Universidad Autónoma del Estado de México
Centro Universitario UAEM Zumpango

Ingeniería en computación

Lenguajes de Programación Kotlin y Java para desarrollo de Aplicaciones de Dispositivos Móviles

ENSAYO

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACIÓN

PRESENTA:

Armando Fragoso Alameda

DIRECTOR DE ENSAYO:
Dr. Asdrúbal López Chau

Mayo, 2022

Resumen

En este trabajo se presenta una descripción de dos de los lenguajes de programación más importantes para desarrollo de aplicaciones nativas para sistema operativo Android: Java y Kotlin. El primero de ellos fue el lenguaje oficial para desarrollo en Android, mientras que el segundo es el que la empresa Google recomienda para nuevos desarrollos. En el documento se presentan los principales componentes de una aplicación Android; los fundamentos de la programación orientada a objetos, en los que se incluyen ejemplos sobre la forma de implementar los conceptos en Java y en Kotlin; el proceso para crear una aplicación para Android y un ejemplo completo desarrollado en ambos lenguajes.

Abstract

This work presents a description of two of the most important programming languages for developing native applications for the Android operating system: Java and Kotlin. The first of them was the official language for Android development, while the second is the one that Google recommends for new developments. The document presents the main components of an Android application; the basics of object-oriented programming, including examples of how to implement the concepts in Java and Kotlin; the process to create an Android application and a complete example developed in both languages.

Contenido

Lista de figuras

Lista de tablas

Lista de códigos

Capítulo 1

INTRODUCCIÓN

Actualmente, existen un gran número de lenguajes de programación, por nombrar a algunos, se tienen a Fortran, C, LISP, ALGOL, BASIC, COBOL, Smalltalk, C++, Haskell, Python, PHP, Java, Kotlin o DART. Algunos de estos lenguajes tienen muchos años de haber sido creados, otros son muy conocidos actualmente, mientras que otros son usados en ambientes o aplicaciones muy específicas.

Hace menos de dos décadas, estuvieron disponibles para el público general los teléfonos inteligentes, con ello, se creó un nuevo tipo de mercado para el desarrollador de software: el mercado de las aplicaciones para dispositivos móviles.

Uno de los lenguajes de programación que se popularizó masivamente entre los desarrolladores de aplicaciones para teléfonos inteligentes fue Java. En muchas universidades del país se ha enseñado este lenguaje desde hace varios años. Como siempre sucede, los cambios en la tecnología permiten avances, a cambio de adaptarse a nuevas herramientas, paradigmas o formas de trabajo. En el desarrollo de aplicaciones para dispositivos móviles con sistema operativo Android, un nuevo lenguaje de programación entró en escena hace unos cuantos años (en 2011 para ser exactos). A diferencia de muchos otros lenguajes, Kotlin fue diseñado para inter-operar con Java, es decir, el código que se genera al compilarse un programa en Kotlin, es capaz de ejecutarse directamente en una plataforma Java (o máquina virtual de Java). Esto significa que Kotlin puede

sustituir a Java, si no es que ya lo está haciendo.

En este ensayo se presentan de manera general ambos lenguajes de programación, Java y Kotlin, mostrando sus similitudes y diferencias principales. Además, se muestra como implementar una aplicación en ambos lenguajes.

Java es el lenguaje de programación nativo para desarrollar aplicaciones Android; sin embargo, recientemente, en 2011, Kotlin ha surgido como un lenguaje capaz de interoperar con código Java. En 2017, Kotlin fue nombrado por Google como lenguaje oficial para desarrollar aplicaciones Android. Pese a que ambos lenguajes comparten varias características (y también tienen diferencias notables), a cinco años del lanzamiento de Kotlin todavía no cuenta con una cantidad de desarrolladores tan grande como los de Java. En este ensayo se presentan las características de ambos lenguajes de programación, dando una opinión del autor sobre las ventajas y desventajas de cada uno de ellos. Con ello, se espera contribuir a las nuevas generaciones a familiarizarse más rápidamente con Kotlin, y reducir su curva de aprendizaje.

El objetivo principal de este trabajo es identificar las diferencias y similitudes del desarrollo de aplicaciones para sistema operativo Android con lenguaje de programación Java y Kotlin. Para el objetivo de este logro, se tiene considerado lograr los siguientes:

- Presentar los elementos básicos de los lenguajes de programación Java y Kotlin.
- Comentar las diferencias y similitudes entre Java y Kotlin.
- Presentar los conceptos de la programación orientada a objetos, y la forma en que estos se implementan en Java y Kotlin.
- Mostrar el proceso general para la creación de una aplicación en Android Studio con los lenguajes de programación Java y Kotlin.
- Discriminar cuál de los dos lenguajes de programación que se considera más adecuado para desarrollo de aplicaciones Android.

1.1 Justificación

El aprendizaje continuo de nuevas tecnologías, lenguajes de programación y herramientas de desarrollo es un requerimiento que los ingenieros en computación y otras carreras afines deben de llevar a cabo en su vida profesional de manera cotidiana. Si bien es cierto que al tener conocimiento de un lenguaje de programación facilita mucho el aprendizaje de otro similar, el autor de este documento considera importante el contar con un documento en el cual se resuman las similitudes y diferencias entre estos dos lenguajes de programación, y que contenga comentarios de alguien con experiencia en el mercado laboral desarrollando aplicaciones en ambos lenguajes.

1.2 Alcance

Se presentarán los elementos fundamentales de los lenguajes de programación Java y Kotlin, así como los conceptos de programación orientada a objetos, y la forma en que estos se implementan en ambos lenguajes.

Opiniones del autor

El documento se incluye información técnica, explicaciones propias y opiniones del autor sobre los temas desarrollados. Estas últimas son presentadas en color verde.

Capítulo 2

DESARROLLO

En este capítulo se realiza una breve revisión al sistema operativo Android, así como a a una de las herramientas de desarrollo más utilizadas para crear aplicaciones para este sistema. Se muestra el diseño de interfaces y las maquetas de diseño. Además, se presentan los conceptos de la programación orientada a objetos y la implementación en Java y Kotlin, explicando diferencias y similitudes, así como las tecnologías y conceptos comunes a ambos lenguajes de programación comparados, Java y Kotlin.

2.1 Breve historia del sistema operativo Android

Android es un sistema operativo basado en el núcleo de Linux [23] [9]. Fue diseñado desde su inicio para ser instalado y usado en dispositivos móviles con tecnología táctil (conocida como interfaz tipo *touch*), es decir, para dispositivos electrónicos que usan una pantalla táctil como interfaz. Este sistema fue inicialmente desarrollado por la empresa Android Inc., a la cual Google respaldó económicamente para el desarrollo del proyecto. En 2005, Google adquirió los derechos de Android, mismo que fue dado a conocer en el año 2007. La versión de Android 1.0 salió públicamente el 23 de septiembre de 2008.

Uno de los primeros dispositivos móviles que usaron el sistema Android fue el HTC Dream (T-Mobile G1), el cual se comercializó en octubre de 2008. La figura 2.1 muestra

Figura 2.1: T-Mobile G1 el primer teléfono con sistema operativo Android

la apariencia de este dispositivo [23]. El dispositivo mostrado en la figura 2.1 no tenía una interfaz touch completa, es decir, dependía de un teclado físico. Salieron varios modelos de teléfonos con este tipo de teclado, sin embargo, ahora predominan los de teclado virtual.

La versión básica de Android es conocida como Android Open Source Project (AOSP). El 25 de junio de 2014 en la Conferencia de Desarrolladores Google I/O, Google mostró una evolución de la marca Android, con el fin de unificar tanto el hardware como el software y ampliar mercados.

Actualmente Android es el sistema operativo instalado en más dispositivos móviles en el mundo, seguido por iOS, de la empresa Apple. En los últimos años, he observado que aunque Android sigue siendo el sistema operativo dominante en plataformas de dispositivos móviles, la cantidad de usuarios de iOS cada vez es mayor. Esto quizás pueda deberse a que los precios de los teléfonos inteligentes y tabletas electrónicas Android de gama alta, llegan a ser similares a los precios que ofrece Apple para sus dispositivos móviles.

La figura 2.2 presenta una comparativa entre el número de usuarios de cada uno de

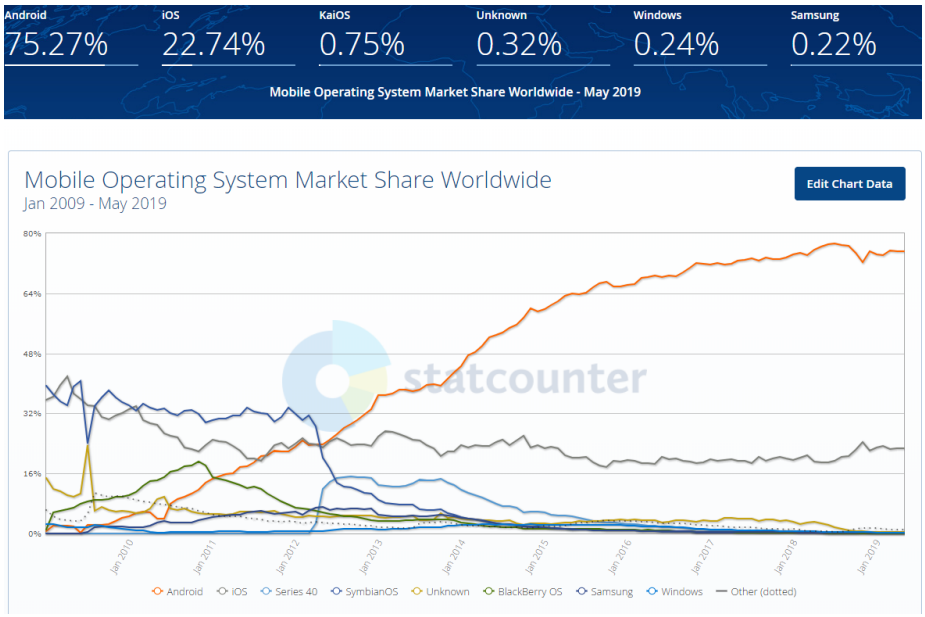

Figura 2.2: Cuota de mercado del sistema operativo móvil en todo el mundo en 2010. Consultado en marzo de 2022.

estos dos sistemas operativos [2].

Android ha tenido varios cambios y actualizaciones casi cada año, a modo de resumen histórico, se presentan las versiones con sus fechas de lanzamiento:

- Apple Pie (1.0): 23 de septiembre de
- Banana Bread (1.1): 9 de febrero de 2009
- Cupcake (1.5): 25 de abril de 2009
- Donut (1.6): 15 de septiembre de 2009
- Eclair (2.0 – 2.1): 26 de octubre de 2009
- Froyo (2.2 – 2.2.3) : 20 de mayo de 2010
- Gingerbread (2.3 – 2.3.7): diciembre de 2010
- Honeycomb (3.0 – 3.2.6): 22 de febrero de 2011
- Ice Cream Sandwich (4.0 – 4.0.5): 18 de octubre de 2011

- Jelly Bean (4.1 – 4.3.1): 9 de julio de 2012
- KitKat (4.4 – 4.4.4): 31 de octubre de 2013
- Lollipop (5.0 – 5.1.1): 12 de noviembre de 2014
- Marshmallow (6.0 – 6.0.1): 5 de octubre de 2015
- Nougat (7.0 – 7.1.2): 15 de junio de 2016
- Oreo (8.0 – 8.1): 21 de agosto de 2017
- Pie (9.0): 6 de agosto de 2018
- Q (10.0): agosto del 2019
- Android 11 (11.0): 8 de septiembre de 2020
- Android 12 (12.0): septiembre de 2021
- Android 13 (13.0): en pruebas (beta para programadores)

En la siguiente sección, se presenta una revisión de los lenguajes de programación Java y Kotlin. El primero de ellos ha sido usado ampliamente para desarrollar aplicaciones para Android, mientras que el segundo es un lenguaje que cada vez es más usado para desarrollar este tipo de aplicaciones. Sin embargo, actualmente Google ha declarado que Kotlin lo es. De acuerdo con mi experiencia desarrollando aplicaciones con ambos lenguajes de programación, considero que esta decisión de Google fue acertada, el lenguaje se siente más fresco y actual.

A continuación se presentan los principales componentes de una aplicación para dispositivos con sistema operativo Android.

2.2 Componentes de una aplicación para Android

2.2.1 Componente View

Las vistas (*View*) son elementos de una aplicación Android con las que el usuario interactúa. Permiten contener elementos, como textos o etiquetas, cajas de texto para que el usuario introduzca datos, botones, imágenes, vídeos, animaciones, etc. [7].

2.2.2 Componente Layout

Los diseños (*Layout*) los podemos interpretar como una hoja en blanco sobre la cual podemos colocar los elementos necesarios que requiera la pantalla de la aplicación. Sobre estos diseños se ordenan y se definen el alto, ancho, tipo de alineación ya sea vertical, horizontal, o agregar pesos que definan el porcentaje del componente View a ocupar [7]. Estos componentes me han resultado bastante útiles para la creación de interfaces de usuario.

2.2.3 Clase Activity

Una de las clases principales para el desarrollo de aplicaciones Android es Activity. Esta hereda de la clase base AppCompatActivity, encargada de hacer que algunas características de versiones recientes de Android puedan ser usadas en dispositivos que tienen instaladas versiones anteriores de Android.

La clase AppCompatActivity tiene un componente View relacionado, y tiene un método específico en el que se agrega la parte del código que tiene la lógica de la aplicación [1]. Como puede observarse, el concepto de herencia es bastante usado en el desarrollo de aplicaciones para dispositivos móviles, por lo que es necesario tener un conocimiento profundo de la programación orientada a objetos.

