condicionales como el "if-else" como expresiones, lo que significa que se pueden utilizar para asignar valores a variables en lugar de solo ejecutar bloques de código (Jetbrains, 2023).

If / else

```
import java.time.LocalDate
import java.time.DayOfWeek

val hoy = LocalDate.now().dayOfWeek
if( hoy.equals(DayOfWeek.SUNDAY) ) {
    println("Hoy se descansa!!!")
} else if(hoy.equals(DayOfWeek.MONDAY)) {
    println("Se acabo el descanso :(")
} else {
    println("A trabajar!!!")
}
```

Código 12 Ejemplo de Kotlin if/else
Fuente: Elaboración propia

```
val personasEnFila = 51
if( personasEnFila in 0..50 ) {
    println("Cerrar y atender a todos")
} else {
    println("Cerrar y atender a las primeras 50, al resto entregar número para atención mañana.")
}
```

Código 13 Ejemplo Kotlin if/else con rango
Fuente: Elaboración propia

```
var row = 2
val bg  = if(row%2 == 0) "#ccc" else "#fff"
```

Código 14 Kotlin no tiene operador ternario, pero puede usar if/else como una expresión
Fuente: Elaboración propia

when

```
val personasEnFile = 51

// como declaración
when( personasEnFile ) {
    0           -> println("Cerrar")
    in 1..50    -> println("Cerrar y atender a todos")
    else        -> println("Cerrar y atender a las primeras 50, al resto entregar número para atención mañana.")
}

// como expresión
var mensaje = when( personasEnFile ) {
    0           -> "Cerrar"
    in 1..50    -> "Cerrar y atender a todos"
    else        -> "Cerrar y atender a las primeras 50, al resto entregar número para atención mañana."
}
```

Código 15 Ejemplo de WHEN en Kotlin como declaración y expresión
Fuente: Elaboración propia

Bucles

Las estructuras de bucle en Kotlin permiten repetir bloques de código de manera controlada. La estructura más común es el bucle "for", que se utiliza para recorrer elementos de una colección o para iterar un número específico de veces. Kotlin también ofrece el bucle "while", que ejecuta un bloque de código mientras se cumple una condición booleana. Además, Kotlin proporciona el bucle "do-while", que es similar al "while", pero garantiza que el bloque de código se ejecute al menos una vez antes de verificar la condición (Jetbrains, 2023).

En el caso del bucle "for", se puede iterar sobre una variedad de elementos, como listas, arreglos o rangos numéricos, utilizando la sintaxis "for (elemento in colección)".

for loop

```
val clientes = arrayOf("juan@123.cl", "ramon@123.cl", "gonzalo@123.cl")

// For in
for( email in clientes) {
    println("Enviando correo a ${email} \n")
}

// Using an index
for( (index, email) in clientes.withIndex() ) {
    println("Email cliente #${index+1} ${email} \n")
}
```

Código 16 Ejemplos de For Loop con y sin índice en Kotlin
Fuente: Elaboración propia

```
/* Steps and ranges */
for (i in 1..5) print(i) // 12345

for (i in 5 downTo 1) print(i) // 54321

for (i in 1..5 step 2) print(i) // 135

for (i in 'a'..'d') print(i) // abcd

for (i in 'a'..'d' step 2) print(i) // ac
```

Código 17 Ejemplo de For Loop en Kotlin con Rangos y Pasos
Fuente: Elaboración propia

while loop

```
// while
var vueltas = 5
while(vueltas > 0) {
    println("quedan $vueltas vueltas \n")
    vueltas--
}
```

Código 18 Ejemplo de While Loop en Kotlin
Fuente: Elaboración propia

```
// do while
vueltas = 5
do {
    println("quedan $vueltas vueltas \n")
    vueltas--
} while (vueltas > 0)
```

Código 19 Ejemplo de DO WHILE en Kotlin
Fuente: Elaboración propia

repeat loop

```
repeat(2) {
    print("chao ")
}
// chao chao
```

Código 20 Ejemplo de REPEAT LOOP en Kotlin
Fuente: Elaboración propia

Funciones

Las funciones en Kotlin son bloques de código reutilizables que realizan una tarea específica. Se definen utilizando la palabra clave "fun" seguida del nombre de la función, los parámetros entre paréntesis y el tipo de retorno.

