

Libro

Luis Jaramillo

2024-08-11

Table of contents

Programación Orientada a Objetos	5
I Unidad 1: Nombre de la Unidad	7
1 Introduction	8
II Unidad 2: Entorno de Desarrollo	9
2 ENTORNO DE DESARROLLO.	10
III Unidad 3: Paradigmas de Programación	15
3 Introducción a los Paradigmas de Programación	16
3.1 Paradigma Imperativo	16
3.2 Paradigma Declarativo	17
3.3 Paradigma Orientado a Objetos	17
3.4 Paradigma Funcional.	19
IV Unidad 4: Conceptos Generales de la POO	21
4 Conceptos Generales de la POO	22
4.1 Principios Fundamentales de la POO	22
4.2 1 Clases y Objetos	22
4.2.1 Clases	22
4.2.2 Objetos	23
4.3 2. Encapsulación	23
4.3.1 Ejemplo de Encapsulación	24
4.4 3 Herencia	24
5 Polimorfismo	27
5.0.1 Ejemplo de Polimorfismo	28

V	Unidad 5: Modelamiento de clases y objetos	29
6	Modelamiento de clases y objetos.	30
6.1	TIPOS DE RELACIONES DIAGRAMAS DE CASOS DE USOS.	32
6.2	Elementos de un diagrama de casos de uso	32
6.2.1	Actores	33
6.3	Relaciones	34
6.4	Tipos de relaciones.	36
6.5	Tipos de relaciones en diagramas de clases UML:	39
6.5.1	Relación (Línea continua con flecha)	39
6.5.2	Implementación (Línea continua con triángulo)	40
6.5.3	Composición (Línea continua con diamante lleno)	40
6.5.4	Asociación (Línea continua)	41
6.5.5	Herencia (Línea discontinua con triángulo)	41
6.5.6	Dependencia (Línea discontinua con flecha)	41
6.5.7	Agregación (Línea continua con diamante vacío)	41
6.6	Asociación:	42
6.7	Agregación:	42
6.8	Composición:	43
6.9	Multiplicidad.	44
6.10	PlantiumUML herramienta	45
6.11	PlantiumUML Video	46
6.12	Video	46
6.13	TEST	46
VI	Unidad 6: Código Limpio	47
7	Nombres Claros y Descriptivos	48
7.1	Funciones y Métodos Los nombres de funciones y métodos deben describir lo que hacen. Es una buena práctica que los nombres sean verbos o frases verbales.	48
7.2	Funciones Simples y de Responsabilidad Única	49
7.3	Evitar Comentarios Innecesarios	49
7.4	Evitar Código Duplicado	50
7.5	Control de Excepciones y Errores	50
7.6	Uso de Formato Consistente	51
7.6.1	Ejemplo	51
7.7	Reducir el Número de Parámetros	52
7.8	Pruebas Unitarias	53

VII Unidad 7: Estructura General de un Programa	54
8 Estructura General de un Programa en JAVA	55
8.1 ESTRUCTURA GENERAL DE UN PROGRAMA EN JAVA.	59
8.2 Video	64
VIIIUnidad 8: Lectura y Escritura de Datos por consola	65
9 Lectura y Escritura de Datos por Consola	66
9.1 Introducción a la Entrada y Salida en Consola	66
9.2 Entrada de datos	66
9.2.1 Clase Scanner	66
9.2.2 Uso de BufferedReader	69
9.3 Escritura de Datos en la Consola	69
9.3.1 Uso de System.out.println	69
IX Unidad 9: Manejor de Exepciones	71
10 Manejo de excepciones.	72
10.1 Introducción	72
10.2 Conceptos Básicos	72
10.2.1 ¿Qué es una excepción?	72
10.2.2 Excepciones Propias de java	73
10.2.3 Jerarquía de Excepciones	74
10.2.4 Ejemplo de Excepciones	75
10.2.5 Ejemplo de Excepciones Manejando TRY-CATCH-FINALLY .	77
10.2.6 Captura de Múltiples Excepciones:	78
10.2.7 Diferencias Clave entre los programas anteriores	81
11 Summary	82
References	83

Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de “objetos”, los cuales son instancias de clases. En POO, un “objeto” puede entenderse como una entidad que encapsula datos y funciones que operan sobre esos datos. Este enfoque permite modelar el mundo real de manera más efectiva, ya que se pueden representar entidades y sus interacciones de manera más natural.