Figura 2.3: Las actividades no pueden contener actividades

2.2.4 Componente Fragment

Los Fragments al igual las Activity, heredan de una clase, y tienen relacionada un componente View. La diferencia más notable entre un Fragment (que representan comportamientos o partes de una interfaz gráfica) y una Activity es que este último puede contener uno o más componentes Fragment. Sin embargo, un Activity no puede contener más activities, y dentro de un Fragment no pueden haber Activities. Los Fragments para ser invocados necesitan de un Activity. En la figura 2.3 se muestra un ejemplo de como las actividades no pueden contener actividades, la figura 2.4 muestra cómo una actividad puede contener uno o más fragments y la imagen 2.5 muestra como los fragments pueden contener una o más vistas (Views) [1].

2.2.5 Servicios

Los servicios son aplicaciones que pueden ser escritas en diversos lenguajes, la mayor parte de los servicios ayudan a almacenar información de las aplicaciones ya sea fotos, vídeos, textos, etc. La mayor parte de las aplicaciones hace uso de uno o más servicios, ya sea para iniciar sesión o hacer cálculos en la nube [1], la imagen 2.6 muestra como un servicio se comunica con una aplicación móvil.

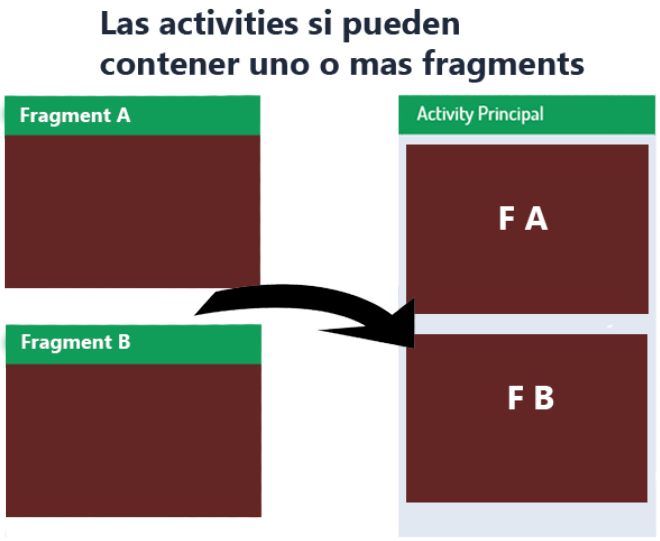

Figura 2.4: Las Activities si pueden contener más de un fragment

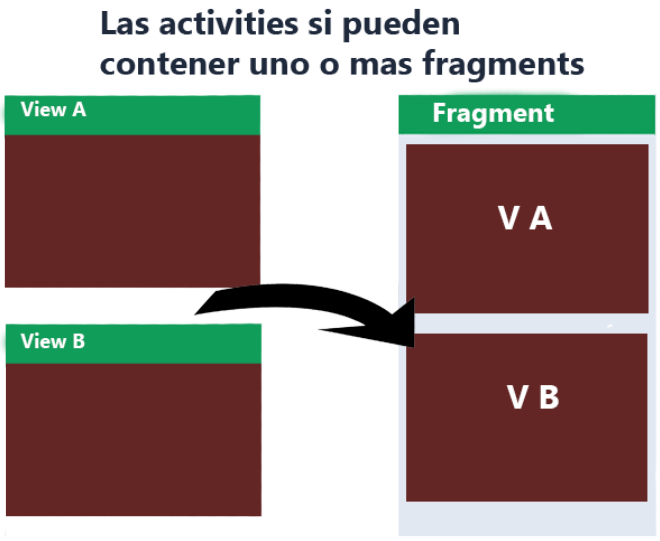

Figura 2.5: Los fragments al igual que las actividades pueden contener vistas(views)

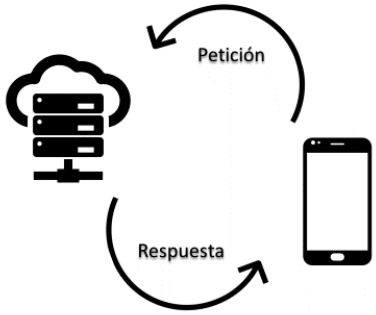

Figura 2.6: Modelo cliente servidor usado en aplicaciones para dispositivos móviles.

2.2.6 Intent

De manera resumida, un Intent es el mecanismo de comunicación entre los componentes de una aplicación. Cuando en las aplicaciones se necesita hacer la invocación a una actividad (Activity), se hace uso de Intent. Por ejemplo, si en una aplicación hay un botón para tomar fotografías, cuando el usuario da clic sobre él, dentro de la aplicación se crea un Intent que inicia la cámara [13], cuando se termina de usar la cámara, se regresa a la aplicación mediante la creación de otro Intent. La figura 2.7 muestra este ejemplo de manera visual.

2.2.7 Ciclo de vida de una aplicación Android

Para entender cómo se integran en una aplicación los elementos revisados en la sección anterior, se presentará ahora el ciclo de vida de una aplicación para dispositivos móviles con sistema operativo Android.

Cada pantalla o interfaz que se le presenta al usuario tiene asociada un componente Activity o uno o más componentes de tipo Fragment. Cada uno de estos tipos de componentes pasa por una serie de estados (onCreate, onRestart, onResume y onPause), que en su conjunto crean un ciclo de vida [2]. Esto tiene cierta similitud con el ciclo de los seres vivos: nacer, crecer o desarrollarse e, inevitablemente morir. Por supuesto, en Android los conceptos biológicos no se aplican literalmente. Veamos ahora cada uno de

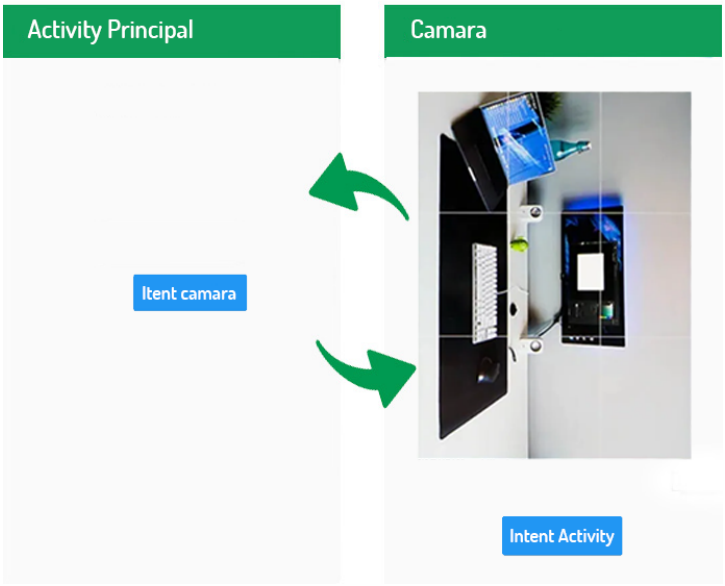

Figura 2.7: Un ejemplo de componente Intent para abrir la cámara, y para para regresar a la pantalla principal.

los estados del ciclo de vida de aplicaciones Android.

Estado onCreate

Este es el estado inicial de una aplicación, y cuando se inicia la vista del componente, cuando el usuario da clic en el icono de la aplicación. Este estado se utiliza generalmente para inicializar valores de variables o declarar constantes a usar en la aplicación. En el método que se ejecuta en este estado, se pueden pasar datos dinámicos iniciales. Es importante aclarar que se pasa por este estado cada vez que se crea una vista, pero antes de que se visualice.

Estado onRestart

La aplicación pasa por este estado cuando el sistema operativo Android detuvo por alguna razón la ejecución de la aplicación.Esto puede ocurrir, por ejemplo, cuando el usuario minimiza la aplicación o cuando cambia a otra aplicación sin cerrar la anterior [12]. Cuando la aplicación vuelve a ejecutarse, se ejecuta el método correspondiente al

estado onStart.

Estado onStart

Durante su ciclo de vida, una aplicación pasa por el estado onStart inmediatamente después de haber estado en cualquiera de los estados onRestart u onCreate.

Este estado sucede previamente a la visualización de la interfaz al usuario.

Estado onResume

Cuando el usuario está interactuando con la aplicación, esta se encuentra en el estado onResume. Esto significa que la aplicación está siendo mostrada en la pantalla del dispositivo móvil, y que el usuario está haciendo uso de ella.

Estado onPause

Este estado ocurre cuando el usuario abandona la vista pero no precisamente la finaliza, esto nos ayuda hacer llamadas a varias vistas y no perder información. Una vez que se llama a una vista sin finalizarla, esta pasa al estado onPause, una vez se regresa cambia a estado onResume [12]. En la práctica, cuando se ejecuta el método asociado con este estado, generalmente se detienen los procesos que consumen muchos recursos como lo es la reproducción de multimedia, además, he notado que en este estado no deberían de involucrarse operaciones que consumen mucho tiempo de procesamiento, ya que en caso contrario la aplicación se percibe lenta.

Estado onStop

La aplicación pasa por el estado onStop cuando no está visible para el usuario. Los dos posibles estados siguientes pueden ser onRestart o onDestroy, este último será explicado a continuación.

Estado onDestroy

Este es el estado de una aplicación que está a punto de terminar. La actividad puede terminar de manera deliberada por parte del usuario, o por que el sistema operativo así lo decide, siendo la escasa cantidad de memoria principal libre uno de los motivos.

Un resumen visual del ciclo de vida de una actividad de Android se muestra en la figura 2.8.

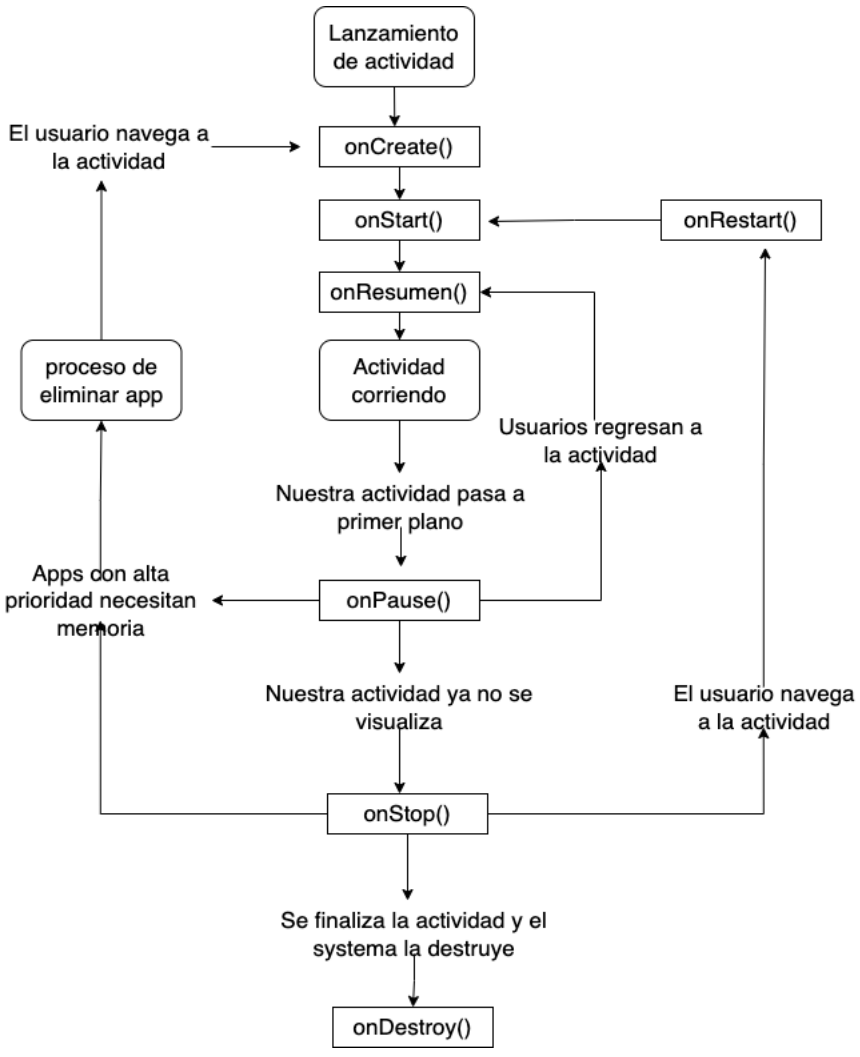

Figura 2.8: Ciclo de vida de una Activity de Android

2.3 Fundamentos de la Programación Orientada a Objetos (POO)

Como se pudo observar en los componentes de una aplicación Android, la POO es utilizada ampliamente para su desarrollo. La POO es un paradigma de programación que modela los objetos del mundo real, identificando los atributos y comportamientos que son considerados de importancia en cierto contexto.

La programación orientada a objetos nos ayuda a ver los problemas como una relación entre diferentes objetos que se relacionan entre sí. Cada cualidad de los objetos es abstraída y pasada a datos, un ejemplo: tenemos el objeto persona y abstraemos sus atributos como peso, edad, nombre apellido, etc.

Es fácil crear sistemas basados en objetos que se comunican entre sí, de esta manera se reutiliza código y se crean módulos del mismo. Existen características que definen la programación orientada a objetos [4]. A continuación se presentan algunas de estas características.

2.3.1 Abstracción

Es la etapa del desarrollo de software donde se analizan, identifican y capturan los comportamientos de las entidades (objetos) involucradas en la solución de un problema. Los objetos del mundo real sirven como modelo de un "agente" abstracto que puede realizar un trabajo, informar, cambiar de estado, y "comunicarse" con otros objetos en el sistema. Todo esto sin revelar detalladamente cómo se implementan estas características [4].

La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a identificar e implementar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar.

La abstracción ayuda a modelar problemas de la vida cotidiana mediante objetos. Es muy útil ya que se pueden abstraer muchas cosas y definirlas como variables y métodos,

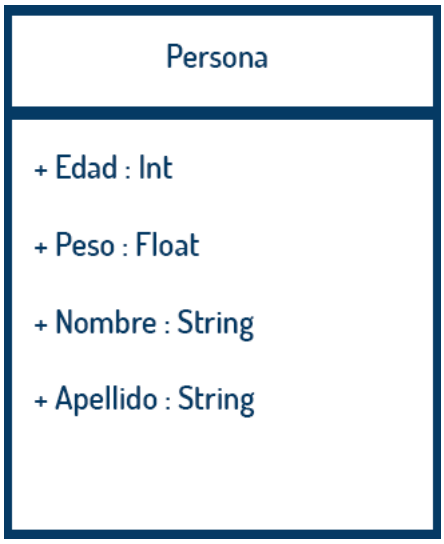

Figura 2.9: Abstracción de un objeto

las variables son los atributos del objeto mientras los métodos son los comportamientos de este.