En Kotlin, es posible especificar valores predeterminados para los parámetros de una función, lo que permite llamar a la función omitiendo algunos argumentos si se desea utilizar los valores predeterminados. Esto proporciona flexibilidad al momento de invocar funciones y reduce la necesidad de sobrecargar funciones con múltiples variantes.

Además, Kotlin permite especificar parámetros requeridos y parámetros con nombre en las funciones. Los parámetros requeridos son aquellos que no tienen un valor predeterminado y deben ser proporcionados al llamar a la función. Por otro lado, los parámetros con nombre permiten especificar los argumentos de una función en cualquier orden, simplemente asignándolos utilizando su nombre correspondiente al invocar la función (Jetbrains, 2023).

Parámetros por Defecto

```
fun conducir(velocidad: String = "rápido") {
    println("conduciendo ${velocidad}")
}

fun main() {
    conducir()              // conduciendo rápido
    conducir("lento")   // conduciendo lento
    conducir(velocidad = "como una tortuga") // conduciendo como una //tortuga
}
```

Código 21 Ejemplo de función con parámetro por defecto en Kotlin
Fuente: Elaboración propia

Parámetros Requeridos

```
fun calcularAreaRectangulo(largo:Int, ancho:Int):Int {
    return largo * ancho
}

fun main() {
    calcularAreaRectangulo(2)       // error: no value paased for .....
    calcularAreaRectangulo(2, 3)    // 6
}
```

Código 22 Ejemplo de función con parámetros requeridos en Kotlin
Fuente: Elaboración propia

Parámetros con Nombre

```
calcularAreaRectangulo(largo = 2, ancho = 3)    // 6
```

Código 23 Ejemplo de llamada a función usando los nombres de los parámetros en Kotlin
Fuente: Elaboración propia

Funciones Compactas

```
// Versión normal
fun calcularAreaRectangulo(largo:Int, ancho:Int):Int {
    return largo * ancho
}
// ------------------------------------------------
// Versión compacta
fun calcularAreaRectangulo(largo:Int, ancho:Int):Int = largo * ancho
```

Código 24 Comparación de función en Kotlin de manera completa vs compacta
Fuente: Elaboración propia

Funciones de Primera Clase

Kotlin también ofrece características avanzadas en relación con las funciones. Las funciones son tratadas como ciudadanos de primera clase, lo que significa que pueden ser asignadas a variables, pasadas como argumentos y retornadas como valores. Esto facilita la implementación de patrones de programación como callbacks y programación funcional (Jetbrains, 2023).

Funciones Lambda

Kotlin admite el uso de lambdas, que son funciones anónimas que se pueden utilizar de forma concisa y expresiva. Las lambdas se utilizan comúnmente en combinación con las funciones de orden superior, que son aquellas funciones que pueden recibir otras funciones como parámetros o devolver funciones como resultado. Esto proporciona un alto nivel de flexibilidad y abstracción en el código (Jetbrains, 2023).

```
val saludo: () -> Unit          = { println("Hola") }
val saludo2: (String) -> Unit   = { nombre -> println("Hola ${nombre}")}
val calcArea: (Int, Int) -> Int = { largo, ancho -> largo * ancho }
// ------------------------------------------------------------------------
fun main() {
    saludo()        // Hola
    saludo2("Juan") // Hola Juan
    println( calcArea(3,2) )   // 6
}
```

Código 25 Ejemplo de funciones Lambda asignadas a variables en Kotlin
Fuente: Elaboración propia

