1.

Proyecto Práctico TIC Tecnologías de la Información Empresarial

Introducción a la Programación de Aplicaciones en Android

Universidad Paraguayo Alemana (UPA)

2024

Editor y editorial

	Juan Carlos Miranda, Encarnación 6000, Paraguay.
Email:	juancarlosmiranda81@gmail.com

El archivo PDF de este libro se encuentra disponible en formato libre para su descarga en: https://github.com/juancarlosmiranda/android_recipes/upa/latest

Créditos fotográficos

Las fotografías de portada son propiedad de la Universidad Paraguayo Alemana y se publican bajo la licencia Creative Commons Atribución 4.0 Internacional.

2. Sumario

Dedico este material a todos los fueron parte de mi educación y que a la vez tuvieron la suficiente paciencia para enseñarme.

1 Introducción al Módulo

Este documento se presenta como un material de apoyo a los contenidos teóricos y prácticos impartidos en la materia **“Proyecto Práctico TIC”**, correspondiente al plan de estudios de la carrera **Tecnologías de la Información Empresarial – Universidad Paraguayo Alemana (UPA)**.

La información agrupada en **este material no pretende suplantar la documentación oficial** ofrecida por Google, más bien, hace uso de ella para cubrir los tópicos solicitados en el plan de estudios de la universidad.

A lo largo del documento, el estudiante podrá acceder a contenidos teóricos y prácticos sobre la programación de aplicaciones para el sistema operativo Android. En cada capítulo se introducen contenidos teóricos acompañados de ejercicios prácticos y enlaces con la información necesaria. Esto permite al estudiante trabajar en los ejercicios y a la vez, profundizar en la materia, haciendo uso de la información adicional.

1.1 Objetivos generales

El módulo provee a los estudiantes los **conocimientos fundamentales sobre los proyectos aplicados con énfasis en desarrollo web y mobile**. El punto de vista de programación se amplía al introducir el enfoque mobile y las tecnologías, plataformas y paradigmas que soportan toda la arquitectura. **El punto de vista de desarrollo estará orientado a un proyecto práctico**, con la visión mobile del negocio de manera que los estudiantes estén en condiciones de trabajar soluciones a problemas

en la práctica con estas tecnologías, y puedan aplicar los conocimientos adaptados a la realidad del mercado.

El alumno **tendrá la capacidad de desarrollar aplicaciones para la plataforma Android** que incluyan los elementos más importantes. El curso plantea la implementación de **casos prácticos** para la fijación de los conceptos estudiados. El curso se inicia **con una introducción a Java** para que el alumno **tenga los conceptos más importantes de la Programación Orientada a Objetos** antes de iniciar con todo el desarrollo relacionado a Android.

1.2 Objetivos específicos

1.2.1 Competencia académica y metodológica

Los estudiantes pueden pensar **en función de procesos** y aplicar los **conocimientos en ambientes prácticos** y mobile.

- Pueden **definir la arquitectura necesaria** para un proyecto mobile.
- Conocen los métodos de **modelación y desarrollo**, y pueden llevar a cabo proyectos de procesos de manera autónoma.
- Los estudiantes conocen **diferentes niveles de los procesos mobile** – desde el diseño hasta el nivel de funcionamiento.
- Conocen a profundidad la **metodología de desarrollo orientada a objetos**.

1.2.2 Competencia social y personal

- Los estudiantes están capacitados para manejar e**jercicios con procedimientos analíticos**.
- Pueden **obtener de forma independiente informaciones** orientadas a planteamientos y aplicarlas para la resolución de problemas.

2 Introducción a la Programación Orientada a Objetos con Java

2.1 Instalación del entorno de trabajo

Existen varias alternativas para compilar programas Java entre las cuales se pueden mencionar: Apache Netbeans [1], Eclipse IDE [2], IntelliJ IDEA[3] . Esta unidad se centra en los conceptos de Java, no obstante, se da la libertad al estudiante de que elija el entorno de trabajo que más desee.

- Apache Netbeans [https://netbeans.apache.org/].
- IntelliJ IDEA [https://www.jetbrains.com/idea/].
- Eclipse IDE [https://www.eclipse.org/downloads/packages/installer].

2.2 Conceptos básicos de programación

Java es un lenguaje amplio en cuanto a sintaxis y funcionalidades, tal como se puede ver en la documentación ofrecida por el sitio oficial “ORACLE Java Documentation” [4]. A lo largo de este apartado, **se hará un repaso sobre los tópicos esenciales para escribir programas**, utilizando un subconjunto de elementos del lenguaje Java, cuyos resultados podrán ser visualizados en pantalla por medio de una interfaz de texto. **Se espera que esta unidad sirva como un paso previo** a la programación de aplicaciones móviles.

A continuación se listan los tópicos que se abarcan:

- Estructura de un programa Java.
- Tipos de datos.
- Entrada y salida estándar.
- Manejo de salidas.

- Estructuras de control de flujo: condicionales.
- Estructuras de control de flujo: ciclos repetitivos.
- Manejo de arreglos y matrices. (lo trabajarán los estudiantes mediante ejercicios)
- Manejo de archivos. (lo trabajarán los estudiantes mediante ejercicios).

Con el fin de acompañar las explicaciones teóricas, acompaña a este material un repositorio con ejemplos de códigos, el mismo se encuentra disponible en [https://github.com/juancarlosmiranda/java_recipes/].

2.2.1 Estructura de un programa Java

La estructura básica de un programa Java comprende la declaración del **paquete, clase y el método principal** que contiene las instrucciones del programa (Figura 1). A su vez, se declaran los **argumentos externo**s que recibirá el programa. [https://introcs.cs.princeton.edu/java/11cheatsheet/]

```
package com.mycompany.intro_java;

public class MyFirstClassInJava {
    public static void main(String[] args) {
        System.out.println("My first program in Java!");
    }
}
```

Figura 1: Programa básico en Java “MyFirstClassInJava.java”.

2.2.2 Tipos de datos

Los tipos de datos estándar del lenguaje Java, representan los tipos básicos p**ara crear elementos más complejos a partir de su combinación**. En la Tabla 1, se pueden visualizar los tipos de datos, con referencia al rango de valores, los operadores que soportan, seguidos de ejemplos de expresiones comúnmente utilizadas.

Tabla 1: Tipos de datos incorporados en Java. Adaptado de [5]

Tipo de datos	Conjunto de valores	Operadores	Expresión	Valor
int	-2^{31} a $2^{31} - 1$ (32 bits)	+ (suma)	5 + 3	8
		- (resta)	5 - 3	2
		* (multiplicación)	5 * 3	15
		/ (división)	5 / 3	1
		% (resto)	5 % 3	2
double	(64 bits)	+ (suma)	3.141 - .03	3.111
		- (resta)	2.0 – 2.0e-7	1.9999998
		* (multiplicación)	100 * .015	1.5
		/ (división)	6.02e23 / 2.0	3.01e23
boolean	true o false	! (not)	!false	true
		&& (and)	true && false	false
		|| (or)	false || true	true
		^(xor)	true ^true	false

Tipo de datos	Conjunto de valores	Operadores	Expresión	Valor
char	Caracteres (16 bits)			
String	Secuencia de caracteres	+		

Además, en la Tabla 2, se visualizan las tablas de verdad que se pueden obtener en base a la aplicación de los operadores NOT, AND, OR, XOR.

Tabla 2 Operadores lógicos NOT, AND, OR, XOR.

A		NOT A
False		True
True		False
A	**B**	**A AND B**
False	False	False
False	True	False
True	False	False
True	True	True
A	**B**	**A OR B**
False	False	False
False	True	True
True	False	True
True	True	True
A	**B**	**A XOR B**
False	False	False
False	True	True
True	False	True
True	True	False

Otros tipos de datos primitivos son: **long, short, byte, float.**

Material complementario con ejemplos sobre tipos de datos puede ser consultado en [6] y [7]. Ejemplos de declaración de variables pueden ser visualizados en **"UserInputClass.java"**.