La figura 2.9 nos ayuda a tener un poco más clara la abstracción [13].

2.3.2 Encapsulamiento

En el lenguaje de programación orientado a objetos, el encapsulamiento consiste en ocultar el estado de los datos miembro de un objeto. Para acceder a estos datos, sólo su puede hacer a través de métodos o funciones privadas dentro de la misma clase. La figura 2.10 se muestra la forma en que es posible el acceso a los datos [4].

El ocultar o no dejar que desde cualquier parte del código ayuda a evitar que los datos miembros tengan valores incorrectos, ya que dentro de los métodos de acceso puede escribirse código necesario para verificar cualquier error.

2.3.3 Modificadores de acceso a miembros de un objeto

Todo lenguaje de programación orientado a objetos tiene modificadores de acceso, para el caso de Java y Kotlin, estos modificadores son similares.

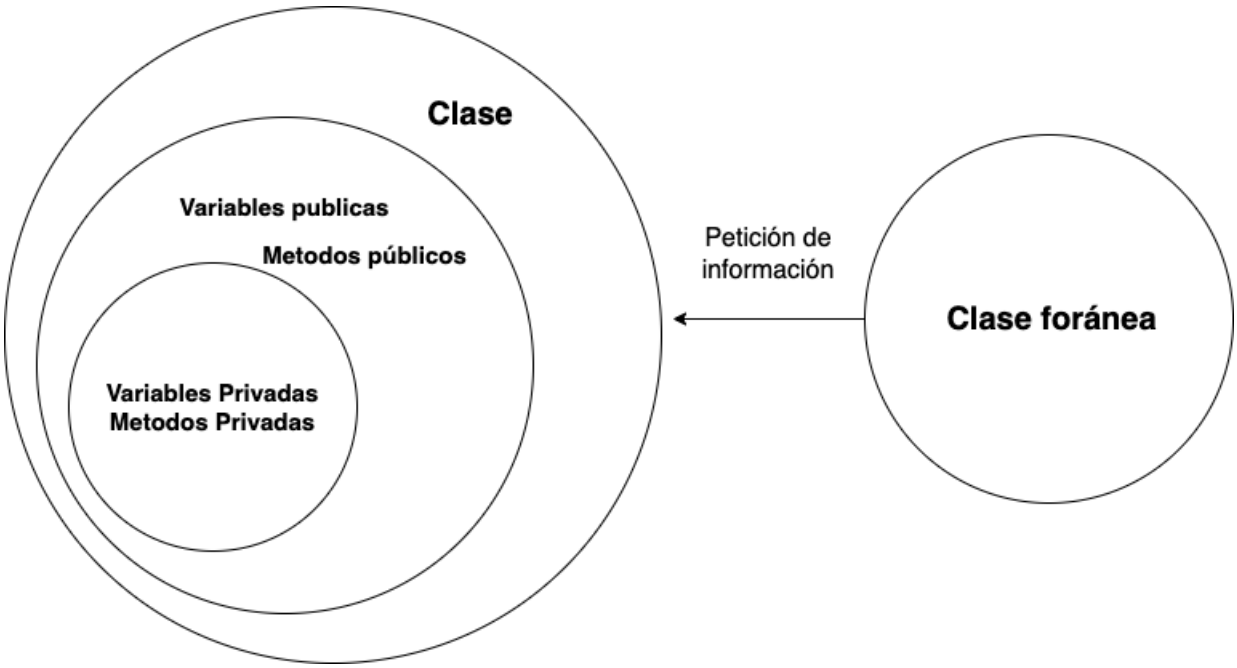

Figura 2.10: Concepto de encapsulamiento de datos

A continuación daremos un pequeño repaso de lo que son los modificadores de acceso, ya que nos ayudan a definir variables que no pueden ser accedidas desde cualquier lugar en el código. Esto se relaciona con el tema mostrado en la sección anterior (encapsulamiento).

Modificador de acceso predeterminado o default

No existe una palabra reservada para este modificador. Es decir, si una clase, dato o método miembro se declara sin escribir algún modificador de acceso, sólo se puede acceder dentro de la misma clase, o subclases dentro del mismo paquete [5].

Modificador de acceso privado o private

Los miembros declarados como privados (private) sólo son accesibles dentro de la clase en la que se declaran. sólo los métodos miembro pueden acceder o modificar los atributos privados del mismo objeto. Las interfaces no se pueden declarar con miembros privados

[5].

Modificador de acceso protegido o protected

Los atributos o miembros de datos que se declaran como protected son accesibles dentro del mismo paquete o sub-clases en paquetes diferentes [5].

Modificador de acceso público o public

El modificador de acceso **public** permite que cualquier clase, método o dato miembro sea accesible desde cualquier parte del código. Cuando se declaran métodos o funciones, atributos o variables como públicos puedes, acceder por dentro de la misma clase, desde cualquier lugar del programa o incluso desde otra biblioteca dependiendo la arquitectura de la aplicación [5].

Los modificadores de acceso son muy importantes, ya que esto permite encapsular a los datos más importantes, sin posibilidad de cambiar su información por valores incorrectos. De igual manera, la información menos sensible es posible compartir sin necesidad de exponer a la prioritaria. En la tabla 2.1 se muestra los tipos de modificadores de acceso en Java y Kotlin.

Tabla 2.1: Modificadores de acceso

Modificador de Acceso	Clase	Paquete	Subclase	Todos
Public	si	si	si	si
Protected	si	si	si	no
Default	si	si	no	no
Private	si	no	no	no

2.3.4 Herencia

La herencia es el proceso en el que un objeto adquiere características de otro objeto. En programación orientada a objetos, la herencia funciona de manera similar que la herencia biológica. En esta última, los hijos adquieren características del padre o de

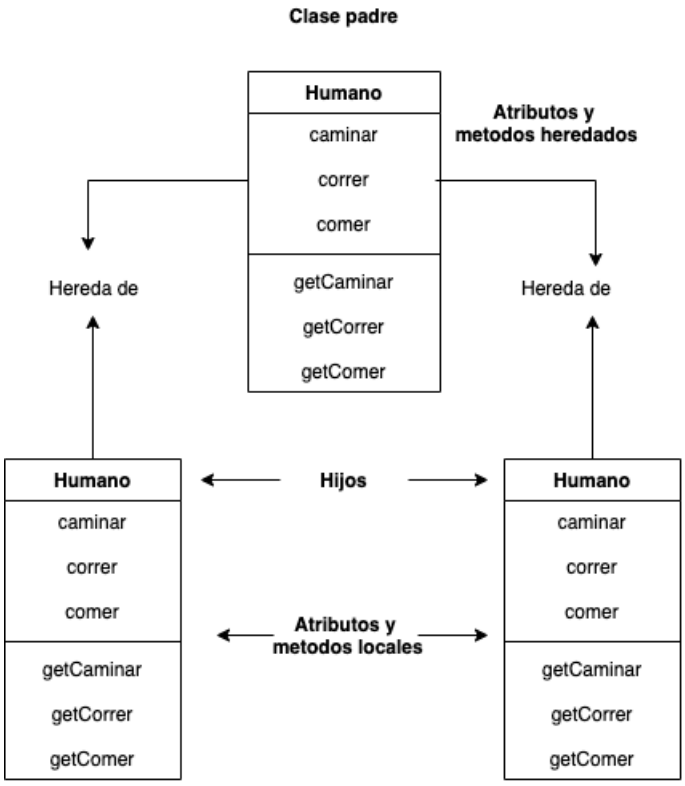

Figura 2.11: Diagrama que muestra el proceso de herencia.

la madre, algunas de estas características son visibles, como los rasgos físicos, mientras que otras no lo son (pero sí se heredan), como por ejemplo, el tipo de sangre.

En programación, las características de un objeto pueden ser empleadas para crear otros objetos en la resolución de problemas. Esto usado comúnmente cuando se intenta hacer un objeto muy similar o con algunas características de algo ya implementado, así nos ahorramos la definición de métodos que ya teníamos escritos o implementados [4].

La Figura 2.11 muestra la representación de la herencia en un diagrama de clase de UML. Se hablará más de este tipo de diagramas en el documento.

2.3.5 Polimorfismo

La palabra polimorfismo se compone de poli(muchas) y morfo (formas), por lo que su significado es “muchas formas”. Personalmente, considero que es un término un poco

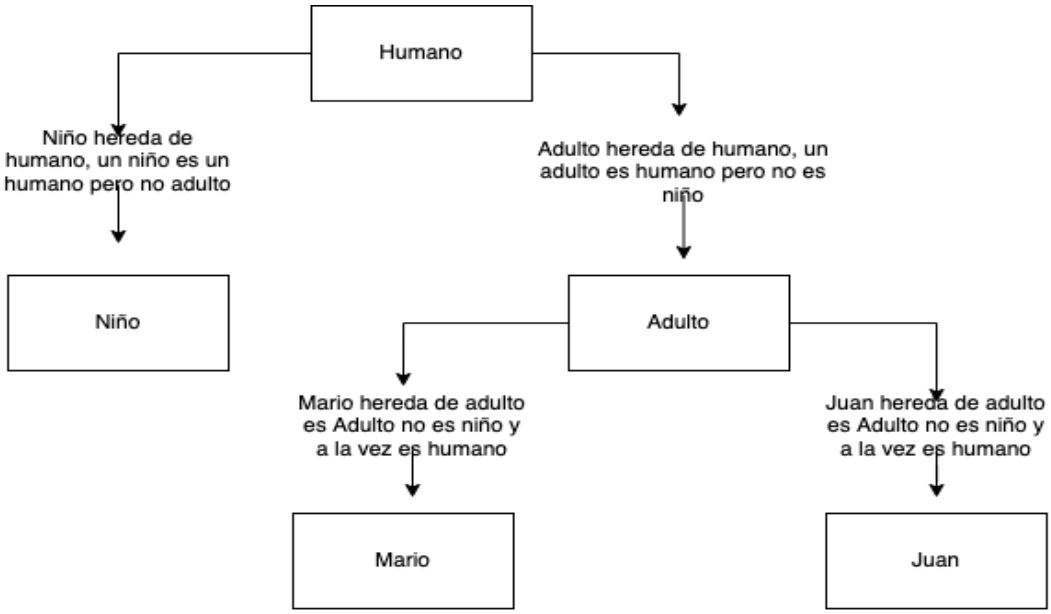

Figura 2.12: Diagrama de clases que muestra el polimorfismo

complejo de entender. La mayoría de las veces se llega a comprender a fondo mediante la experiencia en el desarrollo de programas.

Un ejemplo de polimorfismo es el siguiente. Se Tiene una clase llamada MedioDeTransporte, la cual tiene el método miembro avanza(). Las clases Auto, Camión y Barco heredan de la clase MedioDeTransporte. Es claro que un barco y un auto físico avanzan de forma muy diferente (uno avanza sobre el agua, mientras que el otro sobre el suelo). Por lo tanto, cada una de las subclases de MedioDeTransporte implementa de manera distinta el método avanzar(). La figura 2.12 muestra el diagrama de clases del ejemplo [4].

Ya que los programas escritos en Kotlin pueden correr en la misma maquina virtual que los programas escritos en Java, los fundamentos de programación orientada a objetos son prácticamente los mismos en ambos lenguajes. Considero que esto representa una ventaja para todos los desarrolladores Java con experiencia de años en el desarrollo de aplicaciones para sistema operativo Android, ya que pueden fácilmente adaptarse al nuevo lenguaje de programación oficial (Kotlin).

2.3.6 Beneficios de la Programación Orientada a Objetos

Existen varios paradigmas de programación además del orientado a objetos. Algunos ejemplos de estos otros paradigmas son el estructurado, por procedimientos, funcional y declarativo. La gran mayoría de los sistemas de software modernos son desarrollados usando programación orientada a objetos, debido a sus múltiples beneficios, los cuales se mencionan a continuación:

- El encapsulamiento de los datos miembros, que permite proteger los datos miembros de los objetos, para evitar que éstos últimos estados inválidos.

- El modelado de un problema, a través de la abstracción de objetos independientes pero que interactúan entre ellos para resolver de manera conjunta el problema, permite que un equipo de desarrolladores pueda trabajar de manera asíncrona en la implementación de código.

- La herencia permite reutilizar código fuente que ha sido probado previamente, lo que acelera el desarrollo de software.

- El polimorfismo facilita que los sistemas sean más flexibles, ya que los comportamientos generales de los objetos hacen más específicos los objetos heredados.

La información la imagen 2.13 presenta las diferencias entre la OOP y la programación orientada a procedimientos [13].

Una vez que se se han cubierto los fundamentos de la programación orientada a objetos, nos enfocaremos en describir e ir comparando los lenguajes de programación Java y Kotlin.

OOP	POP
Un paradigma de programación basado en el concepto de objetos, que contienen información almacenada en campos llamados atributos	Un paradigma de programación basado en el concepto de llamadas
Enfasis en objetos	Enfasis en funciones
Divide la programación en múltiples objetos	Divide la programación en múltiples funciones
Modificaciones fáciles de realizar, se realizan en objetos independientes	Las modificaciones son mas difíciles ya que se puede comprometer el funcionamiento del programa
Cada objeto tiene su propia información	Las funciones comparten funciones globales
Se puede ocultar información	No se puede ocultar la información
Se cuenta con accesos específicos	No se cuenta con accesos específicos

Figura 2.13: Diferencias entre Programación Orientada a Objetos y Programación Orientada a Procedimientos

2.4 Lenguajes de programación Java y Kotlin

2.4.1 Inicios de Java y Kotlin

En esta parte de dará un resumen sobre los orígenes de los dos lenguajes de programación que son el objeto principal de este ensayo.