En POO, una “clase” es un plano o plantilla para crear objetos. Define la estructura y el comportamiento de los objetos que se crearán a partir de ella. Los objetos son instancias concretas de una clase, y cada objeto puede tener sus propios datos (llamados atributos) y funciones (llamadas métodos).

Los cuatro principios fundamentales de la POO son la encapsulación, la herencia, el polimorfismo y la abstracción. La “**encapsulación**” implica la ocultación de los detalles internos de un objeto y la exposición solo de lo necesario. La “**herencia**” permite la creación de nuevas clases basadas en clases existentes, heredando sus atributos y métodos. El “**polimorfismo**” permite que un objeto pueda tomar varias formas, es decir, un mismo método puede comportarse de manera diferente según el contexto. La “**abstracción**” consiste en simplificar la representación de un objeto centrándose en los aspectos esenciales y omitiendo los detalles innecesarios. Para el siguiente documento revisaremos las siguientes temáticas:

Primer Parcial

- Control de versionamiento
- Entorno de Desarrollo
- Paradigmas de programación
- Conceptos Generales de la POO
- Modelamiento de clases y objetos
- Código limpio
- Estructura General de un Programa - _____
- Lectura y escritura de datos por consola
- Manejo de Excepciones
- Encapsulamiento
- Constructores
- Métodos Getter y Setters
- Persistencia de datos
- Arreglos y Colecciones

2 Parcial - Relaciones entre clases (Generalización/Especialización) - Revisiones de Código - Polimorfismo - Interfaces de Programación - Modelo Vista Controlador - Bases de Datos no SQL - Componentes y Objetos Gráficos

3er Parcial - Principios SOLID - Modularidad - Introducción a Patrones de diseño - Patrones de creación - Patrones de Estructura - Patrones de Comportamiento