2.2.3 Entrada y salida estándar

Java permite el manejo de la entrada y salida estándar, para introducir datos y presentar resultados. Entiéndase entrada estándar el **uso del teclado para introducir datos** y salida estándar el uso de una **interfaz de texto** en pantalla. En la Figura 2, se presenta el

fragmento de código necesario para introducir números y cadenas de caracteres y mostrarlos en pantalla.

```
Scanner sc01 = new Scanner(System.in);
System.out.print("Enter an integer number (eg. 1234):");
intNumber = sc01.nextInt();

Scanner sc02 = new Scanner(System.in);
System.out.print("Enter a string data: ");
stringData = sc02.nextLine();
```

Figura 2: Entrada de datos por teclado, salida por consola de texto. “UserInputClass.java”

La salida estándar, a su vez, **puede ser modificada para presentar distintos tipos** de datos según el formato que se requiera. Un ejemplo, puede ser revisado en el archivo **“UserOutputClass.java”**.

2.2.4 Estructuras de control de flujo: condicionales

Ejemplos de las estructuras condicionales **if - else** (Figura 3), **if – else if** (Figura 4) y **switch** (Figura 5) pueden verse en el archivo **“ControlFlowClass01.java”**.

```
if (testValue < limit) {
    System.out.println("testValue < limit");
} else {
    System.out.println("Maybe ... testValue >= limit");
    if (testValue == limit) {
        System.out.println("testValue == limit");
    } else {
        System.out.println("testValue > limit");
    }
}
```

Figura 3: Condicional if-else. “ControlFlowClass01.java”.

Generalmente, la estructura **if – else if** suele ser utilizada para el manejo de opciones.

```
int testscore = 76;
char grade;

if (testscore >= 90) {
    grade = 'A';
} else if (testscore >= 80) {
    grade = 'B';
} else if (testscore >= 70) {
    grade = 'C';
} else if (testscore >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
System.out.println("Grade = " + grade);
```

Figura 4: Condicional if-else if

La estructura **switch**, se aplica al control de opciones numéricas o de caracteres.

```
String message = null;
int optionSwitch = 5;

switch (optionSwitch){
    case 0: message = "Option 0";
         break;
    case 1: message = "Option 1";
         break;
    case 2: message = "Option 2";
         break;
    case 3: message = "Option 3";
         break;
    case 4: message = "Option 4";
         break;
    default: message = "Option DEFAULT";
          break;
}
```

Figura 5: Condicional switch

2.2.5 Estructuras de control de flujo: ciclos repetitivos

Java provee tres tipos de estructuras repetitivas que pueden ser aplicadas según la necesidad. La primera es el ciclo **while** (Figura 6), el cual ejecuta una condición de control como primera opción. La segunda, el el ciclo **do – while** (Figura 7), que realiza un control de flujo al final de las instrucciones. Por último, la estructura **for** (Figura 8), con la cual se pueden procesar rangos de valores definidos. Ejemplos de las estructuras mencionadas pueden ser consultadas en el archivo "**ControlFlowClass02.java"**.

```
int countWhile = 1;
int limitWhile = 10;

while (countWhile < limitWhile){
    System.out.println(countWhile);
    countWhile++;
}
```

Figura 6: Ciclo repetitivo while. "ControlFlowClass02.java".

```
int countDoWhile = 1;
int limitDoWhile = 10;

do{
    System.out.println(countDoWhile);
    countDoWhile++;
} while(countDoWhile < limitDoWhile);
```

Figura 7: Ciclo repetitivo do..while. "ControlFlowClass02.java".

```
int countFor = 1;
int limitFor = 10;

for (countFor = 1; countFor < limitFor; countFor++){
    System.out.println(countFor);
}
```

Figura 8: Ciclo for. "ControlFlowClass02.java".

Existen dos sentencias que permiten modificar el flujo de los programas, estas son: **break** y **continue**. La primera **break**, se utiliza en los ciclos **while**, **do - while** y **for** para interrumpir la iteración. En el caso de la segunda, **continue**, cuando se aplica en un ciclo, permite omitir la iteración actual y continuar a la siguiente sentencia.

2.2.6 Ejercicios propuestos

Durante el desarrollo de los ejercicios, se podrá hacer uso de las hojas de referencias para Java [8] y las guías de estilo [9]. A continuación se citan los objetivos esperados:

- Instalar las herramientas necesarias para compilar programas en Java.
- Escribir programas en Java haciendo uso de las estructuras condicionales y ciclos.
- Aplicar las guías de estilo para la codificación en Java.
- Utilizar arreglos, matrices y colecciones de datos.

Estructuras condicionales y ciclos

1. Instalar el ambiente de programación para Java. El alumno deberá descargar el entorno de programación y los paquetes correspondientes al Java Development Kit (JDK). Puede utilizarse como referencia [10] según el IDE que se desee instalar.
2. Descargar el ejemplo **“ControlFlowClass02.java”**. A partir de este archivo modificar los valores de entrada para que pueda contabilizar **X** números enteros. Donde la

variable **X** es un número entero introducido por el usuario entre 1 y 100. Al final, el programa deberá emitir un reporte en pantalla tal como sigue:

- Cantidad de números > 10.
- Cantidad de números < 10.
- Cantidad de números > 5 y < 10.

3. Un número par es aquel divisible por 2 y un número impar es aquel que no es divisible por 2. Elaborar una programa que dado un número ingresado por el usuario determine si es par o impar.
4. Elaborar un programa que permita al usuario ingresar **10 números enteros** en el rango de 0 – 254, y por cada número determine si es par o impar. Al finalizar deberá emitir un conteo de números pares e impares.
5. Crear un programa que permita guardar datos en memoria (con arreglos de números) de temperaturas semanales registradas en grados Celsius. El usuario deberá ingresar el día de la semana (martes por ejemplo), seguido de la temperatura registrada (35.5). **La capacidad de almacenamiento es para 7 días**. Calcular:
    - Promedio de temperaturas.
    - Temperatura mínima y el día. Ejemplo miercoles = 30.0.
    - Temperatura máxima y el día. Ejemplo martes = 35.0.

Procedimientos, estructuras

6. Crear una programa que muestre un menú en pantalla de texto. El menú de usuario debe contener: OPCION_01, OPCION_02, OPCION_03, SALIR. Cada selección de opción debe llevar a un procedimiento y mostrar un mensaje al usuario sobre la opción elegida. La opción SALIR debe cortar la ejecución.

Matrices y arreglos

7. Haciendo uso de estructuras de datos arreglos (arrays), cargar por teclado 10 números y obtener los siguientes datos: mínimo, máximo, promedio. Se puede utilizar como base el ejemplo **"ArrayMatrixCollections01.java"**.
8. **Modificar el ejercicio 6)** para que la cantidad total que almacenará el array sea un número entero que el usuario introduzca por teclado. El dato ingresado por el usuario estará limitado al rango de datos **0 <= TAMANO <= 20**.
9. Copiar una matriz cargada por usuario a una nueva.
10. Cargar dos matrices por usuario. Realizar una comparación lógica posición por posición. Como resultado, se debe mostrar una matriz con los datos que son distintos.

Colecciones

11. A partir del ejemplo **"ArrayMatrixCollections02.java"**, crear un programa que permita al usuario cargar números enteros en una matriz **4 x 4** y determinar la suma de la diagonal principal (esquina superior izquierda a esquina inferior derecha).