Java

James Gosling se encargó del desarrollo de este lenguaje de programación cuando trabajaba para Sun Microsystems. La empresa Sun Microsystems fue constituida en 1982, pero adquirida en el 2010 por Oracle. Según los datos de Wikipedia (https://es.wikipedia.org/wiki/Sun_Microsystems consultado el 13 de abril 2022) la compra fue por $7,400 millones de dólares. Como un dato curioso, en sus inicios a

Java le dieron el nombre de Oak, pero le cambiaron el nombre debido a que Oak ya era usado por una empresa llamada Oak Technologies. Actualmente esta empresa ya no existe, y el dominio https://www.oak.com/ (consultado el 13 de abril de 2022) pertenece a una empresa que hace desarrollo y servicios de redes de comunicaciones.

Kotlin

La empresa Checa JetBrains –cuyo nombre era inicialmente fue IntelliJ Software– reveló en julio del 2011 un nuevo lenguaje de programación para la maquina virtual de Java (JVM). Su creador, Dmitry Jemerov, tenía una visión amplia de lo que necesitan los lenguajes de programación modernos, y observó algunas características que los actuales lenguajes no cumplen. Uno de los objetivos de Kotlin es que pueda compilar con la misma velocidad que lo hace Java. En febrero de 2012, JetBrains liberó el código fuente del proyecto bajo la Licencia Apache. Google, por su parte, en 2017 lo nombra como el lenguaje oficial de Android.

En mi opinión, Java revolucionó el mercado al ofrecer la promesa "escribe una vez, ejecuta donde sea" (en inglés: WORA Write Once Run Anywere). Esto le permitió llegar a ejecutarse en navegadores web, televisores y otros electrodomésticos, computadoras personales y teléfonos inteligentes. Java tiene cierta similitud con el lenguaje C++ (queda fuera del alcance de este ensayo la comparación entre estos dos lenguajes), haciendo más sencilla la gestión de memoria, eliminando punteros y herencia múltiple, etc. Kotlin, por otro lado, surge mucho después de Java, por lo que la experiencia acumulada en esos años le permite sacar ventaja en su diseño desde el inicio.

2.4.2 Máquina virtual

Una máquina virtual (VM por las siglas en inglés Virtual Machine) permite ejecutar un código máquina a partir de algún lenguaje de programación, en alguna plataforma específica. Una VM interpreta, convierte y ejecuta instrucciones en la plataforma sobre la cual se ejecuta. Esto permite que las aplicaciones generadas sean multiplataforma [1].

En palabras simples, una aplicación multiplataforma es aquella que puede ejecutarse en distintos tipos de máquinas, es decir, en computadoras con diferente arquitectura, con cualquier sistema operativo. Para lograr lo anterior, se requiere que la plataforma (alguna computadora con un sistema operativo) tenga instalado un software especial, la máquina virtual.

Debido a que los compiladores para Java y para Kotlin generan el mismo tipo código máquina (conocido como byte code), no importa en cuál de los dos lenguajes se escriba un programa, el resultado en la ejecución será similar.

En opinión del autor de este documento, Java y Kotlin son similares en cuanto a desempeño para el usuario, ya que se ejecutan en la misma maquina virtual. Además, por experiencia propia, las aplicaciones se perciben igual de rápidas.

2.4.3 Tipos de datos fundamentales

Java cuenta con un conjunto de datos de tipo primitivos, o en otras palabras, tipos de datos fundamentales o incorporados en el lenguaje de programación. Los datos primitivos ayudan a gestionar los tipos de información básicos, como números (enteros o reales) y valores lógicos (verdadero o falso). A continuación se listan los tipos de datos primitivos que se usan tanto para Java como Kotlin.

- Tipos numéricos enteros:
  - **byte:** Utiliza un solo byte (8 bits) de memoria. Almacena valores en el rango [-128, 127]. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Byte**.
  - **short:** Su espacio en memoria es de dos bytes (16 bits), lo cual hace posible representar cualquier valor en el rango [-32.768, 32.767]. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Short**.
  - **int:** Ocupa 4 bytes (32 bits) de memoria, y es el tipo de dato entero más empleado. El rango de valores que puede representar es en el rango -2^{31} a

$2^{31} - 1$. En Kotlin la palabra reservada para declarar una variable de este tipo es **Int**.

- **long:** Su espacio en memoria emplea 8 bytes (64 bits), con un rango de valores desde -2^{63} a $2^{63} - 1$. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Long**.

- Tipos numéricos reales : En programación, los números reales tienen punto decimal y opcionalmente, un exponente.

  - **float:** Es conocido como tipo de dato de precisión simple, emplea un total de 32 bits. los datos que se pueden representar con este tipo de dato son números entre los rango 1.4x10^{-45} a 3.4028235x10^{38}. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Float**.

  - **double:** Es un tipo de dato muy similar a el flotante, pero duplicamos el espacio a usar siendo que ahora usamos 64 bits en lugar de 32 que es de los float. Esto no ayudan a representar números mucho más grande así como mucho más pequeños que van del rango de 4.9x10^{-324} a 1.7976931348623157x10^{308}. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Double**.

- Booleanos y caracteres

  - **boolean:** Nos ayudan a tomar decisiones más cerradas ya que sólo tenemos dos estados true o false. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Boolean**.

  - **char:** Nos ayuda a contener caracteres, ya sean letras o caracteres especiales. En Kotlin, la palabra reservada para declarar una variable de este tipo es **Char**.

En mi opinión, es un acierto el reutilizar los tipos de datos que ya se usan en Java, ya que Kotlin se centra principalmente en optimización de código, el control de excepciones,

programación funcional, etc. los tipos de datos primitivos siempre serán útiles para la programación por que nos facilita las operaciones entre ellos.

La tabla 2.2 muestra un resumen de los tipos de datos en ambos lenguajes.

Tabla 2.2: Resumen de tipos de datos primitivos

Java	**Kotlin**	**Espacio en memoria (en Bytes)**	**Información**
byte	Byte	1	Un byte
short	Short	2	Entero corto
int	Int	4	Entero
long	Long	8	Entero grande
float	Float	4	Real precisión simple
double	Double	8	Real precisión doble
char	Char	1	Un caracter
boolean	Boolean	Depende de la MV	Valor lógico

Los tipos de datos que se mencionaron en la sección anterior se caracterizan por poder almacenar un único valor de un tipo específico. A partir de este reducido conjunto de tipos de datos primitivos, se pueden crear nuevos tipos de datos llamados estructurados que son modificados o accedidos a través de métodos o funciones [13]. A continuación se presenta el tipo de dato estructurado más comúnmente utilizado, las cadenas de caracteres.

Cadenas de caracteres

Para Java una cadena de caracteres es una instancia a la clase String. Esta última es una clase ya incorporada en Java. Tiene varias funcionalidades, como por ejemplo, concatenar caracteres u otras cadenas de caracteres; buscar subcadenas, encontrar el índice de un caracter etc. En muchos casos se piensa que es un dato primitivo, por lo común que es utilizado [13], sin embargo, no lo es.

La forma en que podemos declarar un String es la misma manera de que usamos para declarar una variable de tipo primitivo, en Java se declara como se muestra en código 2.1 [17].

```
1 private String cadena = "Hola mundo";
```

Código 2.1: Ejemplo de declaración de variable String en Java

Para declarar una variable tipo String en Kotlin sólo es necesario igualar la cadena a nuestra variable. Esto es debido a que Kotlin identifica el tipo de dato que se le asigna a una variable, y la declara de manera automática. Un ejemplo de esto se muestra en el código 2.2.

```
1 var cadena = "Hola mundo"
```

Código 2.2: Ejemplo de declaración de variable String en Kotlin

2.4.4 Variables

Una variable es un espacio en memoria principal, el cual es usado para almacenar información. El tamaño que ocupa una variable es distinto, dependiendo el tipo de dato de la variable. El nombre de las variables se le conoce como identificador, el cual debe de seguir ciertas reglas: un identificador no puede empezar con un número, tampoco puede contener espacios, ni caracteres especiales como símbolo de porcentaje, admiración, interrogación, asteriscos, guiones medios, etc. Finalmente, una variable no puede ser una palabra reservada del lenguaje de programación.

Las variables pueden ser locales o de objeto (también llamadas de instancia). Las variables locales son las que se declaran dentro de un método, se crean al invocar al método y se destruyen al regresar de este. Son conocidas como variables automáticas. Por otro lado, las variables de instancia son declaradas dentro de una clase, pero no se encuentra declarada dentro de algún método.

A continuación se muestra la forma de declarar variables de objeto en Java y Kotlin.

Variables en Java

Una de las formas en que podemos declarar las variables en Java se muestra en el siguiente código 2.3:

```
1  private String nombre;
2  public String apellido;
3  protected int edad = 17;
```

Código 2.3: Ejemplo de declaración de variable en Java

En el ejemplo anterior tenemos la estructura básica para declarar variables en Java. La declaración se compone de cuatro elementos: tipo de acceso que puede ser (private, public, protected) o simplemente puede quedar vacío o por defecto; seguido del tipo de dato; el nombre de la variable y opcionalmente un valor inicial de la misma [4]. Opcionalmente, una variable inmutable, es decir, aquella que no cambia de valor una vez asignada, se declara usando como prefijo la palabra reservada **final**.

Variables en Kotlin

En Kotlin, se hace necesario el uso de dos palabras clave para declarar variables, estas son **val** y **var**. El primero, val, se usa para una variable cuyo valor no cambia nunca (inmutable) [3], es decir, para declarar constantes. No se puede volver a asignar un valor a una variable que se declaró mediante val. Por otra parte, se usa var para una variable cuyo valor puede cambiar (mutable), ahora veamos un ejemplo de como se declaran las variables en Kotlin en el siguiente código 2.4 [21]:

```
1     private var nombre : Int = 10;
2     private val apellido : String;
```

Código 2.4: Ejemplo de declaración de variable en Kotlin

En el ejemplo anterior se muestra como se declara dos tipos de variables un inmutable y la otra mutable, respectivamente. Como se observa iniciamos con el tipo de acceso, el tipo de variable mutable o inmutable, el nombre de la variable, el tipo de dato y para variable inmutable finalizamos con un valor por defecto que en todo el tiempo de ejecución mantendrá ese valor [18].

Declarar variables tanto para Kotlin como para Java es muy similar, todos lo lenguajes de programación necesitan variables y en este caso Kotlin no parece innovar, en mi

opinión no tenemos ahorro de sintaxis, sólo lo hace de una manera diferente.

2.4.5 Arrays o vectores

Los vectores son colecciones de datos de un mismo tipo. También se le conoce como arrays, o como "arreglos" en español. Un vector es una estructura de datos en la que a cada elemento le corresponde una posición identificada por uno o más índices numéricos enteros.

No sólo en la programación estructurada se hace el uso de los arrays, uno de los conceptos que se tiene es que es almacenamiento contiguo de elementos del mismo tipo de dato. Estos arrays son considerados como matrices de 1 x n [13].

Arrays en Java

En Java, sólo existen arrays dinámicos, es decir, que se crean en tiempo de ejecución. existe varias formas de declarar arrays dinámico o estáticos.

Para declarar un array, es necesario agregar el tipo de dato, seguido del nombre y corchetes, se asigna uno valor o más valores que van cerrados entre llaves. Cada elemento es separado por comas. La longitud del array es definida por el número de elementos indicado [13] [21]. En el código 2.5 se muestra un ejemplo de lo explicado.

```
1    int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10};
```

Código 2.5: Ejemplo de declaración de arreglo en Java

Otra forma de declarar un array en Java es indicar el número de elementos que contendrá. Posteriormente se llenará con valores correspondientes. En el código 2.6 se muestra la manera de realizar esto.

```
1     int intArray[] = new int[9];
```

Código 2.6: Segundo ejemplo de declaración de arreglo en Java

Existen estructuras de datos con más funcionalidades que los arreglos. Un ejemplo de ellas son las listas enlazadas. Para definir una lista enlazada genérica es necesario agregar el tipo de dato, seguido del nombre y creamos una instancia de ArrayList y para agregar un elemento a mi Array usamos la función **add()** se muestra en el siguiente código 2.7.

```
// Declaramos el Arreglo dinamico

      ArrayList<Integer> intArray = new ArrayList();

// Agregamos datos
       intArray.add(1)
       intArray.add(2)
       intArray.add(3)
       intArray.add(4)

// Ahora tenemos un Array de longitud 4
```

Código 2.7: Ejemplo de declaración de Arrays dinámicos en Java

Para recorrer un arreglo o un ArrayList, puede realizar de la siguiente manera: se accede a cada elemento del array mediante un índice de tipo entero. El primer elemento tiene como índice o posición el 0 (cero), y el último elemento tiene índice o posición N-1, siendo N la longitud total. El código 2.8 muestra la forma de acceder a los elementos de un arreglo.

```
// Declaramos el Arreglo estatico o dinamico
  ArrayList<Integer> intArray = new ArrayList();

// Agregamos datos
       intArray.add(1)
       intArray.add(2)
       intArray.add(3)
       intArray.add(4)

// Recorremos el arreglo con un for
    for(int i = 0 ; intArray.length()>i ; i++) {

       System.out.println(intArray[i])

    }

// Salidas
    1
    2
    3
    4
```

Código 2.8: Como acceder a los elementos de un arreglo en Java

Arrays o vectores en Kotlin

Para definir un arreglo en Kotlin, es necesario usar la palabra reservada **arrayOf**, y en lugar de los corchetes que se usan en Java, se usan paréntesis triangulares, es decir, signos "menor que" (<) y "mayor que" (>). Se especifica el tipo de dato, y entre paréntesis se escriben los valores iniciales [3]. El código 2.9 muestra un ejemplo de declaración de un arreglo en Kotlin.

```
1     val intArray = arrayOf<Int>(1,2,3,4,5,6,7,8,9,0)
```