Funciones de Orden Superior

Las funciones de orden superior toman funciones como parámetros o retornan una función.

```
val numeros = listOf(1, 2, 3)
val elDoble = numeros.map( {numero -> numero * 2} )
// cuando es el último parámetro no son necesarios los paréntesis
val tambienDoble = numeros.map { numero -> numero * 2}
// cuando hay un solo argumento, puedes usar "it"
val tambienDoble2 = numeros.map { it * 2}
// también puedes hacerlo como una variable
val elDobleFn: (Int) -> Int = { numero -> numero * 2 }
val tambienDoble3 = numeros.map(elDobleFn)
// si no es un Lambda puedes usar el operador ::
class Utils {
    companion object { // es como un miembro estático
        fun doble(numero:Int):Int = numero * 2
    }
}
val tambienDoble4 = numeros.map(Utils::doble)
```

Código 26 Ejemplo de uso de funciones de orden superior
Fuente: Elaboración propia

¿Cuál es la relevancia de que las funciones faciliten la implementación de patrones de programación?

Clases y Objetos

En Kotlin, las clases son estructuras fundamentales utilizadas para definir objetos y modelos de datos. Una clase es un molde que describe las propiedades y comportamientos que los objetos de esa clase pueden tener. Proporciona una plantilla para crear múltiples instancias (objetos) que comparten características comunes (Jetbrains, 2023).

La sintaxis en Kotlin para declarar una clase es bastante concisa. Para definir una clase, se utiliza la palabra clave "***class***", seguida del nombre de la clase y, opcionalmente un constructor primario y 0 o más constructores secundarios. Las propiedades de la clase se pueden definir dentro del constructor o en el cuerpo de la clase utilizando las palabras clave "**val**" o "**var**" para indicar si son de solo lectura o mutables, respectivamente (Jetbrains, 2023).

```
// declaración mínima de una clase
class Vacia
```

Código 27 Declaración mínima de una clase en Kotlin
Fuente: Elaboración propia

```
class Persona constructor(nombre:String, edad:Int)
```

Código 28 Clase en Kotlin con solo 1 constructor primario
Fuente: Elaboración propia

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

    // el constructor secundario debe "heredar" del primario usando this
    constructor(nombre: String) : this(nombre=nombre, edad=18) {
        // cuerpo constructor secundario
    }
}
```

Código 29 Clase en Kotlin con 1 constructor primario y 1 secundario
Fuente: Elaboración propia

```
// hay un constructor primario y un secundario
class Perro() {
    // cuando hay un primario los secundarios deben llamarlo con "this"
    constructor(dueno: String) : this() {
    }
}
```

Código 30 Otro ejemplo de clase en Kotlin con constructor primario y secundario
Fuente: Elaboración propia

Para instanciar una clase no es necesario la palabra clave ***new*** como en Java.

```
fun main() {
    val p = Perro() // usando constructor primario
    val p2 = Perro("Juan") // usando constructor secundario
}
```

Código 31 Ejemplo de creación de instancias de clase en Kotlin
Fuente: Elaboración propia

Si el constructor primario tiene anotaciones o modificadores debe usar la palabra clave "***constructor***", en caso contrario es posible omitirla.

```
class Persona public @Inject constructor(nombre:String)
```

Código 32 Constructor primario con modificadores y anotaciones
Fuente: Elaboración propia

El constructor primario no puede tener código, si necesita puede usar bloques de inicialización.

```
class Persona(nombre:String) {
    init {
        println("código de inicialización")
    }
}
```

Código 33 Ejemplo de Bloque de Iniciailzación en Kotlin
Fuente: Elaboración propia

Usando **val** (solo lectura) y **var** (mutable) puede declarar las propiedades de la clase.

```
class Persona(val nombre:String, var tieneEmpleo:Boolean = true)
```

Código 34 Ejemplo de clase en Kotlin creando propiedades en el constructor con val y var
Fuente: Elaboración propia

Si un constructor primario usa valores por defecto, automáticamente el compilador creará un constructor sin parámetros.