12. **Indagar sobre los tipos de colecciones en Java (paquete java.util)**. Clases: Vector, List, Set, Map. Se sugiere consultar [11] y [12].
13. Crear un programa ejemplo utilizando la clase Vector, cargar y mostrar los valores en pantalla. Aplicar los métodos: size, add, elementAt, insertElementAt y remove. Se puede consultar el ejemplo **"ArrayMatrixCollections03.java".**
14. Definir un **ArrayList**, cargar, determinar el tamaño y mostrar los elementos en pantalla.
15. Definir un **HashTable**, cargar, determinar el tamaño y mostrar los elementos en pantalla.

Cadenas (String)

16. Manejo de cadenas. Ingresar una cadena y verificar si palindromo.
17. Recibir mediante teclado una cadena de 20 caracteres. Crear una clase que facilite la búsqueda de una subcadena dentro de la cadena dada.

Archivos

18. Crear un programa que permita guardar 10 cadenas de texto ingresadas por teclado en un archivo de texto.
19. Agregar al ejercicio anterior una rutina que indicando el nombre del archivo muestre su contenido en pantalla.
20. Leer datos desde un archivo de texto separado por comas. Cargar en una lista y ordenarlo alfabéticamente utilizando las funciones de colecciones.

Deinición y Creación de Paquetes

21. Crear un paquete que contenga métodos con cuatro operaciones aritméticas con dos números como parámetros. Los métodos deben retornar resultados del tipo float. Luego, se debe incluir en el programa principal el llamado al paquete y sus funciones. La clase no debe ser instanciada.
22. Crear un paquete que contenga métodos con cuatro operaciones aritméticas con dos números como parámetros. Los métodos deben retornar resultados del tipo float. Luego, se debe incluir en el programa principal el llamado al paquete y sus funciones. La clase DEBE SER INSTANCIADA (se debe crear un objeto).

2.3 Programación Orientada a Objetos (OOP)

El paradigma de Programación Orientada a Objetos (OOP), proporciona abstracción para diseñar software. **Un objeto es una construcción en la cual se agrupan datos (propiedades) y procedimientos para operar y acceder a esos datos (métodos).** Este diseño **favorece la reutilización de componentes software** permitiendo escalar a diseños más complejos.

Al agrupar código fuente en objetos de software individuales, se puede obtener una serie de beneficios, entre ellos [13]:

- **Modularidad:** el código fuente de un objeto se puede escribir y mantener independientemente del código fuente de otros objetos.
- **Ocultación de información (encapsulamiento):** al interactuar únicamente con los métodos de un objeto, los detalles de su implementación interna permanecen ocultos al mundo exterior.
- **Reutilización de código:** si ya existe un objeto (tal vez escrito por otro desarrollador de software), se puede utilizar dicho objeto en un programa.
- **Facilidad de depuración:** si un objeto en particular resulta ser problemático, se puede eliminar de la aplicación y conectar un objeto diferente como reemplazo. Cada objeto puede ser depurado sin afectar a otros componentes.

Un elemento en la vida real puede ser modelado como un objeto. **Cada objeto pertenece a un determinado tipo, y este tipo se denomina clase.** Las clases, representan las bases o plantillas para que los desarrolladores de software puedan **crear objetos con comportamientos y propiedades especializadas**. Además, cuando se crea un objeto, se copia la definición de una clase a memoria; a este hecho se le conoce como **instancia de una clase**.

Por último, las clases pueden ser organizadas según la funcionalidad implementada en el código, creando así construcciones más complejas. **Un paquete es un espacio de**

nombres que permite organizar clases e interfaces de manera conceptual, siendo análogo a los directorios en las computadoras.

En OOP, se aplica el concepto de **herencia**, bajo este concepto, las clases (tipos) pueden heredar y transferir propiedades y métodos, permitiendo la reutilización de código. En base a lo anterior, las clases pueden ser denominadas como: **subclase, superclase, clase base**.

Se denomina **subclase (subtype / subclass),** a la clase que hereda propiedades de otra clase. Se entiende por **superclase (supertype / superclass)**, la clase que transfiere las propiedades a una subclase. Por último, una clase que posee varias subclases es denominada **clase base (base type)**. A continuación, un ejemplo puede ser visualizado en la Figura 9.

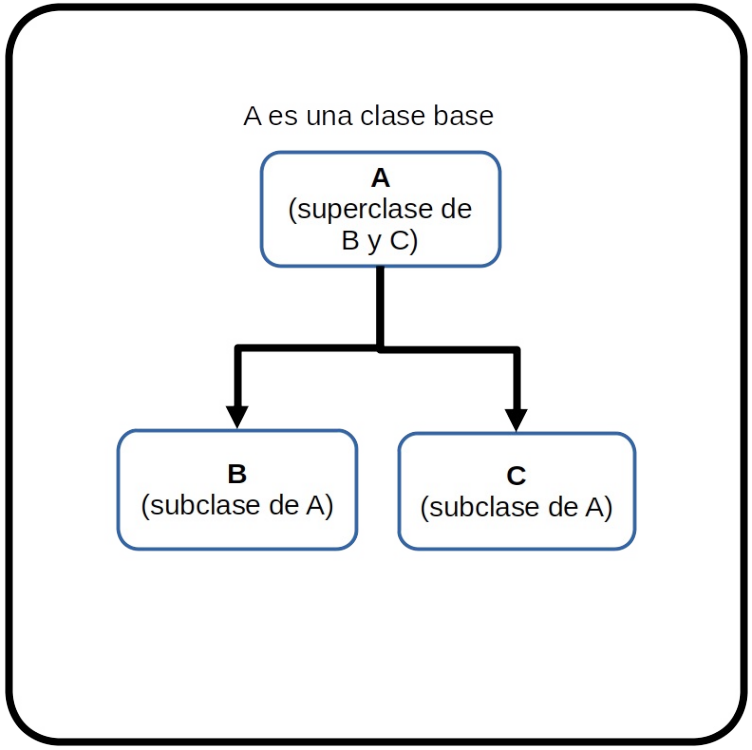

Figura 9: Superclase, subclase, clase base

Veamos un ejemplo de la vida real, un perro puede tener estados y comportamientos. Los estados podrían ser: color, raza, hambre, entre otros (propiedades). El comportamiento (métodos) podría ser modelado como: ladrar, comer, buscar, mover cola.

2.3.1 Creación de objetos

Un objeto **es una instancia de una clase**, en Java, se puede crear con la palabra reservada **“new”** y el operador **“=”**. Cabe aclarar, que todas las variables internas a la clase son inicializadas por Java con **0 (cero)** para tipos numéricos y **null** para **String**. Ej. **Trivial trivial = new Trivial();**.

Las clases se definen mediante la palabra reservada **class** tal como se pude ver en la (Figura 10). Los valores internos de las clases son inicializados por defecto. Pero se puede utilizar un método especial denominado **“constructor”** para definir los valores internos.

```
class Trivial {
    private long ctr;

    public void incr(){
        ctr++;
    }

    public long getIncr(){
        return ctr;
    }
}

public class OopClassCreation01 {
        public static void main(String[] args) {
            Trivial trivial = new Trivial();
            System.out.println(trivial.getIncr());
        }
}
```

Figura 10: Definición de clase Trivial y creación de un objeto. “OopClassCreation01.java”

“OopObjCreation02.java” muestra un caso más completo de definición de una clase con constructor, creación de objeto, seguido de acceso a una propiedad y modificación de la misma.

2.3.2 Objetos, Herencia y Polimorfismo

En este apartado se presentan dos conceptos claves en OOP, uno de ellos es la **herencia** y el otro es el **polimorfismo**.