Código 2.9: Ejemplo de declaración de Arrays Kotlin

Otra forma de declarar un array en Kotlin es escribir el número de elementos que contendrá. En el codigo 2.10 usamos un constructor **arrayOfNulls¡Int¿(10)**, con esto delimitamos nuestro arreglo a 10 elementos.

```
1   val intArray = arrayOfNulls<Int>(10)
```

Código 2.10: Ejemplo de declaración de arreglo en Kotlin

Un ejemplo de declaración de arreglo en Kotlin se muestra en el código 2.11.

```
1  // Creamos array dinamico
2
3     val intArray = arrayListOf<Int>()
4
5  // Agregamos datos
6
7     intArray.add(1)
8     intArray.add(2)
9     intArray.add(3)
10    intArray.add(4)
11
12 // Ahora tenemos un Array de longitud 4
13 // Para kotlin podemos inicializar de otra manera mas practica
```

Código 2.11: Ejemplo de declaración de arreglo en Kotlin

Si queremos imprimir todos los elementos en un array, no necesitamos usar un ciclo, la variable como tal tiene la propiedad de imprimir todo el array [19], como se muestra en el código 2.12.

```
// Creamos array dinamico

  val intArray = arrayListOf<Int>()

// Agregamos datos
   intArray.add(1)
   intArray.add(2)
   intArray.add(3)
   intArray.add(4)

// imprimimos la informacion del array
   println(intArray.contentToString())
```

Código 2.12: Ejemplo de Imprimir arreglo en Kotlin

Para acceder a los elementos de un array de forma individual, es de la misma forma como se usa en Java, como se presenta en el código 2.13 :

```
// Creamos array dinamico

   val intArray = arrayListOf<Int>()

// Agregamos datos

   intArray.add(1)
   intArray.add(2)
   intArray.add(3)
   intArray.add(4)

// imprimimos la informacion del elemento seleccionado

   println(intArray[0])
   println(intArray[1])
   println(intArray[2])
   println(intArray[3])

// el resultado es el siguiente

   1
   2
   3
   4
```

Código 2.13: Ejemplo de declaración de Array dinamico en Kotlin

Los arreglos en la OOP nos ayudan mucho para almacenar datos del mismo tipo,

recorrerlos, eliminar datos de ellos etc. Bajo mi punto de vista, Kotlin tiene formas más interesantes de acceder y realizar operaciones con estos, sus clases orientadas a arrays son más completas que en Java. lo que nos ayuda a resolver problemas de listas, matrices, arreglos, etc., de muchas maneras más simples.

2.4.6 Clases en Kotlin y Java

Las clase pueden ser consideradas como una plantilla, molde o modelo para crear nuevos objetos. Es uno de los más grandes paradigmas en el desarrollo de software en el día de hoy. Una clase está compuesta por características, propiedades o atributos (variables de instancia), y por su comportamiento (métodos) que trabajan sobre las propiedades [19].

Una de las diferencias entre una clase de Kotlin y una de Java, se presenta al acceder y establecer el valor a un atributo. Mientras que en Java se hace el uso de métodos conocidos como getters y setters (get para obtener el valor y set para asignar valores), en Kotlin con declarar el tipo de atributo se puede acceder a su valor de forma directa.

Clases en Kotlin

Los constructores se usan para inicializar variables. Los constructores en las clases declaran de diferente manera en Java que en Kotlin.

Para el ejemplo mostrado en el código 2.14, algunos parámetros no son obligatorios, es decir, se pueden establecer valores predeterminados [19]. Esto facilita para cuando no se requiere pasar todos los valores de los parámetros.

```
1 // Constructor en Kotlin
2
3 class Persona(
4   private var nombre: String?,
5   private var apellido: String? = "",
6   private var edad: Int
7 )
```

Código 2.14: Ejemplo de clase en Kotlin

Como se puede observar, en el constructor podemos definir la variable ya sean privada o pública del cualquier tipo de dato, nulas o no nulas, obligatorias o no obligatorias.

Clase en Java

Las clases en Java pueden tener datos miembro los cuales pueden ser inicializadas con valores por defecto.

En el constructor se pasan los argumentos necesarios para que se inicialicen los datos miembros cada vez que se crea una nueva instancia de la clase, es decir, cada vez que se crea un objeto. Para acceder al valor de los atributos se deben de crear métodos getters y setters. Esto último para implementar el concepto de encapsulación. En el código 2.15 podemos observa un ejemplo de la declaración de una clase en Java .

```
1  // Clase en Java
2
3  public class Persona {
4
5    private String nombre = "";
6    private String apellido = "";
7
8      public Persona(String nombre) {
9          this.nombre = nombre;
10     }
11
12   public String getNombre() {
13     return nombre;
14   }
15
16   public void setNombre(String nombre) {
17     this.nombre = nombre;
18   }
19
20   public String getApellido() {
21     return apellido;
22   }
23
24   public void setApellido(String apellido) {
25     this.apellido = apellido;
26   }
27 }
```

Código 2.15: Ejemplo de clase en Java

He observado que en Kotlin la declaración de las clases usa un menor número de lineas de código comparado con Java. Además, da la impresión de ser un código más

limpio. Esto ayuda a comprender más fácilmente el código de otras personas, y que otras personas entiendan mejor nuestro código.

Constructores en Java y en Kotlin

Es posible que en los ejemplos anteriores se observara que no había ningún constructor a la vista, esto es por que una clase en Java siempre tiene un constructor [22]. Si no se crea de manera explícita un constructor, entonces se crea uno implícito pero vacío. Por defecto, este tipo de constructor no acepta parámetros. Un ejemplo de clase con un constructor explícito vacío se muestra en el código 2.16. Este constructor es el que se crea de manera automática si no se declara uno explicitamente.

```
1  // Clase en Java
2
3  public class Persona {
4
5      private String nombre = "";
6
7      public Persona() {
8
9      }
10 }
```

Código 2.16: Ejemplo de clase en Java con constructor

Como se puede observar, la forma en que se declara un constructor en Java es con el mismo nombre de la clase, especificando el tipo de acceso. Ahora vamos a ver como se crea un constructor con parámetros en el siguiente código 2.17

```
1  // Clase en Java
2
3  public class Persona {
4
5      private String nombre = ""
6
7      public Persona(String nombre) {
8          this.nombre = nombre;
9      }
10
11 }
```

Código 2.17: Ejemplo de clase en Java con constructor con parámetros

Cada vez que se instancia una clase, se debe pasar un parámetro de tipo String. Esto permite tener al nuevo objeto sus variables inicializadas desde que se crea [22]. Se pueden tener número ilimitado de constructores, la única condición es que no sean iguales en el número u orden de parámetros. Un ejemplo de esto se presenta en el código 2.18. En este caso, se está aplicando el concepto de sobrecarga, que significa que un mismo método puede tener varias versiones, cada una de las cuales difiere de las otras en cuanto al número y/o tipo de parámetros.

```
// Clase en Java

public class Persona {

  private String nombre = "";
  private String apellido = "";


    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public Persona(String nombre, String apellido) {
        this.nombre = nombre;
        this.apellido = apellido;

    }

    .
    .
    .
}
```

Código 2.18: Ejemplo de clase en Java con 2 constructores

Constructor Kotlin

Kotlin, al igual que Java, cuenta con métodos constructores en sus clases, que nos ayudan a inicializar las variables de los objetos. A diferencia de este Java, Kotlin lo hace de una forma que ayuda a ahorramos líneas de código. Esto lo logra cambiando la metodología, y no define el constructor principal como un método dentro de la función embebido dentro de la clase, como se muestra en el ejemplo 2.19 [20]

```kotlin
// Clase en Kotlin

class Persona(
  var nombre : String?,
  var apellido : String?
)
```

Código 2.19: Ejemplo de clase Kotlin con constructor

En el ejemplo pasado observamos que como la clase persona tiene un constructor embebido, al mismo tiempo que definimos las variables estamos obligando a que se pasen los parámetros para nombre y para apellido, esto nos ayuda a no escribir un método para definir el constructor como en el caso de Java que creamos una función pública con el nombre de la clase de igual manera ya estamos inicializando todas las variables de la clase, si bien en Java podemos crear constructores donde se especifique que variables deben ir inicializadas en Kotlin podemos igual definir que variables son obligatorias inicializar como en el siguiente ejemplo 2.20 [24]:

```kotlin
// Clase en Kotlin

class Persona(
  var nombre : String?,
  var apellido : String?
)
```

Código 2.20: Ejemplo de clase Kotlin con constructor con variables no obligatorias

La variable **var nombre : String?** es de tipo no requerido el símbolo **?** nos ayuda a declarar variables de tipo, son muy útiles cuando no se sabe si el dato que se esta esperando llegara o no, con esto se evita el error de Null Exception, el siguiente ejemplo muestra como se declara variables con un valor por defecto 2.21 [14]

```kotlin
// Clase en Kotlin

class Persona(
  var nombre : String = "",
  var apellido String = ""

)
```

Código 2.21: Ejemplo de clase Kotlin con constructor con variables definidas

En la mayoría de los casos los constructores nos ayudan a inicializar variables, Kotlin es quien mejor innova ya que podemos ahorrar lineas de código, esto se logra gracias a que simplifica la forma en como declaramos las variables, a la vez tenemos clases más limpias y ordenadas.

Objetos en Java y en Kotlin

Un objeto es una instancia de una clase, una instancia es un ejemplar, un caso específico de una clase, Se pueden crear varios objetos de la misma clase y cada objeto no comparte la misma información, sólo atributos y comportamientos [2]. Instancia es cuando declaramos un objeto de cualquier clase. Los atributos de un objeto son las variables definidas en la clase y el comportamiento es definido por sus métodos de esta misma. A continuación se muestra la forma en que se declaran objetos tanto para Kotlin como para Java.

Java

A continuación se muestra en el código 2.22 como se declara un objeto en Java de la clase Persona y como se hace el uso de sus funciones [6].

```
1 // Clase en Java
2    Persona personaObj = new Persona ()
3
4 // declaramos el una variable de tipo String
5
6     String contieneString = " "
```

Código 2.22: Ejemplo de creación de objeto en java

Las clases que contienen un constructor y a su vez este contiene parámetros se declara de la siguiente manera en el código. 2.23

```
1
2     Persona personaObj = new Persona("Juan", "Sanchez");
```

Código 2.23: Ejemplo objeto en Java pasando parámetros al momento de instanciar la clase

Para asignar el valor a la variable hacemos el uso de la función setNombre(), esta función recibe parámetros de tipo String, un buen ejemplo lo podemos apreciar en el código 2.24

```
// Declaramos un objeto tipo Persona

    Persona personaObj = new Persona("Juan", contieneString);

// Accedemos a las funciones de los objetos a trav s de la
   siguiente manera

    personaObj.setNombre(contieneString)
```

Código 2.24: Ejemplo del uso de las funciones de un objeto en Java y asignando valores

El objeto personaObj almacena valores en sus atributos y pueden ser accedidos mediante la funciones, una de ellas es la función .getNombre() y este a su vez puede ser asignado a una variable de tipo String, en el siguiente código se muestra un ejemplo 2.25.

```
// Declaramos un objeto tipo Persona

    Persona personaObj = new Persona("Juan", contieneString);

    contieneString = personaObj.getNombre()
```

Código 2.25: Ejemplo del uso de las funciones de un objeto en Java y obtener valores del objeto creado

2.4.7 Objeto en Kotlin

Se muestra en el siguiente código la clase Persona en Kotlin 2.26 la cual se utiliza para los ejemplos por venir. [15].La clase persona que no tiene parámetros obligatorio en el constructor.

```
// Clase en Kotlin

class Persona(
  var nombre : String?,
  var apellido : String?
)
```

Código 2.26: Ejemplo clase en Kotlin

Creamos un objeto de tipo Persona y le asignamos algunos valores ejemplo en el código 2.27

```
// Crear objeto

    var personaObj = Persona()
    personajeObj.nombre = "Juan"
```

Código 2.27: Ejemplo como crear objeto en Kotlin

El siguiente ejemplo muestra una clase con parámetros obligatorios en el constructor, para crear un objeto en Kotlin para este tipo de clases realizamos lo que se muestra en el código 2.28

```

    var personaObj = Persona("Juan", "Perez")
```

Código 2.28: Ejemplo como crear objeto en Kotlin y paso de parámetros en constructor

Usualmente los objetos son empleados en la programación orientada a objetos, no todos los lenguajes orientados a objetos se les llame objetos, intentan solucionar los problemas de una manera muy similar, Kotlin intenta simplificar el uso de ellos, pero en mi opinión el uso de estos en Java y Kotlin es muy similar.

2.5 Sentencias:

Los programas de Java se ejecuta por lineas ordenadas, a estas lineas de código se les llaman sentencias, Existen sentencias de control de flujo este tipo de sentencias nos permiten alterar el flujo de ejecución para tomar decisiones o repetir sentencias [6]. Dentro de las sentencias de control de flujo tenemos las siguientes:

2.5.1 Sentencia de decisión Java:

Las sentencias de decisión nos permiten ejecutar un nuevo bloque de sentencias o simplemente no realizar ninguna acción, algunos ejemplo de sentencias de decisión son: if-then-else y switch, Mediante if-then-else podremos evaluar una decisión y elegir por un bloque u otro [6].

ejemplo de sentencia de decisión en el fragmento de código 2.29:

```
1     if(string.isEmpty()) {
2           .
3           //Sentencias
4           .
5           .
6           .
7
8     }
```

Código 2.29: Ejemplo del uso IF en java

- if - es la sentencia de decisión.
- String.isEmpty() - es el que define la ruta que va a tomar las siguientes sentencias.
- Sentencias - Es la sentencia siguiente a ejecutar una vez se cumpla la sentencia dentro del if.