```
class Persona(
    var esEstudiante: Boolean = false,
    var nacionalidad: String = "Chile", // puedes usar trailing comma
)

fun main() {
    val p  = Persona() // funciona
    val p2 = Persona(true, "Venezuela") // funciona
    println(p.nacionalidad)
    println(p2.nacionalidad)
}
```

Código 35 Ejemplo en Kotlin de constructor sin parámetros creado automáticamente
Fuente: Elaboración propia

Además de las propiedades, el cuerpo de la clase puede contener funciones, que definen el comportamiento de los objetos de esa clase. Estas funciones o métodos

se declaran utilizando la palabra clave "**fun**", seguida del nombre del método, los parámetros y el tipo de retorno.

```
class Recangulo(val base:Int, val altura:Int) {

    fun calcularArea():Int {
        return base * altura
    }

    fun calcularPerimetro():Int {
        return (base + altura) * 2
    }
}
```

Código 36 Ejemplo de funciones o métodos en Kotlin para calcular el área y el perímetro de un rectángulo
Fuente: Elaboración propia

Regularmente se hace necesario crear clases que su principal motivo es mantener datos. En este tipo de clases, la funcionalidad necesaria se puede inferir a partir de sus propiedades. En Kotlin, estas clases son llamadas "**data classes**" y son marcadas con la palabra clave *data* (Jetbrains, 2023). Son muy parecidas a una clase POJO de Java, pero con mucho menos código.

```
data class Producto(val nombre:String, val precio:Int)
```

Código 37 Ejemplo de una Data Class en Kotlin

El compilador automáticamente deriva los siguientes miembros de todas las propiedades declaradas en el constructor primario:

- equals() y hashCode()
- toString() en la forma Clase(propiedad1=Valor1, propiedad2=Valor2)
- funciones componentN() para permitir la destructuración
- función copy()

Kotlin también ofrece la posibilidad de crear clases anidadas, interfaces, clases abstractas y herencia de clases utilizando el operador de dos puntos (:).

Herencia y Polimorfismo

La herencia es un mecanismo mediante el cual una clase puede heredar propiedades y comportamientos de otra clase. En Kotlin, se utiliza la palabra clave "***open***" o "***abstract***" para indicar que una clase puede ser heredada y la palabra clave "***override***" para indicar que un método o una propiedad está siendo sobrescrita en una clase derivada (Jetbrains, 2023).

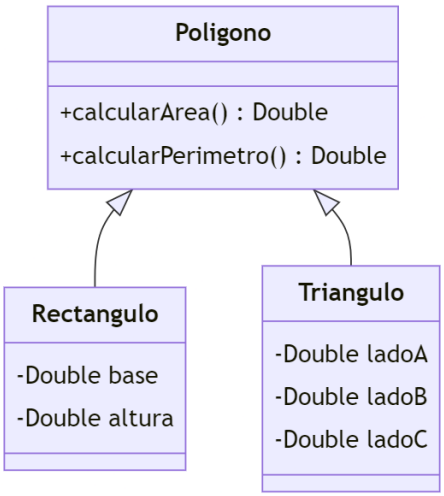


Figura 1 Diagrama UML Herencia clase Polígono
Fuente: Elaboración propia

```
abstract class Poligono {
    abstract fun calcularArea():Double
    abstract fun calcularPerimetro():Double
}
class Recangulo(val base:Double, val altura:Double):Poligono() {
    override fun calcularArea():Double = base * altura
    override fun calcularPerimetro():Double = (base + altura) * 2
}
class Triangulo(val ladoA:Double, val ladoB:Double, val ladoC:Double):Poligono() {
    override fun calcularArea(): Double {
        val s = calcularSemiperimetro()
        return Math.sqrt( s * (s-ladoA) * (s-ladoB) * (s-ladoC) )
    }
    fun calcularSemiperimetro():Double = calcularPerimetro() / 2
    override fun calcularPerimetro(): Double = ladoA + ladoB + ladoC
}
```

Código 38 Implementación en Kotlin de la jerarquía Polígono, Rectángulo y Triángulo
Fuente: Elaboración propia

El polimorfismo es la capacidad de una clase para tomar diferentes formas. En Kotlin, el polimorfismo se logra a través de la herencia y el uso de referencias de tipos más generales.