Se entiende por **“herencia”** cuando es posible extender clases (subclases) a partir de una clase existente (superclase). Por otro lado, se entiende por **“polimorfismo”**, cuando los subtipos (subclases) de una clase dada pueden ser asignados a una variable del tipo de clase base (superclase). Es decir, si existe una relación de herencia entre clases, un objeto puede cambiar de forma.

```
class Car {
    private long odometer;

    // contructor
    public Car(){
        this.odometer = 0;
    }

    public void drive(){
        System.out.println("Car - Driving...");
        this.odometer++;
    }

    @Override
    public String toString(){
        return Long.toString(odometer);
    }
}

class Ragtop extends Car{
    @Override
    public void drive(){
        super.drive();
        System.out.println("Ragtop - Driving...");
    }
}
```

Figura 11: Clase Ragtop hereda de la clase Car “OopObjInheritance01.java”. Fuente [14]

La clase Ragtop hereda características de la clase Car, mediante **extend**, además reescribe el método **drive()** con **@Override**.

Más ejemplos sobre herencia y polimorfismo pueden ser enconrados en los archivos **“OopObjInheritance01.java”** y **“OopObjInheritance02.java”**.

2.3.3 Modificadores de acceso en Java

Habitualmente, un bloque de sentencias se define con llaves **{}**, lo que limita el alcance de las definiciones de variables, métodos y tipos que se hagan dentro de ese bloque. Por otro lado, en este material, se han utilizado los términos **“public”** y **“private”** en definiciones sin aportar mayores explicaciones. Es por eso, que en este apartado se introduce el concepto de **modificador**.

Los modificadores pueden cambiar el comportamiento de un objeto definen el alcance y la visibilidad de las clases, métodos y propiedades (variables). **Su funcionalidad cambia dependiendo de la combinación en las definiciones**. Por defecto, en Java, **si no se especifica un modificador de acceso**, las clases, métodos y propiedades son accesibles dentro del paquete que se han definido. A continuación en la Tabla 3, se ofrece un panorama detallado sobre estas combinaciones y el efecto que se obtiene al aplicar los modificadores.

Tabla 3: Modificadores de accesos en Java

Modificador	Aplicado a	Efecto
public	class	La clase es visible fuera del paquete al que pertenece. No presenta restricciones, es accesible desde cualquier parte del programa.

Modificador	Aplicado a	Efecto
	method	El método es accesible anteponiendo el **nombre de la instancia** del objeto.
	variable	El acceso a la variable es permitido anteponiendo el nombre de la instancia del objeto. **Se recomienda evitar las definiciones de propiedades con modificador "public".**
protected	Class, method, variable	Los métodos declarados como protegidos son **accesibles dentro del mismo paquete** o subclases en distintos paquetes.
private	class	La clase NO es visible fuera del paquete al que pertenece.
	method	El acceso al método se encuentra limitado a la clase a la que pertenece.
	variable	El acceso a la propiedad se encuentra limitado a la clase a la que pertenece.
final	class	Cualquier intento de definición de subclases dará un error. La clase es inmutable.
	method	El método no puede ser anulado en una subclase (@Override).
	variable	Cuando se asigna un valor, el valor permanece constante. Una variable final es asignada exactamente una vez.
static	class	Las clases son visibles utilizando el nombre calificado sin especificar el nombre de la instancia.
	method	El acceso al método se realiza

Modificador	Aplicado a	Efecto
		anteponiendo el nombre de la clase.
	variable	Indica que la propiedad definida pertenece a la definición de clase, no a la instancia de la clase.

Por último, para finalizar, para referirse a variables internas dentro de una clase, se antepone el operador **“this”**. Este indicaque que el nombre de la variable a la cual se está llamando corresponde a la clase donde se encuentra la instrucción.

2.3.4 Clases abstractas

Las **clases abstractas** permiten declarar conjuntos de métodos omitiendo la implementación de código de los mismos. Este tipo de clases son utilizadas como **superclases**, para luego implementar código específico en las **subclases**.

La utilidad de las **clases abstractas**, se hace visible cuando se desea implementar un patrón de métodos a modo de plantilla. Este patrón, serviría como una pieza reutilizable de código, a continuación se puede visualizar un ejemplo en la Figura 12.

```
abstract class TemplateService {
   abstract void runService(); //abstract definition
   abstract void stopService();
   abstract void statusService();
   abstract void versionService();
}
class ServiceLinux extends TemplateService {
    void runService() {
       String s="s";
       System.out.println("ServiceLinux - runService()");
    }
    void stopService() {
       System.out.println("ServiceLinux - stopService()");
```

```
    }
}
```

Figura 12: Clase abstracta y la implementación “OopObjAbstract01.java”.

2.3.5 Interfaces

En Java existe la posibilidad de extender clases utilizando **interfaces**. **Las interfaces permiten la declaración de tipos de datos sin tener que definir la implementación del código**. Por ejemplo, se pueden definir métodos de manera genérica en un archivo e implementar el código en otro. Es obligatorio que cada método definido en la interface, tenga su implementación correspondiente. A continuación, se puede visualizar un ejemplo en la Figura 13. También, en el código anexo **“OopObjInterface01.java”**, se puede visualizar un ejemplo que aplica una y varias interfaces en la creación de objetos.

```
interface Animal {
  public void animalSound();
  public void sleep();
}

class Pig implements Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

Figura 13: Ejemplo de interface “OopObjInterface01.java”.

2.3.6 Gestión de errores, manejo de excepciones

En el lenguaje Java, los errores son manejados a través de excepciones [15]. Una excepción **“Exception”**, hace referencia a una condición fuera de lo normal que se produce en tiempo de ejecución de la aplicación. Habitualmente, las excepciones se generan cuando se produce un falla en la aplicación como consecuencia de un error de utilización por parte del usuario.

La herramienta para el manejo de errores se encuentra contemplada en la estructura **try – catch – finally**. Dentro del bloque **try** se definen las instrucciones que serán vigiladas. En el bloque **catch**, se colocan las instrucciones que se encargan del manejo del error, es decir, especifican como continuará el programa. Por último, el bloque **finally**, se encarga de ejecutar código independientemente si ocurrió o no una excepción.

Para mayor información, se pueden consultar los ejemplos (Figura 14) **“ErrorManagement01.java”** y **“ErrorManagement02.java”**.

```
try{
    quotient01 = dividend01 / divisor01;
    reminder01 = dividend01 % divisor01;
} catch(Exception e){
System.out.println("I catched and error ->" +
e.getMessage());
System.out.println("I catched and error ->" + e);