2.5.2 Sentencia de decisión Kotlin:

Kotlin tiene todas las sentencias de decisión que existen en Java, Kotlin anexa una nueva sentencia de control, ayuda en la toma de decisiones de una manera más organizada.**when** es su palabra reservada y en el código 2.30: se muestra una parte de su funcionamiento [16].:

```
1 // When en Kotlin
2
3     when {
4         isVariableBool1 -> openFunction1()
```

```
        isVariableBool2 -> openFunction2()
        isVariableBool3 -> openFunction3()
    }
```

Código 2.30: Ejemplo de como se usa when en Kotlin sin variable a evaluar

En el código 2.30 ejemplifica como hacer toma de decisiones con Booleanos, al igual que otras sentencias de decisión **when** pueden hacer uso de otras comparaciones para cumplir una sentencia, en el siguiente ejemplo de código 2.31 se muestra otros ejemplos para **when**.

```
// When en Kotlin
    var variable : String = "OK"
    const val CONSTANT1 = "OK"
    const val CONSTANT2 = "FAILD"
    const val CONSTANT3 = "SUCCES"

    when (variable) {
        CONSTANT1  -> openFunction1()
        CONSTANT2 -> openFunction2()
        CONSTANT3 -> openFunction3()
    }
```

Código 2.31: Ejemplo de como se usa when en Kotlin con variable a evaluar

En el siguiente ejemplo se observa que **when** hace el uso de constantes para evaluar las condiciones, como funciona esto, internamente hace la comparación de la **variable** con las **CONSTANT** esto es similar a esto **variable == CONSTANT1** si no cumple hace la siguiente comparación **variable == CONSTANT2** y final mente **variable == CONSTANT3** en caso de que ningún caso coincida no entra a ningún flujo, en el siguiente código se muestra lo explicado anteriormente 2.32 [16].

```
// When en Kotlin
    var variable : String = "OK"
    const val CONSTANT1 = "OK"
    const val CONSTANT2 = "FAILD"
    const val CONSTANT3 = "SUCCES"

    when (variable) {
        CONSTANT1  -> openFunction1()
        CONSTANT2 -> openFunction2()
        CONSTANT3 -> openFunction3()
        else / default ->
    }
```

Código 2.32: Ejemplo de como se usa when en Kotlin con variable a evaluar

Por último como ya es común tenemos palabras reservadas **else** y **default** ambas nos ayudan a controlar el flujo en caso de que las sentencias no se cumplan [8].

La toma de decisiones son muy importantes en los lenguajes de programación, no importa que tipo de lenguaje estés usando hasta el más básico lo ha implementado. Kotlin con la integración de **when** ayuda a mantener un código más limpio y entendible, ayuda a ahorrar lineas de código y mantenemos el Single responsibility (Una sola responsabilidad) uno de los principios de **SOLID**.

2.5.3 Sentencias de bucle Java:

Nos ayudan a ejecutar un bloque de sentencias tantas veces sea necesario, algunas sentencias de bucle están formadas por sentencias de decisión, estas determinan el tiempo que el bucle se ejecute, cuando se cumple la condición se finaliza el bucle, algunos ejemplos de sentencias de bucle son while, do-while y for, etc... en el código 2.33 observamos un ejemplo de for.

```
1    for(int i = 0;i <= 5;i++) {
2         .
3         .
4         //Sentencias
5         .
6         .
7         .
8    }
```

Código 2.33: Ejemplo de for en java

2.5.4 Sentencias de bucle Kotlin:

Kotlin hace el uso de **for** de una manera muy similar a Java, en este caso sólo vemos una diferencia en la forma en que se define, no se define una variable y simplifica el incremento o tope, el uso es igual tenemos una variable que se va incrementando, se

inicializa y se ejecuta n veces, el siguiente ejemplo 2.34 nos ayuda a comprender más sobre el for de Kotlin.

```
1     for(i in 1..5) {
2         println("Contando $i")
3     }
```

Código 2.34: Ejemplo de for en Kotlin

El código 2.34 imprime el siguiente resultado **Contando 1**, **Contando 2**, **Contando 3**, **Contando 4**, **Contando 5**. Existe muchas maneras de realizar recorridos con el for de Kotlin, se puede recorrer caracteres como se muestra en el siguiente ejemplo 2.35

```
1  // iteracion regular
2      for (char in 'a'..'f') print(char)
3
4  // iteracion con avance de 2
5      for (char in 'a'..'f' step 2) print(char)
6
7  // iteracion en reversa
8      for (char in 'f' downTo 'a') print(char)
9
10 // iteracion excluyendo el limite superior
11     for (char in 'a' until 'f') print(char)
```

Código 2.35: Ejemplo de for en Kotlin diferentes tipos de incrementos

La impresión de estos caracteres es el siguiente **abcdef ace fedcba abcde**.

Los nuevos lenguajes de programación son inspirados en lenguajes de programación viejos, el uso de **for** siempre sera necesario, la forma en que se usan no ha cambiado mucho desde sus inicios, Kotlin lo implementa de una manera más simple y entendible.

2.5.5 Programación funcional en Kotlin:

En Kotlin existe la programación funcional que es nativa de este lenguaje, nos ayuda a trabajar con colecciones de datos, algunos ejemplos para la programación funcional son los siguientes:

2.5.6 Apply (Aplicar)

Como su nombre indica aplicar información de un objeto a variables o campos específicos, el ejemplo siguiente 2.36 muestra como en Java se aplican datos.

```
// Creamos una variable con algunos atributos por default

    val circulo = Circulo(4, "Azul", circulo2)

//Agregamos valores a nuestras variables


    var radio = circulo.radio
    var color = circulo.color
    var nombre = circulo.nombre
```

Código 2.36: Ejemplo como asignar valores directamente desde el objeto en Kotlin

El ejemplo del código 2.36 hace referencia a una práctica muy común en Java, Kotlin nos ayuda a tener un código más organizado, el siguiente ejemplo nos ayuda a aplicar información de una más ordenada y entendible 2.37

```
// Creamos una variable con algunos atributos por default

    val circulo = Circulo(4, "Azul", circulo2)

//Agregamos valores a nuestras variables aplicando apply

    circulo.applay {
        var radio = radio
        var color = color
        var nombre = nombre
    }
```

Código 2.37: Ejemplo de como usar applay en Kotlin

Como se muestra en el código 2.37 se esta aplicando la información del objeto Circulo a nuestras variables, nos ayuda a organizar y delimitar que información viene de dichos objetos [11].

2.5.7 let

Kotlin cuenta con **let**, que nos ayuda a verificar información que tiene una variable no sea nula, es como si tuviera una condicional antes de ejecutar dicha sentencia, el

siguiente código nos ayuda a comprender más la función de **let** 2.38 [24].

```
// Creamos una variable que tomara numeros aleatorios del 0 a el
    10

repeat(10) { cuenta ->
    val variableAleatoriaNula = when (Random.nextBoolean()) {
        false -> null
        true -> "not null"
    }
    variableAleatoriaNula?.let { println("$cuenta $it") }
}
```

Código 2.38: Ejemplo de como usar let en Kotlin

El código 2.37 sólo muestra números aleatorios que no sean nulos y serán impresos.**let** ayuda a reducir lineas de código manteniendo la filosofía de código limpio.

2.5.8 map (mapa)

map para las listas son muy útiles ya que no es necesario agregar bucles para recorrer la información o abstraerla de la lista, el siguiente código muestra un ejemplo de map. 2.39 [24].

```
 val lugares = listOf(
        CiudadTuristica("Torre Eiffel", "Francia"),
        CiudadTuristica("Coliseo Romano", "Italia"),
        CiudadTuristica("Acropolis de Atenas", "Grecia"),
        CiudadTuristica("Taj Mahal", "India")
    )

println(lugares.map { it.nombre })
```

Código 2.39: Ejemplo de como usar map en Kotlin

El código 2.39 se observa que gracias a map podemos imprimir la propiedad nombre de nuestra lista lugares, igual se observa que no es necesario alguna propiedad que nos ayude a recorrer esta misma.

2.5.9 it

En Kotlin es muy común ver la palabra reservada **it**, es una expresión lambda que igual es usada en Java, es un parámetro unico implícito, en el código 2.39 hacemos uso de esta variable, en la linea que donde se usa lugares.map it es un objeto de tipo CiudadTuristica y esta puede acceder a todos lo elementos de este objeto, en el ejemplo se muestra de manera implícita pero igual se puede definir de la siguiente manera **lugares.map it -¿** , igual se puede renombrar **lugares.map objetoCiudadTuristica -¿** en el siguiente código podemos ver el ejemplo más claro 2.40 [24].

```
1 val lugares = listOf(
2         CiudadTuristica("Torre Eiffel", "Francia"),
3         CiudadTuristica("Coliseo Romano", "Italia"),
4         CiudadTuristica("Acropolis de Atenas", "Grecia"),
5         CiudadTuristica("Taj Mahal", "India")
6     )
7
8 println(lugares.map { objetoCiudadTuristica ->
    objetoCiudadTuristica.nombre })
```

Código 2.40: Ejemplo de como usar it en Kotlin

Aunque en Java ya se cuenta con este tipo de programación, es donde Kotlin empieza a resaltar como lenguaje de última generación, así como la OOP nos ayudo a separar los problemas en objetos e innovo en este aspecto ahora Kotlin nos ayuda mantener un orden, limpieza y escalabilidad de código.

2.6 Interfaces

Es una clase donde se puede especificar uno o más métodos que no tienen cuerpo o no están definidos. Esos métodos deben ser implementados por una clase para que se definan sus acciones. Una interfaz especifica como realizar la tarea con el nombre de la función, pero no cómo hacerlo. Una vez que se define una interfaz cualquier cantidad de clases puede implementarla. Una clase puede implementar cualquier cantidad de interfaces.

2.6.1 Implementar Interfaces

Para implementar una interfaz, una clase debe proporcionar cuerpos para los métodos descritos por la interfaz. Cada clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero cada clase aún admite el mismo conjunto de métodos.

2.6.2 Interfaz Java

Las interfaces en Java usan la palabra reservada **interface** quien ayuda a definir que se esta creando una interfaz, seguida del nombre de la interface, el siguiente ejemplo se muestra como se crea una interface2.41 [24].:

```
// Interfaz en Java

public interface GeometricsFigures {

    float area(float lado1, float lado2 . . . . etc);
    String nameFigure(String nombre);

    float perimeter(float lado1, float lado2 . . . . etc);
    int numberSides(String evalueName);

}
```

Código 2.41: Ejemplo creación de una interface en java

Para poder extender esta interfaz en la siguiente clase 2.42 Rectangle, se agrega la palabra **implements** al final de la definición de la clase seguida del nombre de la interface previamente creada, el siguiente código muestra como se realiza la implementación 2.43.

```
// Clase en Java sin implementar interfaz

public class Rectangle {

  private String nombre = "";
  String numeberSides = "";
    ..
    .
}
```

Código 2.42: Clase ejemplo para implementar nuestra interface en java

```
// Clase en Java implementando interface

public class Rectangle implements GeometricsFigures {
  private String nombre = " "
  String numeberSides = " "
  ..
  .
}
```

Código 2.43: Ejemplo de implementación de interface en Java en la clase Rectangle

Se muestra en el siguiente código 2.44 como ahora se están definiendo las funciones en nuestra clase **Rectangle**, esto ayuda a definir el comportamiento de una clase

```
public class Rectangle implements GeometricsFigures {

  private String nombre = "";
  String numeberSides = "";

    float area(float lado1, float lado2 . . . . etc) {
        ..
        .
    }

    String nameFigure(String nombre) {
        ..
        .
    }

    float perimeter(float lado1, float lado2 . . . . etc) {
        ..
        .
    }

    int numberSides(String evalueName) {
        ..
        .
    }
}
```

Código 2.44: Ejemplo de Definiendo el comportamiento de las funciones que se heredaron de la interface en java

Conclusión :

- 1.-) La clase que implementa la interfaz necesita proporcionar funcionalidad para los métodos declarados en la interfaz.

- 2.-) Todos los métodos en una interfaz son implícitamente públicos y abstractos.
- 3.-) Una interfaz no puede ser instanciada.
- 4.-) Una interfaz puede extenderse desde una o varias interfaces. Una clase puede extender una sola clase pero implementar cualquier cantidad de interfaces.

2.6.3 Interfaz Kotlin

Las interfaces para Kotlin pueden tener código es decir las funciones están definidas. Esto ayuda a implementar un especie de herencia múltiple. Al igual que Java podemos hacer que una clase implemente varias interfaces, y que herede el comportamiento de cada una de ellas. en el siguiente código 2.46 se muestra como crear interface en Kotlin y en el siguiente código creamos una segunda con funciones definidas2.45.

```
1 // Interface Kotlin
2
3     interface Interface1 {
4         fun getMessage() : String
5         fun isShowMessage = false
6     }
```

Código 2.45: Ejemplo de interface en Kotlin interface 1

```
1 // Interface Kotlin
2
3     interface Interface2 {
4         fun isShowView() = Boolean
5     }
```

Código 2.46: Ejemplo de interface en Kotlin interface 2

Para poder implementar las interfaces en nuestra clase Kotlin ya no es necesario la palabra reservada implements, ahora con sólo colocar ":" al final de la declaración de la clase el código ya sabe que estamos implementando una interfaz

Para el siguiente ejemplo se muestra implementadas las funciones getMessage() y isShowView(), getMessage() corresponde a la Interface1 e isShowView() corresponde a la Interface2, si observamos no se esta implementando la función isShowMessage() la

razón es que ya tenemos definido un valor por defecto y no es necesario asignar un valor de retorno o sobre escribir la función. pero el valor que siempre va a tomar esa función es **false**, el siguiente código muestra dicho comportamiento 2.47.

```
// implementar Interface Kotlin

class MyClass : Interface1, Interface2 {

    override fun getMessage() : String {
        return "string"
    }

    override fun isShowView() = false

}
```

Código 2.47: Ejemplo de interface en Kotlin implementada en una clase

En caso que se requiera darle cuerpo a la función podemos hacerlo de la misma manera que se hace para las otras funciones que se implementan como se muestra en el código 2.48.

```
// Interface Kotlin

class MyClass : Interface1, Interface2 {

    override fun getMessage() : String {
        return "string"
    }

    override fun isShowView() = false

    override fun isShowMessage(): Boolean {
        return Objerto.isShowObjetc()
    }
}
```

Código 2.48: Ejemplo de interface en Kotlin implementada en una clase y definiendo el comportamiento de las funciones

Las interfaces en Kotlin ayudan a la reutilización de código más de la que se conseguía con Java, la razón es por que se puede añadir código a tus interfaces. Si bien en Java 8 ya venia usando de esta manera. para Java 6 sólo podemos definir el comportamiento pero no implementarlo.