Part I

Unidad 1: Nombre de la Unidad

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

Part II

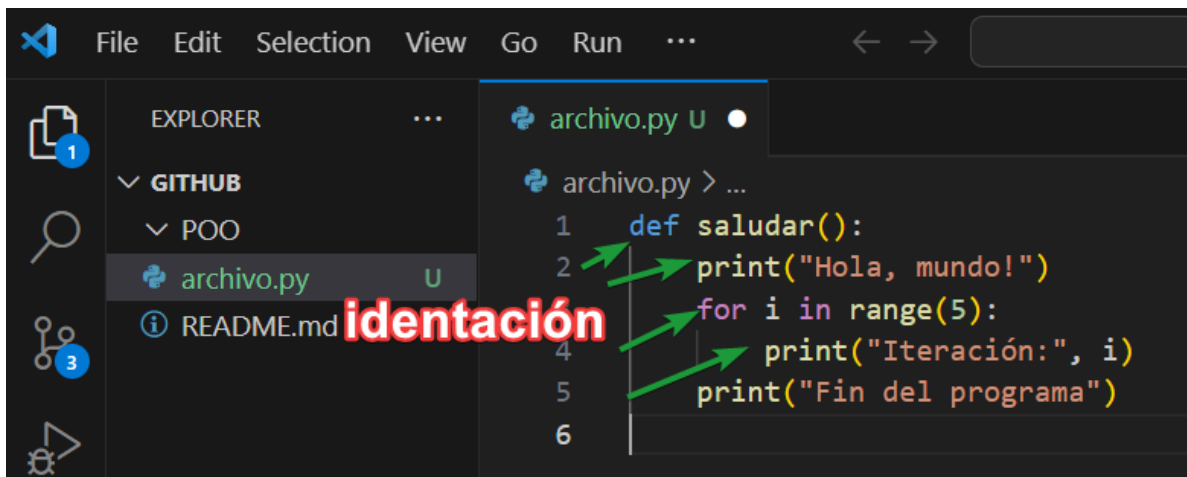
Unidad 2: Entorno de Desarrollo

2 ENTORNO DE DESARROLLO.

Un entorno de desarrollo, comúnmente conocido como IDE por sus siglas en inglés (Integrated Development Environment), es una herramienta de software integral que se utiliza para facilitar la programación y el desarrollo de software. Un IDE proporciona un conjunto de características y herramientas que ayudan a los programadores a escribir, depurar, compilar y administrar código de manera más eficiente, en otras palabras es un paquete de software que combina todas las herramientas de desarrollo en una única interfaz gráfica de usuario (GUI). Algunos ejemplos de IDE más utilizados son NetBeans, Microsoft Visual Studio, Adobe Flex Builder y Eclipse. Algunos componentes comunes de un entorno de desarrollo incluyen:

Editor de código: Ofrece funciones de edición de texto avanzadas, como resaltado de sintaxis, autoindentación y autocompletado, lo que facilita la escritura de código.

La Identación se refiere al formato visual utilizado para estructurar y organizar el código fuente en un lenguaje de programación.



Depurador: Permite a los programadores identificar y corregir errores en su código al proporcionar herramientas para establecer puntos de interrupción, inspeccionar variables y seguir la ejecución del programa paso a paso.

Compilador/Intérprete: Proporciona la capacidad de compilar y ejecutar el código directamente desde el IDE. Algunos IDE también son específicos para un lenguaje y pueden incluir un intérprete para ejecutar el código directamente.

Gestión de proyectos: Permite organizar y administrar proyectos de desarrollo, lo que incluye la creación, apertura y gestión de archivos y directorios.

Herramientas de versionado: Puede integrar sistemas de control de versiones, como Git, para realizar un seguimiento de los cambios en el código y colaborar con otros desarrolladores.

Asistencia en el desarrollo: Ofrece sugerencias y advertencias en tiempo real mientras se escribe código, lo que ayuda a detectar posibles errores y mejorar la calidad del código.

Herramientas de construcción: Facilita la construcción y automatización de tareas, como la compilación, la generación de documentación y la ejecución de pruebas.

Herramientas de depuración visual: Algunos IDE proporcionan una interfaz gráfica para depurar aplicaciones de manera más visual, lo que puede incluir representaciones gráficas de estructuras de datos y flujos de programa.

Integración con SDK y bibliotecas: Puede integrar SDK (kits de desarrollo de software) y bibliotecas relevantes para el lenguaje o plataforma de desarrollo específico.

Finalización de código: conocida como prompt de código, esta función se utiliza para analizar todo el código y para identificar e insertar los componentes de código que faltan. Su uso permite ahorrar tiempo y minimizar los fallos o errores

Por ejemplo, un SDK de Android proporciona las herramientas y recursos necesarios para desarrollar aplicaciones para dispositivos Android, incluyendo bibliotecas para interactuar con el hardware y el sistema operativo, herramientas de depuración, un emulador de Android para probar aplicaciones y documentación detallada.

Los IDE varían según el lenguaje de programación y la plataforma de desarrollo. Ejemplos populares de IDE incluyen Visual Studio (para desarrollo en C# y .NET), Eclipse (para Java), IntelliJ IDEA (para Java y otros lenguajes), PyCharm (para Python) y muchos otros. La elección de un IDE suele depender de las preferencias del desarrollador y del entorno de desarrollo específico en el que trabaje.

CARACTERÍSTICAS E INSTALACIÓN.