} finally {
    System.out.println("I always walk around here");
}
```

Figura 14: Ejemplo de uso try – catch - finally “ErrorManagement01.java”.

2.3.7 Manejo de archivos

Java ofrece distintas maneras de trabajar con archivos de textos, en este material solo se citan algunos ejemplos.

A la hora de trabajar con archivos, se puede mencionar distintas operaciones comunes:

- Abrir un archivo para lectura.
- Crear un archivo para escritura.
- Agregar datos a un archivo sin borrar el contenido previo.
- Cerrar archivos.

Las clases normalmente utilizadas son **“FileReader”** y **“FileWriter”** para lectura y escritura respectivamente. Además, las clses anteriores son utilizadas para combinar el manejo de los flujos de caracteres con **“BufferedReader”** y “**BufferedWriter”**. Ejemplos que acompañan este material pueden ser analizados en **“FileManagement01.java”** y **“FileManagement02.java”**.

2.3.8 Ejercicios propuestos

El objetivo de este apartado es aplicar los conocimientos sobre OOP mediante ejercicios.

- Aplicar las estructuras necesarias para escribir programas siguiendo el paradigma de orientación a objetos.
- Aplicar herencia, polimorfismo, clases abstractas.

1. Implementar una clase abstracta **“Mamiferos”**, la cual deberá contener los siguiente métodos: **run(), sleep(), walk(), animalSound()**. A partir de **“Mamiferos”**, crear las clases Jirafa y Elefante. Se debe implementar los métodos heredados de la clase abstracta. Por último, crear objetos y llamar a cada uno de los métodos en el programa principal.

2. En base a las clases creadas en el ejercicio 1), se debe agregar las clases **“Gato” y “Perro”**, siguiendo el mismo patrón de herencia e implementando los métodos en cada clase. Cargar en un array 4 objetos (Jirafa, Elefante, Gato, Perro) e iterar llamando al método **walk()**. Se puede utilizar como referencia **“OopObjInheritance02,java”**.

3. Crear una clase base **“muebles de oficina”**. A partir de la clase base crear las clases: **mesa**, **silla**. Las **“mesas inteligentes”** derivan de **“mesa”** tienen la posibilidad de almacenar su posición geográfica y moverse mediante el llamado de 4 métodos: **arriba, abajo, izquierda, derecha**. Todas las clases tienen métodos para saber si tienen patas (SI/NO) y un método que devuelve el número de patas.

4. Crear un programa que permita ingresar números del tipo **float y que realice una división**. Mostrar el resultado al usuario. El programa debe capturar la excepción de división por cero y mostrar el siguiente mensaje al usuario “No se puede dividir por cero”.

5. Crear una clase que simule el comportamiento de un recipiente del tipo aerosol. El recipiente será utilizado mediante los métodos: **click(), cargar(), vaciar(), verCarga()**. Deberá llevar registro de un **estado interno**, manteniendo registro de la cantidad de líquido y su unidad respectiva. Los valores internos serán inicializados en el constructor.

2.4 Conceptos avanzados de la programación orientada a objetos (FALTA)

ssss

TEXTO CAMBIAR (Error: no se encontró el origen de la referencia)

2.5 Material complementario

Java es un lenguaje muy versátil, lo que lo hace una herramienta **ideal para varias tareas**. Por eso, algo que suele ser valorado como un extra en la industria es rendir una certificación [16], de manera a demostrar los conocimientos sobre el lenguaje.

En este apartado se agrupan enlaces que el estudiante podrá utilizar como referencia para los ejercicios. Estos materiales abarcan temas tales como: sintaxis del lenguaje Java, tutoriales sobre programación OOP, estilos de codificación. Se espera que el material complemente lo comentado durante clases.

Referencias para el lenguaje Java

- Learn Java. [https://dev.java/learn/]
- The Java Tutorials. [https://docs.oracle.com/javase/tutorial/java/javaOO/index.html]
- Google Java Style Guide. [https://google.github.io/styleguide/javaguide.html].
- Java Programming Cheatsheet. [https://introcs.cs.princeton.edu/java/11cheatsheet/]
- Java Language Basics. [https://dev.java/learn/language-basics/]
- Code Java. Página con muchos ejemplos organizados por área. [https://www.codejava.net/java-se]

Programación Orientada a Objetos

- Conceptos de OOP. [https://dev.java/learn/oop/]
- Interfaces. [https://dev.java/learn/interfaces/]
- OpenDSA, en el capítulo 4 “Design I” ofrece diversos ejemplos para los conceptos de OOP. [https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/]
- Ejercicios de Programación Orientada a Objetos con Java y UML. Es un libro que trata bastante bien los fundamentos. [https://repositorio.unal.edu.co/handle/unal/80546].

3 Desarrollo de aplicaciones en Android – Nivel Básico

Introducción de Android

Historia

Los sistemas operativos para dispositivos móviles han evolucionado a lo largo de la historia acompañando el avance de la tecnología. Distintos proyectos han aparecido en el mercado, estando algunos de ellos vigentes y otros extintos [17], [18]. Es así, que Android (al momento de redactar este material) no es el único sistema operativo, pero sí el más popular.

Android es un sistema operativo para dispositivos móviles **basado en el kernel de Linux**. Inicialmente surgió como un proyecto desarrollado por Android Inc., la cual fue adquirida más tarde por **Google en el año 2005**, siendo lanzada al mercado la **primera versión del sistema operativo en el 2008** [19]. Según StatCounter GlobalStats, un sitio dedicado al conteo estadístico en Internet, a Junio del 2024, Android se encuentra instalado en una gran cantidad de dispositivos y posee la mayor tasa de mercado a nivel mundial [20].

Desarrollo de aplicaciones

Gran parte de la popularidad de Android, radica en la aceptación por parte de distintos fabricantes, quienes han migrado y adaptado el sistema operativo a diferentes plataformas hardware (teléfonos móviles, tabletas, relojes, televisores, software para automóviles y equipos estéreo entre otros). Además, Android incluye herramientas de desarrollo (*software development kit (SDK)*), que ayudan a

escribir código y ensamblar módulos de software para crear aplicaciones de usuario originales. También, el desarrollo de software, se ve complementado por un sitio de ventas de aplicaciones que brinda la distribución de las aplicaciones desarrolladas.

Desde sus inicios, el lenguaje de programación para Android por excelencia ha sido **Java**, seguido por **C/C++** para aplicaciones que requieren acceso a bajo nivel. A partir del 2019, la empresa **Google junto a Jetbrains**, promueven el uso de **Kotlin** como lenguaje principal. Kotlin es un lenguaje con estructura similar a Java, más expresivo y conciso, que introduce simplificación a la hora de programar aplicaciones. Al momento de redactar este material, Java se encuentra soportado, pero **una gran cantidad de documentación nueva es producida y actualizada enfatizando el uso de Kotlin** para el desarrollo de aplicaciones modernas en Android [21].

El uso de Kotlin como lenguaje para programación **se extiende fuera del alcance de los contenidos de esta materia**. Sin embargo, el estudiante podría profundizar conocimientos a partir de los siguientes materiales sugeridos: [22], [23].

Razones para desarrollar en aplicaciones en Android

Más allá del lenguaje que se seleccione para desarrollar aplicaciones, se puede mencionar, que existen buenas razones que justifican el aprendizaje para crear aplicaciones en Android:

- **Es la plataforma más popular** para aplicaciones móviles, su uso es muy extendido. Su presencia en diversos tipos de dispositivos, hace que sea ideal para llegar a un gran público.
- Existe **un buen soporte**, actualización de versiones y herramientas de desarrollo para la creación de aplicaciones. Constantemente se actualizan los tutoriales y documentaciones.
- Brinda una **buena experiencia de uso** para los usuarios finales.
- Existe un **mercado creciente de aplicaciones** que permiten llegar al usuario final.

Selección de herramientas

Dependiendo del público al cuál se intenta dirigir una aplicación, será necesario tomar decisiones que podrían restringir **la elección de sistemas operativos y con ello las herramientas** que se utilizarán en el proyecto.

Desarrollar aplicaciones **nativas o multiplataformas** es una decisión que influirá en el desarrollo de la solución. Es necesario pensar en el usuario final y responder a los siguientes interrogantes:

- ¿Cuál es el público al que se dirige la aplicación?
- ¿Qué problema intenta solucionar?
- ¿Qué tipos de dispositivos utilizan?
- ¿Uso de datos provenientes de sensores?
- ¿Se requiere almacenamiento interno, externo o ambos?

La preguntas anteriores, podrían guiar un primer avance en cuanto a la selección de herramientas.

En el caso que se requiera hacer un fuerte uso de componentes que estén ligados al hardware seleccionado, **el desarrollo de una aplicación nativa es lo ideal**. Por ejemplo, sacar partido al uso de cámaras, o de dispositivos de navegación por satélite GNSS.

Por otro lado, poner foco en soluciones multiplataforma, nos permite llegar a **más usuarios y cubrir una mayor diversidad de sistemas operativos**. Que en otro caso, representaría desarrollar una misma aplicación y adaptarla a distintos sistemas operativos, teniendo que invertir tiempo, recursos especializados y todo lo que conlleva al desarrollo de software.