2.7 Proceso para crear aplicaciones Android

Una aplicación para dispositivos móviles consiste en procesos que son ejecutados en teléfonos inteligentes, tabletas y otros dispositivos móviles. Este tipo de aplicaciones permiten al usuario efectuar un variado conjunto de tareas. El objetivo de una aplicación es ejecutar procesos con interfaces intuitivas para el usuario. Un ejemplo muy conocido de este tipo de aplicaciones es Google maps, que ayuda a localizar direcciones, da instrucciones como llegar, ofrece recomendaciones de lugares, y visitar virtualmente otros países, etc. Otro ejemplo es la calculadora, que permite realizar cálculos aritméticos o incluso de funciones científicas.

Es muy importante mencionar que las aplicaciones para dispositivos móviles están remplazando a una gran variedad de dispositivos digitales (llamados Gadgets), como a los reproductores de música, cámaras, grabadoras de audio y/o video o incluso flexómetros. Ahora, en la palma de nuestra mano tenemos el poder de realizar llamadas, leer libros, encontrar gente, ver vídeos, etc. cuando no hace mucho tiempo usábamos varios dispositivos diferentes para ello.

2.7.1 Proceso general para la creación de una aplicación

El desarrollo de una aplicación tanto en Java como Kotlin es muy similar, algo que tenemos que definir primero es que todas la aplicaciones corren en Android por medio de pantallas, las cuales en código se denominan Activity.

Elección de dispositivo y versión del sistema operativo

Para poder verificar si el problema que se quiere resolver usando un dispositivo Android se deben de tomar en cuenta algunas consideraciones y limitaciones. Estas se enuncian a continuación:

- Verificar si el dispositivo Android cuenta con la tecnología para poder resolver el

problema. Este punto es muy importante, ya que las aplicaciones para móviles tienen una limitante tanto como en procesamiento, almacenamiento, resolución de pantalla, conectividad.

- Para cuál tipo de dispositivo Android se requiere realizar la aplicación, se tienen como principales opciones las televisiones inteligentes, los teléfonos, tablets, relojes inteligentes, etc.

- Versión de sistema operativo Android. Esto es porque en algunas características sólo se encuentran disponibles en ciertas versiones de Android. También es importante considerar que sólo los dispositivos más recientes podrán ejecutar las últimas versiones del sistema operativo.

Todas estas limitantes es necesario tener en mente para resolver un problema, con el objetivo de que la aplicación a desarrollar funcione apropiadamente.

UX y UI

Hay dos conceptos que se usan mucho en el desarrollo de aplicaciones para dispositivos móviles: UX y UI. A continuación se explicará brevemente la diferencia entre ellos.

UX (User Experience) son las métricas de las necesidades del usuario, se estudia el comportamiento del usuario para poder determinar flujos o interacciones de la aplicación y nos ayuda a realizar algunas pruebas preliminares de la aplicación [10].

UI (User Interface) se centra más en la parte del diseño de la aplicación. Es decir, como se va a visualizar en un teléfono, tablet, reloj, tv, etc. Cómo se ven los botones, textos, cajas de textos para que convivan de una manera orgánica [10].

La imagen 2.14 ayuda a comprender un poco mejor el alcance de cada patrón de diseño para las aplicaciones de dispositivos móviles.

La aplicación se divide en pantallas que interactúan entre ellas para la resolución de un problema.

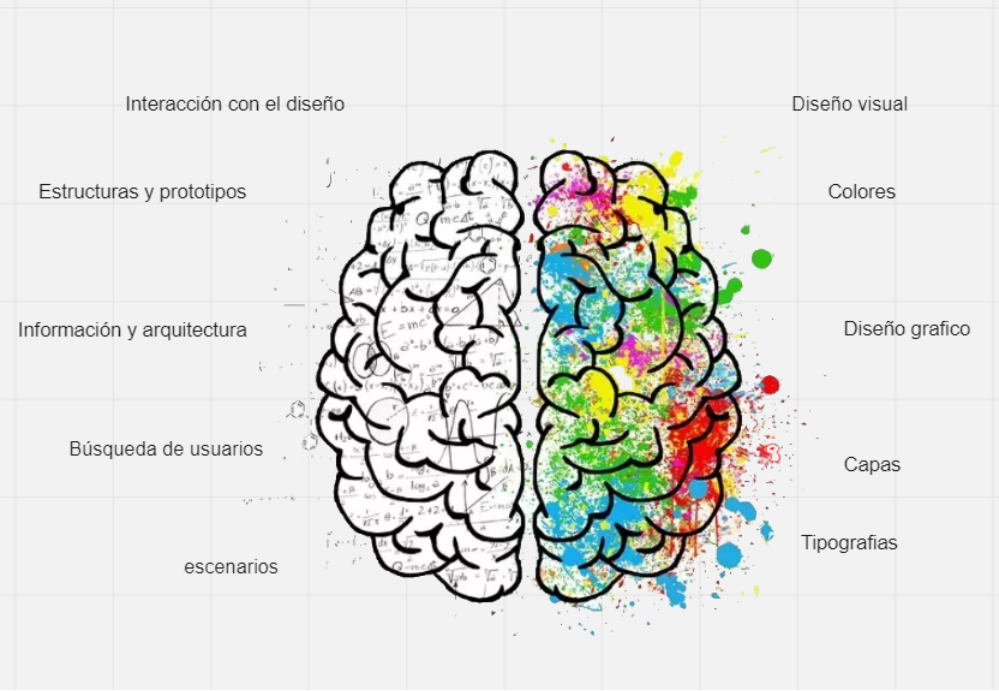

Figura 2.14: A la izquierda se observa los campos orientados de UX y a la derecha el de UI

Tiempo de desarrollo

La planificación del desarrollo de una aplicación es muy importante para evitar retrasos en los tiempos de entrega. Además, permite tener una idea clara del costo de desarrollo.

2.8 Ejemplo de desarrollo de una aplicación

El tipo de aplicación que desarrollaremos será una calculadora muy simple. Para ello, se usará Java y Kotlin, y el sistema operativo será Android.

Primero comenzamos creando la pantalla principal de la aplicación. Lo que se quiere lograr es una interfaz similar a la mostrada en la figura 2.15.

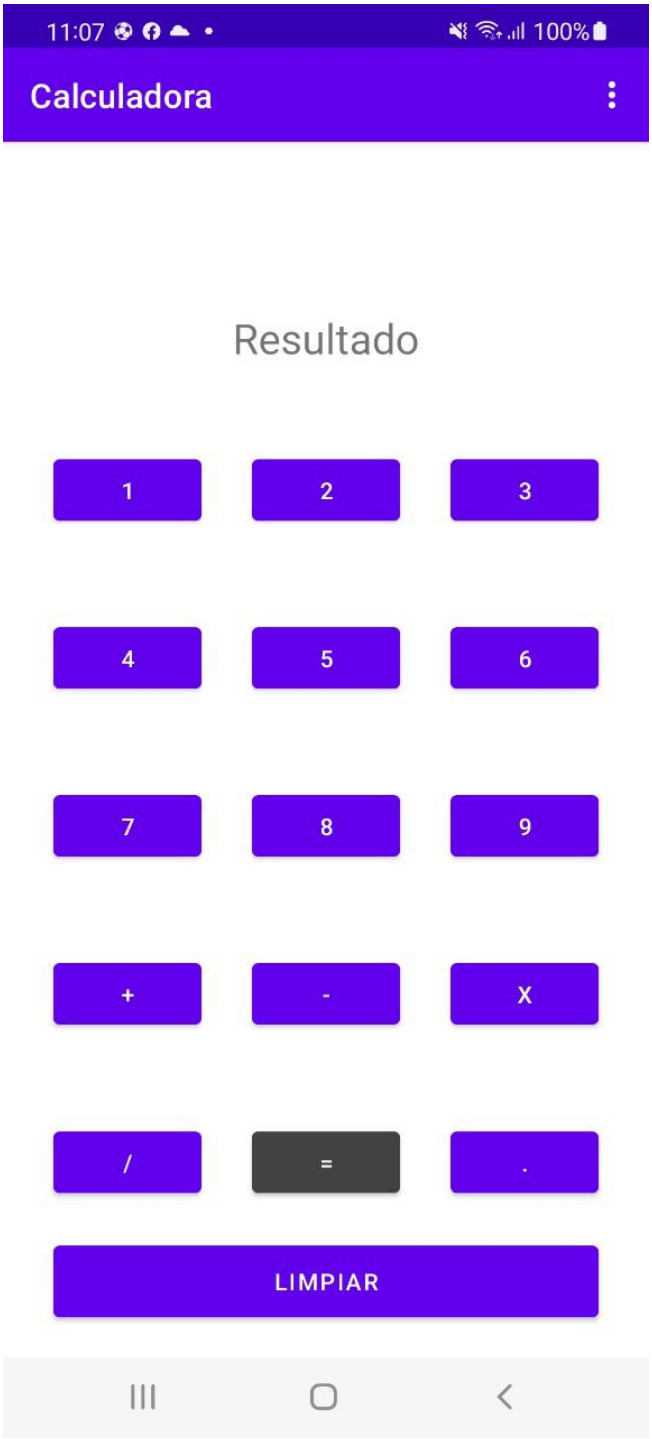

Figura 2.15: Vista principal de la aplicación Calculadora

Creación de una calculadora simple con Java

Para crear esta aplicación en Java, se opta por usar un componente **Fragment**. Esto es una buena práctica de programación. Los paquetes que se deben de importar son los mostrados en el código 2.49.

```
1  package com.example.calculadora;
2
3  import android.os.Bundle;
4  import android.view.LayoutInflater;
5  import android.view.View;
6  import android.view.ViewGroup;
7  import android.widget.Toast;
8  import androidx.annotation.NonNull;
9  import androidx.fragment.app.Fragment;
10 import com.example.calculadora.databinding.FragmentFirstBinding;
```

Código 2.49: Paquetes importados para la aplicación

A continuación, se explica brevemente cada uno de estos paquetes:

- **Boundle**: Contiene todos los recursos y el código de la app.
- **LayoutInflater**: Ayuda a inflar, pintar o visualizar la vista de una actividad, fragmento, view, etc...
- **View**: Objeto en android que contiene los datos del área, visualmente es donde se muestran los elementos de la vista.
- **Toast**: Imprime mensajes temporales por tiempo limitado para avisar o alertar a el usuario por algún posible error .
- **ViewGroup**: Contenedor especial que agrupa layout, views, contenedores, etc...
- **NoNull**: ayuda a tener variables diferente de null, son utilizados en condicionales específicamente.
- **Fragment**: Es una clase tipo activity que contiene una vista (VIEW) y una clase donde se realiza la parte operacional.

El siguiente paso es crear una clase que herede de la clase **Fragment**. En este caso, la clase se nombró **FirstFragment**, ver figura 2.50. En esta clase heredada, se tienen que sobre escribir los métodos siguientes:

- **onCreateView()**: Inicializa todo lo necesario para que la aplicación pueda correr sin problemas y la ultima operación es inflar la pantalla.
- **onDestroyView()**: Elimina todo el contenido de la vista, desde variables y finalizamos con la vista.

Además de sobre-escribir los métodos anteriormente explicados, se declaran los datos miembros siguientes:

- **binding**: De tipo FragmentFirstbinding, que es una clase que vincula información de una vista para ser usada en código. Alguno de los elementos son botones, imágenes, contenedores, etc..
- **cadenaUno**: Variable de tipo String, que concatena el primer valor ingresado.
- **cadenaDos**: Variable de tipo String, que concatena el segundo valor ingresado.
- **resultado**: Variable de tipo String, donde se almacena el resultado de la operación.
- **operador**: Variable de tipo String, define el tipo de operación que se va realizar.
- **operaciones**: Variable de tipo Operaciones, implementa todas las operaciones definidas para una calculadora simple, tales operaciones son suma, resta, multiplicación y división.

```
1 public class FirstFragment extends Fragment {
2     private FragmentFirstBinding binding;
3     private String cadenaUno = "";
4     private String cadenaDos = "";
5     private String resultado = "";
6     private String operador = "";
```

```
7     private Operaciones operaciones = new Operaciones();
```

Código 2.50: Implementación de una clase que hereda de Fragment

Los manejadores de eventos para los botones de la interfaz de usuario principal son creados como clases anónimas dentro de la clase **FirstFragment**, en el código 2.51 se muestra alguno de los botones configurados para escuchar. EL método setOnclickListener le da esa cualidad, cada evento invoca al método **concatenaDigitos()**.

```
1         binding.button1.setOnClickListener(new View.
    OnClickListener() {
2             @Override
3             public void onClick(View view) {
4                 concatenaDigitos("1");
5             }
6         });
7
8         binding.button2.setOnClickListener(new View.
    OnClickListener() {
9             @Override
10            public void onClick(View view) {
11                concatenaDigitos("2");
12            }
13        });
14
15        binding.button3.setOnClickListener(new View.
    OnClickListener() {
16            @Override
17            public void onClick(View view) {
18                concatenaDigitos("3");
19            }
20        });
21
22        binding.button4.setOnClickListener(new View.
    OnClickListener() {
23            @Override
24            public void onClick(View view) {
25                concatenaDigitos("4");
26            }
27        });
```

Código 2.51: Agregar escuchas con el método setOnclickListener() y clases anónimas

Creamos una clase llamada **Operaciones** con el objetivo de no cargar todo a la vista, y como buena practica. Las vistas sólo muestran la información, estas no deben procesar nada, para eso nos apoyamos de las clases, en el código 2.52 se muestra la clase que nos ayuda hacer los cálculos de nuestra aplicación. Esta clase cuenta

con los métodos **sumaDosNumeros()**, **multiplicaDosNumeros()**, **divideDosNumeros()**, **restaDosNumeros()**, todas con el tipo de retorno double.

```
1  package com.example.calculadora;
2
3  public class Operaciones {
4      // Suma dos numeros de tipo double
5      public double sumaDosNumeros(Double variableUno, Double
       variableDos) {
6          return (variableUno + variableDos);
7      }
8
9      // Multiplica dos numeros de tipo double
10     public double multiplicaDosNumeros(Double variableUno,
       Double variableDos) {
11         return (variableUno * variableDos);
12     }
13
14     // Divide dos numeros de tipo double
15     public double divideDosNumeros(Double variableUno, Double
       variableDos) {
16         return (variableUno / variableDos);
17     }
18
19     // Resta dos numeros de tipo double
20     public double restaDosNumeros(Double variableUno, Double
       variableDos) {
21         return (variableUno - variableDos);
22     }
23 }
```

Código 2.52: Código de la clase Operaciones en Java

El código 2.53 en formato XML se usa para "pintar" la pantalla de la aplicación. Al igual que una clase de Java, Kotlin o cualquier otro lenguaje es necesario agregar algunas bibliotecas para poder hacer uso de componentes.