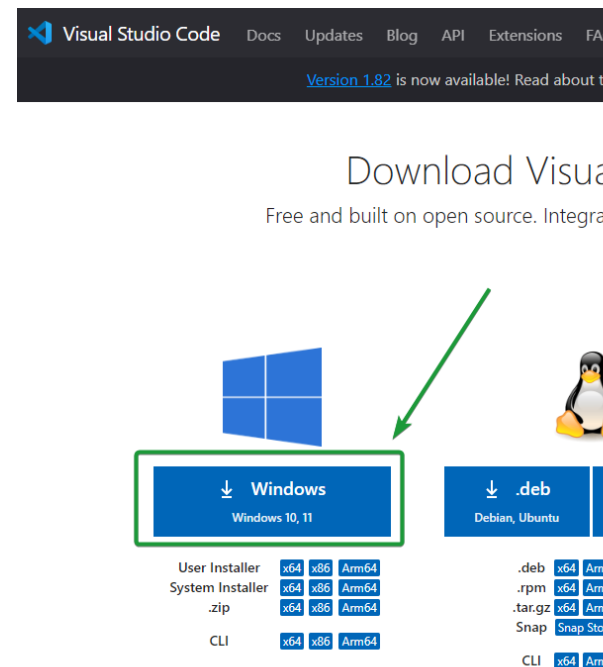
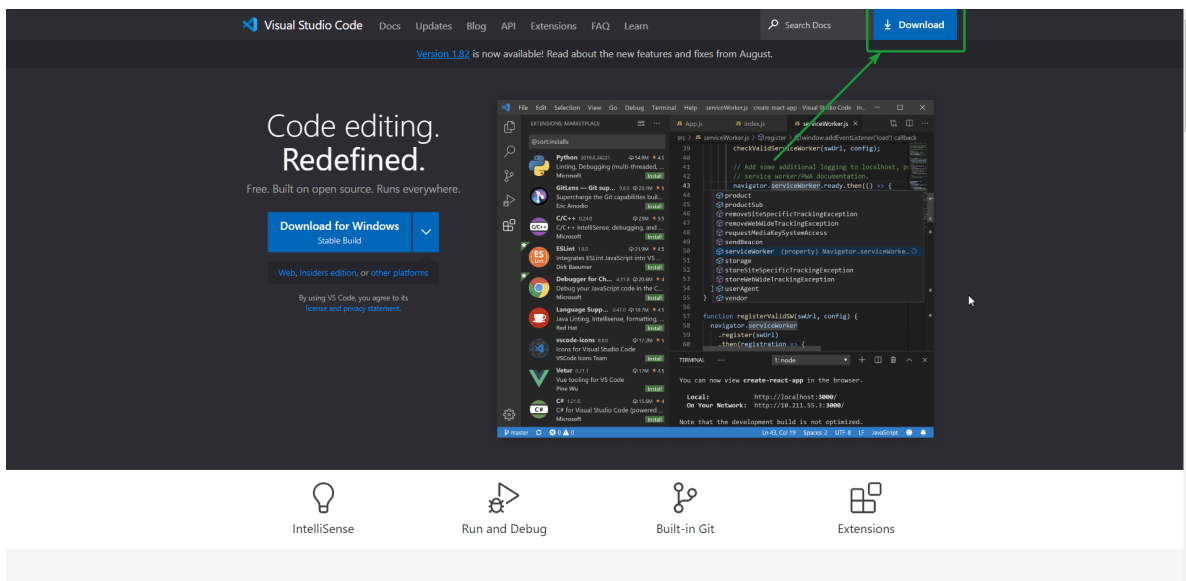
Para este curso se utilizará los lenguajes de programación de Python y Java, por lo que instalaremos dos entornos virtuales, para sacar el mayor provecho en el caso del lenguaje de programación Java.

El primer entorno virtual será Visual Studio Code  y el IDE para el lenguaje de programación será Netbeans IDE 16 .

Para la instalación del IDE Visual Studio Code hay que seguir los siguientes pasos.

Paso 1: Descarga de VS Code

Dirígete al sitio web oficial de Visual Studio Code en <https://code.visualstudio.com/> y busca un botón de descarga, generalmente ubicado en la página principal. Aquí deberías encontrar una imagen que representa el botón de descarga.