Dar respuesta en cuanto al desarrollo de **aplicaciones nativas versus aplicaciones multiplataforma, es un tema ligado finalmente a la funcionalidad y al público** al que se desee llegar. Para esto, una solución posible es el uso de herramientas de prototipado, las cuales se encargan de traducir a las especificaciones a plataformas específicas. Algunas alternativas como IONIC [24], Flutter [25], React Native [26] facilitan la creación de aplicaciones multiplataforma. Lo mencionado sirve para dar un conocimiento general al estudiante, pero dichas herramientas quedan fuera del

alcance de este módulo, **el cuál se centra en el uso del sistema operativo Android y Java**. Una visión general sobre programación para dispositivos móviles se provee en la Figura 15.

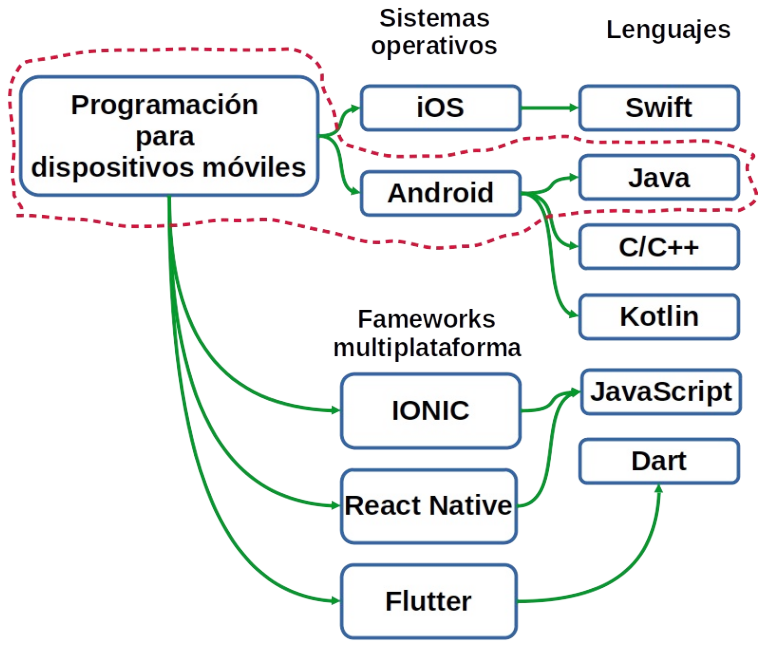

Figura 15: Herramienta utilizadas en la programación de aplicaciones móviles. En líneas punteadas de color rojo, lo que abarca este material.

Más allá del ecosistema ofrecido por Google para aplicaciones para Android, los desarrolladores deberán tener en cuenta que las aplicaciones puedan ejecutarse en **distintos tipos de dispositivos y pantallas**. Se debe proveer **compatibilidad con versiones**

anteriores del sistema operativo, velar por un **buen desempeño** del código fuente y ofrecer actualizaciones constantes.

Entorno de desarrollo Android

Android Studio es la herramienta de desarrollo de aplicaciones oficial para Android. Este entorno de desarrollo (*Integrated Development Environment (IDE)*), basa su editor de código en las herramientas de desarrollo Intellij IDEA (Jetbrains Company, Prague, Czech Republic) [3]. Ofrece ventajas tales como un ambiente unificado para desarrollo, emulador de dispositivos móviles, integración con Github y plantillas de ejemplos, entre otros [27]. La Figura 16 muestra la ventana principal de trabajo del entorno Android Studio.

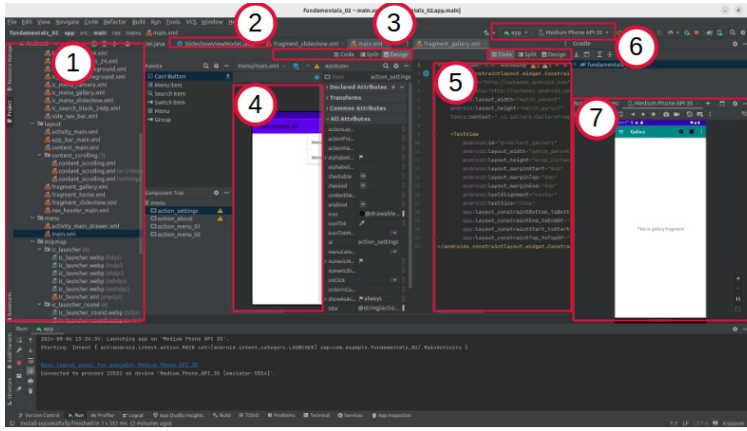

Figura 16: Entorno de trabajo de Android Studio. (1) Ventana de proyecto. (2) Pestaña de archivo. (3) Botones de visualización. (4) Visualización en modo diseño. (5) Visualización en modo código. (6) Ejecución de aplicación. (7) Ventana del emulador.

En la imagen, se visualizan distintas áreas lógicas, que brindan el acceso a las funcionalidades del IDE. En este capítulo se citan las más habituales, pero el lector podrá encontrar información complementaria en los materiales anexos [27], [28]. A continuación, se presentan las áreas lógicas principales:

- **Ventana de proyecto** (Figura 16.1). Permite ver las jerarquías de componentes dentro de un proyecto. En ella se pueden localizar carpetas, código fuente y componentes de diseño.
- **Pestaña de archivo** (Figura 16.2). Indica el archivo sobre el cual el usuario trabaja.
- **Botones de visualización** (Figura 16.3). Estos botones se activan según el tipo de archivo abierto. Para este caso particular, se ofrecen visualizaciones de **código, diseño y una combinación de ambas**.
- **Visualización en modo diseño** (Figura 16.4). Permite ver los componentes de una interfaz de usuario en modo diseño, el usuario puede agregar / eliminar componentes utilizando la interfaz. A la derecha se visualizan las propiedades de los elementos seleccionados.
- **Visualización en modo código** (Figura 16.5). Permite ver el código fuente del archivo seleccionado. En el caso de la figura, se visualiza un fragmento de código en .xml.
- **Ejecución de aplicación** (Figura 16.6) y ventana del emulador (Figura 16.7), se corresponden a la ejecución de

la aplicación y al emulador de teléfono móvil seleccionado para desarrollo.

Estructura de los Proyectos en Android Studio

La **estructura de los proyectos en Android Studio** contiene una serie de recursos tales como archivos de **código fuente, diseños de pantallas, definiciones de configuraciones [29]**. Estos recursos incluyen módulos de la aplicación y librerías propias de Google. A continuación se citan los elementos principales presentes en un proyecto:

- **Manifests.** Almacena el archivo AndroidManifest.xml, el cual contiene definiciones del proyecto en formato xml. Parte del contenido de este archivo incluye: actividades, permisos, definición de conexiones y recepción de información.
- **Java.** Es una carpeta que organiza código fuente por paquetes, sea en lenguaje Java o Kotlin.
- **Res.** Almacena almacena recursos que son utilizados en la aplicación tales como cadenas de texto **(values)**, imágenes e iconos **(drawable)**, diseños de interfaces de usuarios **(layout)**.

La ventaja de la estructura presentada, es que las aplicaciones pueden ser definidas de manera modular, es decir las funciones para interfaces de usuarios, definiciones del proyecto y código fuente se encuentran separadas, mejorando así la legibilidad del código.

Una aplicación Android usualmente consiste en cuatro componentes:

- Actividades (Activity).
- Intent.
- Servicio (Service).
- Proveedor de contenido (Content provider).

Las **actividades** reciben las interacciones de usuario por medio de una interfaz gráfica y actúan como un punto de entrada [30], las mismas son administradas según un ciclo de vida [31]. Cada actividad es una clase separada que se extiende e implementa a partir de una **clase base**. Un proyecto Android puede contener una o varias actividades.

Un **Intent** es una clase especial que permite encapsular las solicitudes a una actividad. Se los utiliza para el paso de parámetros entre actividades.

Un **servicio**, es una clase residente en el sistema que se encarga de realizar tareas en segundo plano. Habitualmente, se programan como servicios tareas que requieren espera o acceso a contenido externo.

Los **proveedores de contenido** son clases utilizadas para manejar el acceso a datos persistentes, sean estos archivos o bases de datos.

En la Figura 17 se muestran las relaciones existentes entre componentes de un proyecto Android, en este ejemplo, se presenta

una actividad con un botón, acompañado de las definiciones correspondientes de métodos y variables.

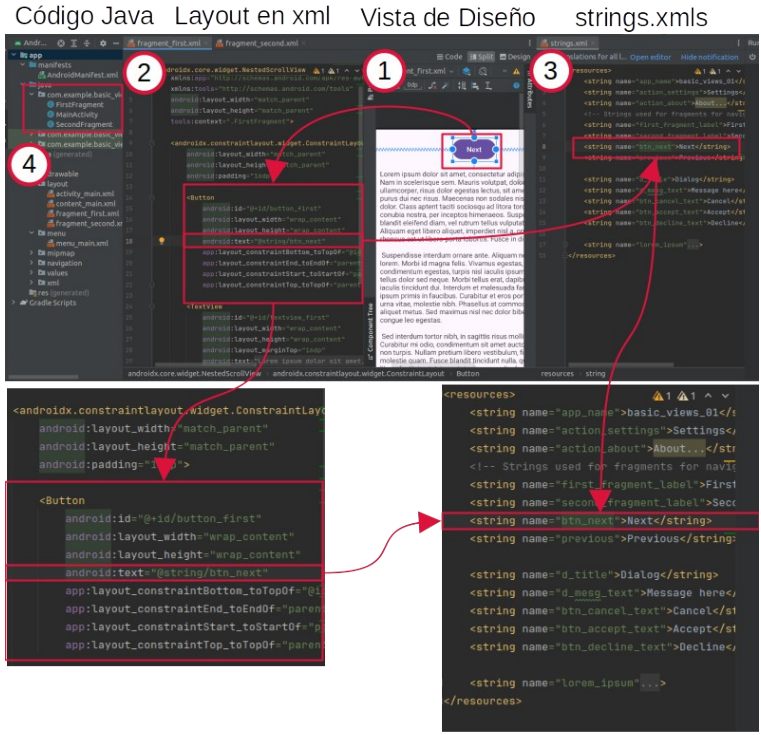

Figura 17: Relación de archivos y componentes de pantalla en un proyecto Android. (1) Botón en modo de diseño. (2) Código en XML del botón y la referencia a visualizar. (3) Archivo con definiciones de cadenas. (4) Panel de árbol con códigos fuentes para las vistas.

Copiar proyectos Android con el IDE

Esta es una operación útil, la cual será utilizada a lo largo del desarrollo de clases para realizar ejercicios. En este apartado se guía

al lector con los pasos necesarios para copiar un proyecto Android. A continuación, se enumera una serie de pasos para crear el proyecto **“basic_views_04”** a partir del proyecto base **“basic_views_01”**.

1. Copiar la carpeta **“basic_views_01”** a **“basic_views_04”**
2. Modificar el archivo **settings.gradle.kts**

   ```
   rootProject.name = "basic_views_04"
   include(":app")
   ```

3. Cambiar en el archivo **settings.gradle.kts** las líneas:

   ```
   namespace = "com.example.basic_views_03"
   applicationId = "com.example.basic_views_03"
   ```

4. Elegir la opción **“Open Project”** en el IDE. Hacer clic en botón Trust Project.
5. Clic derecho sobre la ventana del proyecto (lado izquierdo del IDE), seleccionar la carpeta “java” y elegir la opción **“Refactor”**. Cambiar el campo a **“basic_views_04”**.
6. Reemplazar los valores dentro de los archivos por el nuevo nombre de proyecto.

   Replace basic_views_01 por basic_views_04
7. Cambiar el nombre de la aplicación en el archivo **“strings.xml”**.

   ```
   <string name="app_name">basic_views_04</string>
   ```

8. Menú **“Build”** seguido de opción **“Clean Project”**.
9. Menú **“Build”** seguido de opción **“Rebuild Project”**.
10. Menú **“Run”** para ejecutar la aplicación.

Para el ejercicio se debe seguir los siguientes pasos:

1. Definir las cadenas que se utilizarán y almacenarlas en **“string.xml”**.
2. Definir los layouts de los fragment y organizarlos.
3. Definir las acciones en **“navigation/nav_graph.xml”**.
4. Implementar los métodos de salto en los botones.

Pasos para agregar un botón que cambie a otra actividad.

1. Agregar el botón en el layout.
2. Agregar en **“nav_graph.xml”** la acción.
3. Implementar el método en la clase.

Paso de datos entre actividades

Cada actividad contiene sus datos y métodos propios, el usuario puede interactuar con la misma por medio de la pantalla. En ciertas ocasiones, surge la necesidad de transferir datos de una actividad a otra, para que estos sean procesados. En este apartado se ofrecen dos ejemplos, el primero basado en actividades simples (**parameters_intent_01**) y el segundo combinado con la clase **“Navigation” (parameters_activities_02)**. Información complementaria puede ser consultada en [32].

En líneas generales, se implementa código en una a**ctividad emisora** que recibe datos del usuario y los envía a una a**ctividad receptora** que procesa los datos recibidos. Entonces, el código puede ser analizado teniendo en cuenta la actividad emisora y receptora.

Dentro de la **actividad inicial**:

1. Crear y agregar datos en el objeto **Intent** (Figura 18).