```
fun main() {
    var poligono:Poligono = Recangulo(10.0, 5.0)
    println("El área del polígono es: ${poligono.calcularArea()}")
    poligono = Triangulo(13.0, 15.0, 14.0)
```

```
    println("El área del polígono es: ${poligono.calcularArea()}")
}
```

Código 39 Ejemplo de Polimorfismo en Kotlin
Fuente: Elaboración propia

Anotaciones

Las anotaciones en Kotlin son una forma de agregar metadatos o información adicional a elementos del código, como clases, funciones o propiedades. Estas anotaciones se representan utilizando el símbolo @ seguido del nombre de la anotación.

Recordando el ejemplo de Jetpack Compose visto en el material de la semana anterior se utilizan tres anotaciones diferentes:
@OptIn(ExperimentalMaterial3Api::class), ***@Preview*** y ***@Composable***.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            CalculoAreaUI()
        }
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Preview
@Composable
fun CalculoAreaUI() {
    // ....
}
```

Código 40 Extracto del código de ejemplo de la semana anterior que utiliza algunas anotaciones
Fuente: Elaboración propia

La anotación ***@OptIn(ExperimentalMaterial3Api::class)*** se utiliza para habilitar el uso de una API experimental llamada Material3. Esta anotación permite utilizar características que aún están en fase experimental, lo que puede incluir cambios o actualizaciones en futuras versiones.

La anotación ***@Preview*** se utiliza para indicar que la función CalculoAreaUI() es una vista previa de la interfaz de usuario (UI) y se utilizará para mostrar cómo se ve en el editor de diseño. Esta anotación es especialmente útil durante el desarrollo, ya que permite visualizar rápidamente cómo se renderizará la interfaz de usuario.

La anotación ***@Composable*** se utiliza para indicar que la función CalculoAreaUI() es una función componible. Las funciones componibles son fundamentales en el desarrollo de interfaces de usuario en ***Jetpack Compose***, ya que representan

bloques de código reutilizables que definen una parte de la interfaz de usuario y se pueden combinar para construir interfaces más complejas.

Estas anotaciones son solo algunos ejemplos de cómo se pueden utilizar en Kotlin para agregar información adicional y personalizar el comportamiento de tus componentes y funciones. Puedes encontrar más anotaciones integradas en Kotlin y también puedes crear tus propias anotaciones personalizadas según tus necesidades.

¿Qué beneficios aporta Kotlin en comparación con otros lenguajes de programación?

Jetpack Compose y Vistas XML

Jetpack Compose

Jetpack Compose es un moderno toolkit de IU (Interfaz de Usuario) declarativa para Android, desarrollado por Google. En lugar de utilizar XML y las vistas tradicionales de Android, se utiliza Kotlin para definir la interfaz de usuario. Un ejemplo de uso es el siguiente:

```
@Composable
fun CalculoAreaUI() {
    Column {
        Text("Cálculo Área Rectángulo")
        Button(onClick = {

        }) {
            Text("Calcular")
        }
        Text("El resultado es: " ) // acá irá el resultado
    }
}
```

Código 41 Ejemplo de código Jetpack Compose en el comienzo de la creación de una pantalla para calcular el área de un rectángulo
Fuente: Elaboración propia

Se tiene una función anotada con ***@Composable*** llamada CalculoAreaUI, que define la interfaz de usuario. Dentro de esta función, se utiliza un ***Column*** como contenedor principal para organizar verticalmente los elementos.

Dentro del ***Column***, se encuentra un ***Text*** que muestra el título "Cálculo Área Rectángulo". Además, se crea un ***Button*** con el texto "Calcular" y se establece una acción a ejecutar cuando se haga clic en el botón.

Por último, se encuentra otro ***Text*** que mostrará el resultado del cálculo del área del rectángulo. En este punto, se puede agregar la lógica de cálculo y actualizar dinámicamente el contenido de este Text con el resultado obtenido.

Esta forma de utilizar Jetpack Compose permite construir una interfaz de usuario para el cálculo del área de un rectángulo de manera declarativa y eficiente. Jetpack Compose se encarga de manejar los cambios en el estado y actualizar automáticamente la interfaz de usuario en respuesta a dichos cambios.