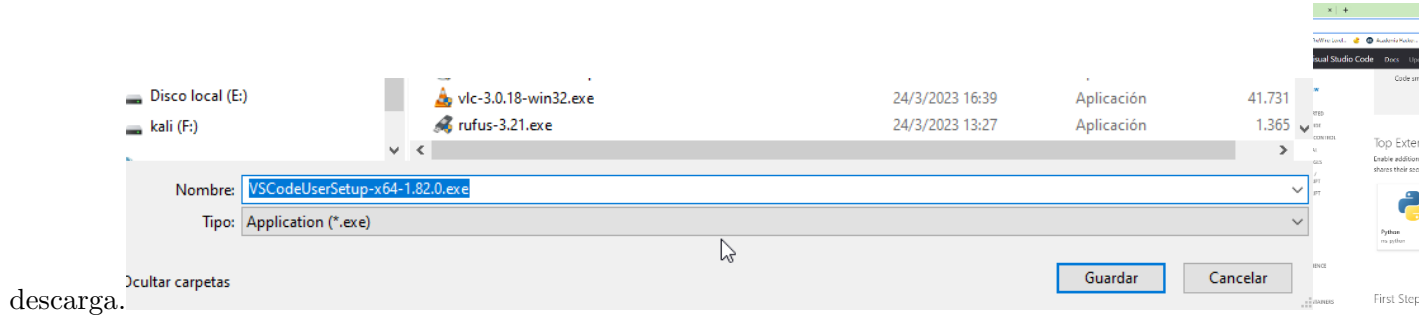


Paso 2: Selecciona tu Sistema Operativo, en mi caso Windows 10.

Paso 3: Descarga del Instalador

Después de seleccionar tu sistema operativo, deberías ser redirigido a una página que te

permitirá descargar el instalador específico para tu plataforma. Aquí encontrarás un enlace de descarga o un botón para iniciar la descarga, donde señalarás la carpeta y realizará la



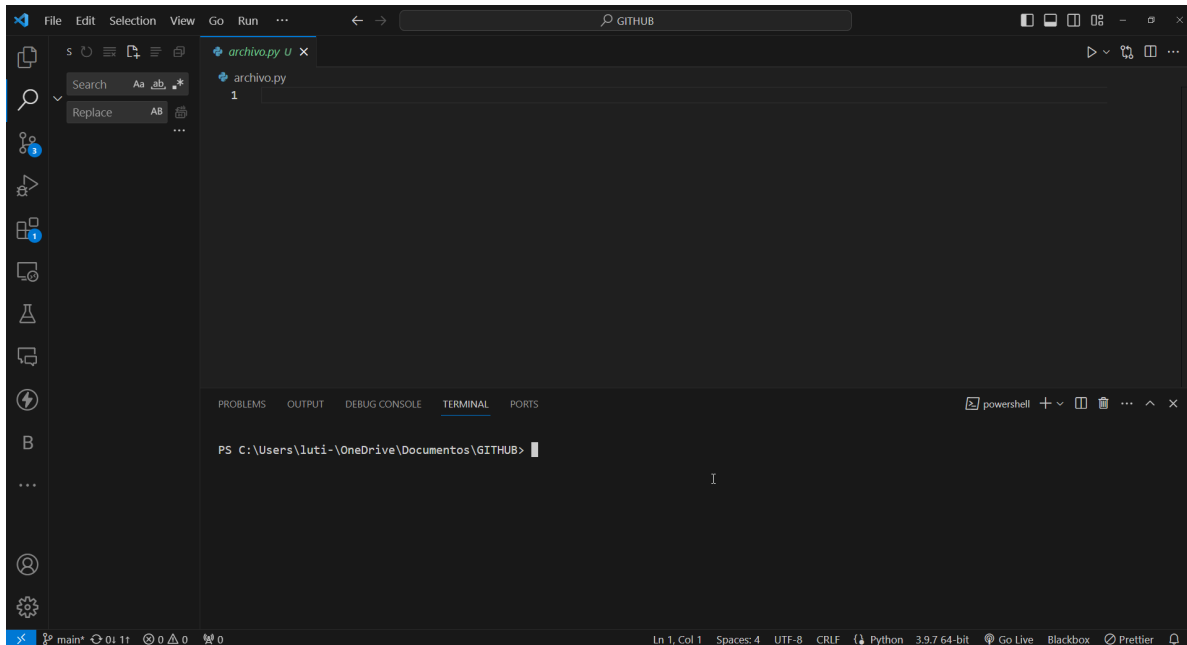
Paso 4: Ejecuta el Instalador

Una vez que se complete la descarga, ejecuta el instalador haciendo doble clic en él (en Windows) o siguiendo las instrucciones específicas para tu sistema operativo.