```
Intent intent = new Intent(view.getContext(), Activity02.class);
intent.putExtra("myTextValue", myString);
intent.putExtra("myIntegerValue", myInteger);
```

Figura 18: Crear y agregar datos mediante Intent

2. Llamar a la actividad receptora.

```
startActivity(intent);
```

Figura 19: Llamar a la actividad receptora

Dentro de la **actividad receptora**:

1. Obtener el objeto **Intent** (Figura 20). Esto puede hacerse en el método OnCreate.

```
Bundle externalPar = getIntent().getExtras();
```

Figura 20: Obtener Intent n actividad receptora

2. Extraer los datos del objeto **Intent** (Figura 21).

```
String myStringP = externalPar.getString("myTextValue");
int myIntegerP = externalPar.getInt("myIntegerValue");
```

Figura 21: Obtener Intent n actividad receptora

Para el caso de la clase **Navigation**, el paso de parámetros se realiza utilizando objetos del **tipo Bundle**. La idea, es almacenar datos en una variable **Bundle** de la **actividad emisora**, la cual será tratada como argumento de entrada en la **actividad receptora**. Se sugiere complementar el aprendizaje haciendo uso del proyecto ejemplo **"parameters_activities_01"** y consultando la documentación oficial [33].

Los pasos que sugeridos para implementar el código en la **actividad emisora** son:

- Obtener los valores de los widgets dentro de la actividad (Figura 22).

```
String aNumberStr = binding.editTextNumber.getText().toString();
int aNumber = Integer.parseInt(aNumberStr);
String aName = binding.editTextText.getText().toString();
```

Figura 22: Cargar valores y aplicar conversiones

- Cargar el objeto Bundle (Figura 23).

```
Bundle bundle = new Bundle();
bundle.putInt("aNumberB",aNumber);
bundle.putString("aNameB",aName);
```

Figura 23: Creación del objeto Bundle.

- Invocar al método Navigate (Figura 23).