Vistas XML

Antes de la llegada de Jetpack Compose, las aplicaciones de Android se construían utilizando XML y las vistas tradicionales de Android. Veamos cómo se crearía el mismo ejemplo de la pantalla de inicio utilizando vistas en XML:

Supongamos que se tiene el siguiente archivo XML llamado activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="30dp"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:text="Cálculo Área Rectángulo"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/editTextBase"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:padding="20dp"
        android:layout_marginTop="32dp"
        android:ems="10"
        android:hint="Base del rectángulo en CM"
        android:inputType="number"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView" />

    <EditText
        android:id="@+id/editTextAltura"
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:padding="20dp"
        android:layout_marginTop="32dp"
        android:ems="10"
        android:hint="Altura del rectángulo en CM"
        android:inputType="number"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editTextBase" />

    <Button
        android:id="@+id/buttonCalcularAreaRectangulo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:text="Calcular"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editTextAltura" />

    <TextView
        android:id="@+id/textViewResultado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp"
        android:text=""
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/buttonCalcularAreaRectangulo" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Código 42 Ejemplo completo de programa para cálculo del área de un rectángulo usando Vistas XML de Android
Fuente: Elaboración propia

En este caso, se utiliza un ***ConstraintLayout*** como contenedor principal. Los atributos ***android:layout_width="match_parent"*** y ***android:layout_height="match_parent"*** especifican que el ***ConstraintLayout*** debe ocupar todo el espacio disponible en su contenedor.

Para que los componentes interiores no queden pegados a los bordes del teléfono se utiliza el atributo ***android:padding="30dp"***. En Android, "dp" (**density-independent pixels**) es una unidad de medida utilizada para garantizar que los elementos de la interfaz de usuario se muestren de manera consistente en diferentes dispositivos con diferentes densidades de pantalla. La unidad dp se ajusta automáticamente en función de la densidad de píxeles del dispositivo, lo que permite que los elementos de la interfaz de usuario mantengan un tamaño relativo y una apariencia visual similar en pantallas con diferentes resoluciones.

El atributo ***tools:context=".MainActivity"*** establece el contexto de la vista, indicando que esta interfaz de usuario se utilizará en la actividad ***MainActivity*** (puedes cambiar esto según la actividad en tu aplicación).

Dentro del ConstraintLayout se definen los diferentes elementos de la interfaz de usuario:

- ***<TextView>***: para mostrar un texto como título.
- ***<EditText>***: para que el usuario pueda ingresar la base del rectángulo
- Otro ***<EditText>***: para que el usuario pueda ingresar la altura del rectángulo
- ***<Button>***: para que el usuario pueda presionar y se calcule el área
- Y otro ***<TextView>***: para mostrar resultado.

```kotlin
package cl.stgoneira.learning.android.learningkotlinxml

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.TextView

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val etBase      = findViewById<EditText>(R.id.editTextBase)
        val etAltura    = findViewById<EditText>(R.id.editTextAltura)
        val btn         = findViewById<Button>(R.id.buttonCalcularAreaRectangulo)
        val tvResultado = findViewById<TextView>(R.id.textViewResultado)

        btn.setOnClickListener {
            val base    = etBase.text.toString().toIntOrNull() ?: 0
            val altura  = etAltura.text.toString().toIntOrNull() ?: 0
            val area    = calcularAreaRectangulo(base, altura)
            tvResultado.text = "El área del rectángulo es ${area}"
        }
    }
}

fun calcularAreaRectangulo(base:Int, altura:Int):Int {
    return base * altura
}
```

Código 43 Ejemplo completo de archivo MainActivity escrito en Kotlin para que funcione la Vista XML
Fuente: Elaboración propia

Este código hace lo siguiente:

1. ***override fun onCreate(savedInstanceState: Bundle?)***: Sobrescribe el método onCreate() de la clase AppCompatActivity. Este método se llama cuando la actividad se crea.
2. ***super.onCreate(savedInstanceState)***: Llama al método onCreate() de la clase base AppCompatActivity para realizar las tareas de inicialización necesarias.