Las bibliotecas que proveen recursos de elemento de diseño se encuentran en la ruta **http://schemas.android.com/apk/res/android**. Estos elementos están representados por sentencias que tienen el prefijo **android:""**, algunos ejemplos son los siguientes:

- **android id** Asigna un valor de tipo cadena de caracteres, esta cadena determina el id que representa a nuestro componente y nos ayuda a unir la parte visual con la parte de código.

- **android layoutWidth** : Define el ancho del componente a usar, El tipo de dato usado es el siguiente **dp** (densidad de píxeles) y **sp**(píxel escalable), este último es usado para definir tamaños en textos.
- **android layoutHeight**: Define el alto de los componentes al igual que **android layoutwidth**
- **android textSize**: Define el tamaño de la fuente de componentes tipo textos.
- **android text**: Asigna una cadena de caracteres.
- **android layoutMarginTop**: Crea un espacio invisible en la parte superior del objeto, el espacio es definido en **dp**

La biblioteca que ayuda a proveer recursos de posicionamiento en la pantalla se encuentra el la ruta **http://schemas.android.com/apk/res-auto**. Estos elementos están representados por sentencias con el prefijo **app:""**, algunos ejemplos son:

- **app layoutConstraintBottomToTopOf**: Fija la parte inferior del componente a otro componente, este recibe una cadena de caracteres con el nombre del Id del componente al cual se fija o "paren" en caso de fijarlo al contraintLayout.
- **app layoutconstraintEndtoEndOf**: Fija un componente hacia la derecha a otro componente, este recibe una cadena de caracteres con el nombre del Id del componente al cual se fija o "paren" en caso de fijarlo al contraintLayout.
- **app ayoutconstraintEndtoStar**: Fija un componente hacia la derecha a otro componente, este recibe una cadena de caracteres con el nombre del Id del componente al cual se fija o "paren" en caso de fijarlo al contraintLayout.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout xmlns:android
     ="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto"
4     xmlns:tools="http://schemas.android.com/tools"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FirstFragment">
```

Código 2.53: Vista XML de aplicación calculadora

Creación de una calculadora simple con Kotlin

Para Kotlin creamos la misma aplicación "calculadora simple". Al igual que en Java, en Kotlin se usa un componente Fragment. En el código 2.54 se listan las bibliotecas que usamos. Como se aprecia, las mismas bibliotecas usadas en Java son importadas en Kotlin, esto es gracias la integración de ambos lenguajes.

```
package com.example.calcula

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.navigation.fragment.findNavController
import com.example.calcula.databinding.FragmentFirstBinding
```

Código 2.54: Fragment de aplicación calculadora en Kotlin

La declaración de las variables son del mismo tipo que en la implementación en Java presentada anteriormente. En el código 2.55 se muestra la implementación de esta parte explicada.

```
class FirstFragment : Fragment() {

    private var _binding: FragmentFirstBinding? = null
    private var cadenaUno = ""
    private var cadenaDos = ""
    private var resultado = ""
    private var operador = ""
    private val operaciones = Operaciones()
```

Código 2.55: Fragment de aplicación calculadora en Kotlin

En el código 2.56 se muestra como se implementa el evento listener a algunos botones, de igual manera, los botones tienen el mismo nombre ya que la interface XML es la misma.

```
        binding.button1.setOnClickListener { concatenaDigitos("1
    ") }

        binding.button2.setOnClickListener { concatenaDigitos("2
    ") }

        binding.button3.setOnClickListener { concatenaDigitos("3
    ") }

        binding.button4.setOnClickListener { concatenaDigitos("4
    ") }
```

Código 2.56: Fragment de aplicación calculadora en Kotlin

Como se puede observar el código Kotlin a comparación del código ejecutado por Java se simplifica mucho, ya que lambda nos ayuda a reducir mucho código y el uso de **When** como sentencia de decisión simplifica el código y es menos engorroso, en Kotlin se definieron más funciones, esto no afecto al tamaño del archivo, ahora veamos que pasa con la clase **Operaciones**, que se muestra en el siguiente código 2.57

```
    // Guardamos los numero para las operaciones
    private fun concatenaDigitos(cadena: String) {
        if (operador.isEmpty()) {
            cadenaUno = cadenaUno + cadena
            mostrarNumero(cadenaUno)
        } else {
            cadenaDos = cadenaDos + cadena
            mostrarNumero(cadenaDos)
        }
    }

    // La etiqueta nos ayuda a visualizar los numeros que van a
    realizar la operacion
    private fun mostrarNumero(cadena: String) {
        binding.textviewResultado.text = cadena
    }

    // Limpia todos los campos para futuras operaciones con
    nueva informacion
    private fun limpiar() {
        binding.textviewResultado.text = ""
        cadenaUno = ""
        cadenaDos = ""
        operador = ""
        resultado = ""
    }

    // Dependiendo del tipo de operador realizamos la operacion
    private fun tipoOperacion() {
```

```
28          when (operador) {
29              OPERADOR_SUMA -> resultado = java.lang.String.
    valueOf(
30                  operaciones.sumaDosNumeros(cadenaUno.toDouble(),
     cadenaDos.toDouble())
31              )
32              OPERADOR_RESTA -> resultado = java.lang.String.
    valueOf(
33                  operaciones.restaDosNumeros(cadenaUno.toDouble()
    , cadenaDos.toDouble())
34              )
35              OPERADOR_MULTIPLICA -> resultado = java.lang.String.
    valueOf(
36                  operaciones.multiplicaDosNumeros(cadenaUno.
    toDouble(), cadenaDos.toDouble())
37              )
38              OPERADOR_DIVIDE -> resultado = java.lang.String.
    valueOf(
39                  operaciones.divideDosNumeros(cadenaUno.toDouble
    (), cadenaDos.toDouble())
40              )
41              else -> Toast.makeText(context, "No se puede
    completar la operaci n", Toast.LENGTH_SHORT).show()
42          }
43          if(resultado.isNotEmpty()){
44              binding.textviewResultado.text = resultado
45          }
46      }
47
48      override fun onDestroyView() {
```

Código 2.57: Fragment de aplicación calculadora en Kotlin

```
1  package com.example.calcula
2
3  class Operaciones {
4      // Suma dos numeros de tipo double
5      fun sumaDosNumeros(variableUno: Double, variableDos: Double)
       = variableUno + variableDos
6
7      // Multiplica dos numeros de tipo double
8      fun multiplicaDosNumeros(variableUno: Double, variableDos:
    Double) = variableUno * variableDos
9
10     // Divide dos numeros de tipo double
11     fun divideDosNumeros(variableUno: Double, variableDos:
    Double) = variableUno / variableDos
12
13     // Resta dos numeros de tipo double
14     fun restaDosNumeros(variableUno: Double, variableDos: Double
    ) = variableUno - variableDos
15 }
```

Código 2.58: Clase operaciones en código Kotlin

Como se puede observar en el código 2.58 el tamaño del código del Fragment disminuyó drásticamente en Kotlin con respecto a Java. Se observa que se redujo entre un 35% a 40%. Para el **XML** es el mismo código 2.53 que se usó para Java. Este XML dibuja la misma pantalla generada mostrada en la figura 2.15.

Como se puede observar, hay una clara ventaja de Kotlin sobre Java en el momento de escribir código. En Kotlin se ve que se reduce el número de lineas. Sin embargo, uno puede preguntarse: ¿ en qué ayuda esto al desarrollador?. Esto se responde fácilmente, ya que las funciones son más sencillas de entender, menos complejas de analizar, el programador puede identificar más fácilmente lo que hace cada método.

Conclusiones

En ocasiones los nuevos lenguajes de programación no sobresalen o pasan desapercibidos, y las empresas u organizaciones que lo impulsaron dejan de darle soporte a una muy temprana edad de su lanzamiento. En otros casos los nuevos lenguajes de programación llegan a desplazar en su totalidad a los ya existentes.

En este trabajo se explicaron los lenguajes de programación Java y Kotlin. Ambos lenguajes son actualmente usados para desarrollar aplicaciones nativas para el sistema operativo Android. Este último es el más popular en los dispositivos móviles, como teléfonos inteligentes y tabletas electrónicas.

Java es un lenguaje con 25 años desde su lanzamiento, mientras que Kotlin tiene apenas 10 años de haber sido liberado. Pese a ser un lenguaje de programación relativamente "joven", Google lo ha nombrado un lenguaje oficial para desarrollar en Android. Kotlin, es un lenguaje que ha cobrado mucha fuerza y popularidad entre los desarrolladores, tanto de aplicaciones móviles como desarrollo backend. Kotlin no desplaza en su totalidad a Java en otros desarrollos, pero en el ámbito de las aplicaciones móviles cada vez es más común escribir y leer código fuente escrito en Kotlin. Uno de los grandes aciertos de este lenguaje es la forma en que simplifica el código, elimina el punto y coma al finalizar sentencias, integra muchas formas de recorrer listas y hace más fácil la toma de decisiones.

A lo largo del documento se presentaron las principales características de los dos lenguajes de programación, junto con un comentario personal basado en la experiencia profesional del autor de este trabajo. Se mostraron los conceptos de la programación

orientada a objetos, así como la forma de implementar estos conceptos en Java y en Koltin. Con el objetivo de mostrar una comparación entre ambos lenguajes, se mostró el desarrollo de una aplicación muy simple pero completa. Se encuentra que con Kotlin los códigos son más cortos y más simples. Se piensa que Kotlin algún día sustituya a Java para desarrollo para Android, esto no por cuestiones técnicas, sino por motivos legales de Google con Oracle.

Referencias

[1] Allen, G. *Android for Absolute Beginners: Getting Started with Mobile Apps Development Using the Android Java SDK*, 1st ed. ed ed. Apress, 2021.

[2] AndroidOS. Android OS, 202.

[3] Arhipov, A. *Kotlin Design Patterns and Best Practices: Build scalable applications using traditional, reactive, and concurrent design patterns in Kotlin*, second ed. Packt Publishing, 2022.

[4] bin Uzayr, S. *Mastering Java A Beginner's Guide*, 1ra ed. 2022.

[5] D. Craig, I. *Object-Oriented Programming Languages: Interpretation*, 1ra ed. 2007.

[6] Faith, K. *A Piece of Java: Introduction to Programming.* Independently Published, 2021.

[7] Forrester, A. *How to Build Android Apps with Kotlin: A hands-on guide to developing, testing, and publishing your first apps with Android*, 1er ed. Packt Publishing, 2022.

[8] Galata, I. *Kotlin Apprentice: Beginning Programming with Kotlin*, first ed. Razeware LLC, 2018.

[9] Google. android11, mar 2022.

[10] Gothelf, J. *Lean UX: Designing Great Products with Agile Teams*, edición 3r ed. O'Reilly Media, 2021.

[11] Griffiths, D. *Head First Android Development: A Brain-Friendly Guide*, 2nd edició ed. 2017.

[12] Horton, J. *Android Programming with Kotlin for Beginners: Build Android apps starting from zero programming experience with the new Kotlin programming language*. Packt Publishing, 2019.

[13] JAMES, K. L. *Android Applications Development in Java*. 2022.

[14] Kousen, K. *Kotlin Cookbook: A Problem-Focused Approach*, edición 1 ed. O'Reilly Media, 2019.

[15] Leiva, A. *Kotlin for Android Developers: Learn Kotlin the easy way while developing an Android App*, 1er edició ed. CreateSpace Independent Publishing Platform, 2016.

[16] McLaughlin, B. *Programming Kotlin Applications: Building Mobile and Server-Side Applications with Kotlin*, 1 er ed. Wiley, 2021.

[17] Sheusi, J. *Android Application Development for Java Programmers*. Delmar Cengage Learning, 2012.

[18] Smyth, N. *Android Studio 4.2 Development Essentials - Kotlin Edition: Developing Android Apps Using Android Studio 4.2, Kotlin and Android Jetpack*. Payload Media, 2021.

[19] Smyth, N. *Android Studio Bumble Bee Essentials - Java Edition: Developing Android Apps Using Android Studio 2021.1 and Java*. Payload Media, Inc., 2022.

[20] Sommerhoff, P. *Kotlin for Android App Development (Developer's Library)*, 1er edició ed. Addison-Wesley Professional, 2018.

[21] Soshin, A. *Kotlin Design Patterns and Best Practices: Build scalable applications using traditional, reactive, and concurrent design patterns in Kotlin*, 2nd edició ed. Packt Publishing; 2nd edición, 2022.

[22] Späth, P. *Beginning Java MVC 1.0: Model View Controller Development to Build Web, Cloud, and Microservices Applications*. 2021.

[23] Steel, J., and To, N. *The Android Developer ' s Cookbook*, primera ed. Addison-Wesley Educational Publishers, 2011.

[24] Stefan, B. *Programming Kotlin*. Packt Publishing, 2017.