```
NavController navController =
NavHostFragment.findNavController(Fragment01.this);

navController.navigate(R.id.action_FirstFragment_to_SecondFragment,
bundle);
```

Figura 24: Llamado a Navigate.

En la **actividad receptora** se implementan los siguientes pasos:

- Obtener los argumentos de entrada. Aquí recupera valores del tipo **int** y **String** desde los argumentos de entrada (Figura 25).

```
Integer aIntArg = getArguments().getInt("aNumberB");
String aIntStrArg = Integer.toString(aIntArg);
String  aStrArg = getArguments().getString("aNameB");
```

Figura 25: Recibir argumentos.

- Cargarlos valores a los widgets de la vista (Figura 26).

```
binding.editTextNumber.setText(aIntStrArg);
binding.editTextText.setText(aStrArg);
```

Figura 26: Carga de valores en widgets.

Interfaz de Usuario (FALTA)

ddd

Fundamentos de las aplicaciones Android (FALTA)

ddd

Interacción con otras aplicaciones (FALTA)

dd

Ejercicios propuestos

Los ojetivos de estos ejercicios son:

- Que el alumno instale el entorno de programación y se familiarice con las herramientas.
- Conocer los componentes más utilizados en una aplicación Android mediante la práctica.

1. Instalar Android Studio y configurar el ambiente de trabajo.
2. Utilizar la herramienta Debug para hacer seguimiento de la ejecución de un ejemplo.
3. Crear un menu con actividades desplegables.
4.

Material Complementario

- Guías y ejemplos para desarrolladores [34].
- Arquitectura de Android [35].
- Android Developer Fundamentals (Version 2) – Concepts [36].

4 Desarrollo de aplicaciones en Android – Nivel Avanzado

Almacenamiento local y en la nube (FALTA)

aaa

Contents providers (FALTA)

aaa

Google Play Services API (FALTA)

aaa

Notificaciones (FALTA)

aaa

Componentes de Material Designing (FALTA)

Ssss

Distribución de la aplicación (FALTA)

Distribución y empaquetado

[https://developer.android.com/studio/publish]

5 Referencias

[1] The Apache Software Foundation, “Apache NetBeans.” Accessed: May 08, 2024. [Online]. Available: https://netbeans.apache.org/

[2] Eclipse Foundation, “Eclipse Installer 2024-06 R.” Accessed: May 08, 2024. [Online]. Available: https://www.eclipse.org/downloads/packages/installer

[3] “IntelliJ IDEA. The Leading Java and Kotlin IDE.” Accessed: Aug. 05, 2024. [Online]. Available: https://www.jetbrains.com/idea/

[4] O. J. D. Team, “The Java Tutorials.” Accessed: Jul. 31, 2024. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/

[5] R. Sedgewick and K. Wayne, *Algorithms*. Addison-wesley professional, 2011.

[6] O. J. D. Team, “Creating Primitive Type Variables in Your Programs.” Accessed: Aug. 16, 2024. [Online]. Available: https://dev.java/learn/language-basics/primitive-types/

[7] R. Sedgewick, “Built-in Data Types.” Accessed: Aug. 16, 2024. [Online]. Available: https://introcs.cs.princeton.edu/java/12types/

[8] R. Sedgewick and K. Wayne, “Java Programming Cheatsheet.” Accessed: Aug. 15, 2024. [Online]. Available: https://introcs.cs.princeton.edu/java/11cheatsheet/

[9] Google, “Google Java Style Guide.” Accessed: Aug. 13, 2024. [Online]. Available: https://google.github.io/styleguide/javaguide.html

[10] O. J. D. Team, “Learn Java.” Accessed: Aug. 13, 2024. [Online]. Available: https://dev.java/learn/

[11] O. J. D. Team, “Lesson: Introduction to Collections.” Accessed: Oct. 04, 2024. [Online]. Available: https://docs.oracle.com/javase/tutorial/collections/intro/index.html

[12] D. Ocean, “Collections in Java - Everything You MUST Know.” Accessed: Oct. 04, 2024. [Online]. Available: https://www.digitalocean.com/community/tutorials/collections-in-java-tutorial

[13] O. J. D. Team, “Objects, Classes, Interfaces, Packages, and Inheritance.” Accessed: Aug. 19, 2024. [Online]. Available: https://dev.java/learn/oop/

[14] S. Mednieks, L. Dorin, G. B. Meike, and M. Nakamura, Eds., *Programming Android*. in Deitel Developer Series. Sebastopol, CA: O’Reilly Media,

2011. [Online]. Available: https://www.finna.fi/Record/helmet.2126215

[15] O. J. D. Team, “What Is an Exception?” Accessed: Sep. 26, 2024. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

[16] O. J. D. Team, “Getting Started with Java Certification.” Accessed: Aug. 23, 2024. [Online]. Available: https://dev.java/learn/java-cert-overview/

[17] Wikipedia, “Mobile operating Systems.” Accessed: Aug. 05, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Mobile_operating_system

[18] Wikipedia, “Comparison of mobile operating systems.” Accessed: Aug. 05, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_mobile_operating_systems#About_OS

[19] Wikipedia, “Android.” Accessed: Aug. 05, 2024. [Online]. Available: https://es.wikipedia.org/wiki/Android

[20] Statcounter GlobalStats, “Mobile Operating System Market Share Worldwide.” Accessed: Aug. 05, 2024. [Online]. Available: https://gs.statcounter.com/os-

market-share/mobile/worldwide#monthly-202406-202406-map

[21] G. D. T. Team, “Android’s Kotlin-first approach.” Accessed: Aug. 05, 2024. [Online]. Available: https://developer.android.com/kotlin/first

[22] G. D. T. Team, “Get started with Android.” Accessed: Aug. 05, 2024. [Online]. Available: https://developer.android.com/get-started/

[23] Jetbrains, “Kotlin. Concise. Multiplatform. Fun.” Accessed: Aug. 05, 2024. [Online]. Available: https://kotlinlang.org/

[24] “ionic Framework The Mobile SDK for the web.” Accessed: Sep. 30, 2024. [Online]. Available: https://ionicframework.com/

[25] “Flutter.” Accessed: Sep. 30, 2024. [Online]. Available: https://flutter.dev/

[26] “React Native.” Accessed: Sep. 30, 2024. [Online]. Available: https://reactnative.dev/

[27] G. D. T. Team, “Meet Android Studio.” Accessed: Aug. 05, 2024. [Online]. Available: https://developer.android.com/studio/

[28] G. D. T. Team, “Get to know the Android Studio UI.” Accessed: Aug. 05, 2024. [Online]. Available:

https://developer.android.com/studio/intro/user-interface

[29] G. D. T. Team, “Projects Overview.” Accessed: Sep. 30, 2024. [Online]. Available: https://developer.android.com/studio/projects

[30] “Introduction to Activities.” Accessed: Sep. 30, 2024. [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities

[31] G. D. T. Team, “The activity lifecycle.” Accessed: Sep. 30, 2024. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle

[32] G. D. T. Team, “Activities and Intents.” Accessed: Sep. 30, 2024. [Online]. Available: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-1-get-started/lesson-2-activities-and-intents/2-1-c-activities-and-intents/2-1-c-activities-and-intents.html

[33] G. D. T. Team, “Pass data between destinations with Bundle objects.” Accessed: Sep. 14, 2024. [Online]. Available: https://developer.android.com/guide/navigation/use-graph/pass-data

[34] G. D. T. Team, “Developer Guides.” [Online]. Available: https://developer.android.com/guide

[35] G. D. T. Team, “Platform architecture.” Accessed: Aug. 05, 2024. [Online]. Available: https://developer.android.com/guide/platform/index.html

[36] G. D. T. Team, “Android Developer Fundamentals (Version 2) — Concepts.” Accessed: Jul. 31, 2024. [Online]. Available: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/