3. ***setContentView(R.layout.activity_main)***: Establece el diseño de la interfaz de usuario para la actividad utilizando el archivo de diseño llamado "**activity_main.xml**".
4. ***val etBase = findViewById<EditText>(R.id.editTextBase)***: Busca y asigna el elemento ***EditText*** con el id "***editTextBase***" en el archivo de diseño a la variable ***etBase***.
5. Lo mismo que el punto anterior con la altura, el botón y el TextView para mostrar el resultado.
6. ***btn.setOnClickListener { ... }***: Configura un ***OnClickListener*** para el botón. Cuando se hace clic en el botón, se ejecuta el código dentro del bloque de llaves.
7. ***val base = etBase.text.toString().toIntOrNull() ?: 0***: Obtiene el texto ingresado en el ***EditText etBase***, lo convierte a un entero o devuelve null si no se puede convertir. Si es null, se asigna el valor 0 a la variable base.
8. Lo mismo del paso anterior para la altura
9. ***val area = calcularAreaRectangulo(base, altura)***: Llama a la función calcularAreaRectangulo(), pasando los valores de base y altura como argumentos, y asigna el resultado a la variable area.
10. ***tvResultado.text = "El área del rectángulo es ${area}"***: Finalmente, se establece el texto del TextView tvResultado para mostrar el resultado del cálculo.

Comparación entre Jetpack Compose y Vistas en XML

Jetpack Compose y las Vistas en XML son dos enfoques diferentes para construir interfaces de usuario en Android. Mientras que Jetpack Compose utiliza Kotlin para definir la interfaz de usuario de forma declarativa y dinámica, las Vistas en XML utilizan archivos XML estáticos para describir la interfaz. Veamos algunas diferencias clave:

Jetpack Compose ofrece una sintaxis más concisa y legible, lo cual se evidencia en el ejemplo anterior. Además, proporciona un flujo de trabajo más flexible y dinámico, permitiendo realizar cambios en tiempo real en la interfaz de usuario.

Por otro lado, las Vistas en XML ofrecen una separación clara entre la lógica de la aplicación y la presentación visual. El ejemplo XML muestra cómo se define la estructura y apariencia de la interfaz de usuario mediante etiquetas XML y atributos.

En resumen, Jetpack Compose es la tecnología más nueva y recomendada para desarrollar interfaces de usuario en Android. Proporciona una forma más intuitiva y eficiente de construir interfaces de usuario, aunque las Vistas en XML siguen siendo una opción válida y utilizada en proyectos existentes. Dependiendo de las necesidades y preferencias del desarrollador, se puede elegir la opción que mejor se adapte al contexto del proyecto.

¿De qué manera el enfoque declarativo de Jetpack Compose podría simplificar el proceso de diseño y desarrollo de interfaces de usuario en comparación con el enfoque basado en vistas XML?

Cómo moverse entre Actividades (Intent)

Para saltar de una actividad a otra en Android, se utiliza el concepto de Intents. Un Intent es un objeto que se utiliza para comunicarse entre componentes de una aplicación, como actividades, servicios y receptores de difusión (Android Developers, 2023).

Aquí hay un ejemplo de cómo se puede realizar la transición de una actividad a otra:

1. Primero, se debe asegurar de tener la definición de la actividad de destino en el archivo de manifiesto (**AndroidManifest.xml**). Debe haber una entrada para la actividad de destino dentro de la etiqueta ***<application>***.
2. En la actividad actual, donde se realiza la transición, se debe crear un objeto ***Intent*** que especifique la actividad de destino. Puedes usar el nombre de la clase de la actividad de destino como argumento en el constructor del Intent.
3. Si deseas pasar datos adicionales a la actividad de destino, puedes utilizar métodos como ***putExtra()*** del objeto Intent para agregar esos datos.
4. Finalmente, se inicia la nueva actividad utilizando el método ***startActivity()*** pasando el objeto Intent como argumento.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Button(onClick = {
                val intent = Intent(this, Otra::class.java)
                startActivity(intent)
            }) {
                Text(text = "Ir a la otra actividad")
            }
        }
    }
}
```

Código 44 Activity con Jetpack Compose que usa un Intent para saltar a otra Actividad al presionar un Botón
Fuente: Elaboración propia

Conclusión

En este material de estudio, se han explorado los fundamentos de Kotlin, un lenguaje de programación moderno y poderoso que se utiliza ampliamente en el desarrollo de aplicaciones para Android. Se ha abordado cómo trabajar con variables mutables y no mutables, así como el uso de diferentes tipos de datos básicos.

Además, se ha adquirido familiaridad con los operadores y las estructuras condicionales y de bucles, elementos fundamentales para controlar el flujo de ejecución en los programas. Se ha comprendido cómo utilizar funciones y los distintos tipos de parámetros, incluyendo los parámetros por defecto y requeridos. También se ha explorado la sintaxis compacta para escribir funciones de forma más concisa.

Un aspecto destacado ha sido el estudio de las funciones de primera clase en Kotlin, que permiten tratar las funciones como variables y utilizarlas como lambdas. Esto brinda una gran flexibilidad a la hora de diseñar y componer el código.

En cuanto a la programación orientada a objetos, se ha abordado el tema de las clases y la herencia en Kotlin, lo cual posibilita la organización del código de manera estructurada y reutilizable. También se ha aprendido sobre las anotaciones y su utilidad para proporcionar metadatos y configuraciones adicionales en las clases y métodos.

Para la construcción de interfaces de usuario, se han explorado los conceptos básicos de Jetpack Compose, una herramienta moderna de desarrollo de interfaces declarativas. A través de la creación de vistas XML, se ha aprendido a diseñar la apariencia visual de las aplicaciones de manera sencilla y flexible.

Finalmente, se ha abordado el uso básico de Intents para la navegación entre Activities, permitiendo saltar de una pantalla a otra en las aplicaciones.

Este material de estudio ha proporcionado los conocimientos básicos necesarios para dar los primeros pasos en el desarrollo de aplicaciones para Android. A partir de aquí, se puede continuar explorando y profundizando en los conceptos y técnicas presentadas, y seguir aprendiendo en este emocionante campo de la programación móvil.

Bibliografía

Android Developers. (24 de 07 de 2023). *Cómo iniciar otra actividad*. Obtenido de https://developer.android.com/training/basics/firstapp/starting-activity?hl=es-419

Jetbrains. (20 de 07 de 2023). Obtenido de https://kotlinlang.org/docs/android-overview.html

Jetbrains. (24 de 07 de 2023). *Basic Syntax*. Obtenido de https://kotlinlang.org/docs/basic-syntax.html

Jetbrains. (24 de 07 de 2023). *Basic Types*. Obtenido de https://kotlinlang.org/docs/basic-types.html

Jetbrains. (24 de 07 de 2023). *Classes*. Obtenido de https://kotlinlang.org/docs/classes.html

Jetbrains. (24 de 07 de 2023). *Conditions and loops*. Obtenido de https://kotlinlang.org/docs/control-flow.html

Jetbrains. (24 de 07 de 2023). *Data Classes*. Obtenido de https://kotlinlang.org/docs/data-classes.html

Jetbrains. (24 de 07 de 2023). *Functions*. Obtenido de https://kotlinlang.org/docs/functions.htm

Jetbrains. (24 de 07 de 2023). *High-order functions and lambdas*. Obtenido de https://kotlinlang.org/docs/lambdas.html

Jetbrains. (24 de 07 de 2023). *Inheritance*. Obtenido de https://kotlinlang.org/docs/inheritance.html

Iplacex

enovus

4 AÑOS

INSTITUCION ACREDITADA

NIVEL AVANZADO

Hasta octubre 2025

Comisión Nacional de Acreditación CNA-Chile

GESTIÓN INSTITUCIONAL Y DOCENCIA DE PREGRADO