Durante la instalación, se te pueden presentar opciones de configuración. Estas opciones pueden variar según la versión y la plataforma. Generalmente, se te preguntará si deseas crear accesos directos en el escritorio o en el menú de inicio (en el caso de Windows), así como otras preferencias.

Paso 5: Finaliza la Instalación

Finalmente, cuando se complete la instalación, verás una pantalla que confirma que Visual Studio Code se ha instalado correctamente.



Paso 6: Configuración de los lenguajes.

La configuración de los lenguajes es muy importante ya que este nos permite al momento de ejecutar los archivos con sus diferentes extensiones que corran de manera correcta, para este curso instalaremos Python y luego netbeans por lo que los pasos a seguir son los siguientes:

Paso 1: Descargar Python: Ve al sitio web oficial de Python en <https://www.python.org/downloads/>.

En la página de descargas, verás las últimas versiones de Python disponibles. Elige la versión más reciente de Python 3 (por ejemplo, Python 3.9.6).

Desplázate hacia abajo hasta encontrar la sección de descargas de archivos ejecutables. Dependiendo de tu sistema operativo (Windows, macOS o Linux), selecciona el enlace de descarga adecuado. Para Windows, es posible que veas dos opciones: una para 32 bits y otra para 64 bits. Si tienes un sistema operativo de 64 bits, selecciona la versión de 64 bits.

Paso 2: Ejecutar el Instalador de Python: Una vez que se complete la descarga, haz doble clic en el archivo ejecutable que descargaste para iniciar el instalador de Python.

Asegúrate de marcar la casilla que dice “Add Python X.Y to PATH” (donde “X.Y” representa la versión de Python que estás instalando). Esto añadirá Python al PATH de tu sistema, lo que facilitará su uso desde la línea de comandos.

Haz clic en “Install Now” para comenzar la instalación.

Paso 3: Verificar la Instalación de Python: Abre una terminal o línea de comandos, Escribe el siguiente comando y presiona Enter:

```
bash
```

Copy code

```
python --version
```

Esto mostrará la versión de Python que has instalado, como “Python 3.9.6” o similar.

Instalación del Netbeans.

Part III

Unidad 3: Paradigmas de Programación

3 Introducción a los Paradigmas de Programación

Los paradigmas de programación son enfoques o estilos para resolver problemas mediante código. Cada paradigma tiene sus principios, ventajas y desventajas, y está diseñado para abordar problemas de diferentes maneras. En este documento, exploraremos cuatro paradigmas principales: imperativo, declarativo, orientado a objetos y funcional.

-

3.1 Paradigma Imperativo

El paradigma imperativo es uno de los enfoques más tradicionales. En él, el programador indica paso a paso cómo se debe realizar cada tarea.

- Características:
 - Enfoque en **cambiar el estado** mediante secuencias de instrucciones.
 - Uso intensivo de **bucles** y **estructuras de control de flujo** (if, else, for, while).
- Ejemplo en Java

En Java, este paradigma se utiliza ampliamente debido a su sintaxis estructurada.

```
public class SumaImperativa {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int i = 1; i <= 5; i++) {  
            sum += i;  
        }  
        System.out.println("La suma es: " + sum);  
    }  
}
```

-

3.2 Paradigma Declarativo

En el paradigma declarativo, el programador especifica **qué** se quiere lograr, no **cómo** hacerlo. Esto permite centrarse en el resultado y no en los pasos intermedios.

- Características:
 - Describe el resultado deseado en lugar de los pasos detallados.
 - Más conciso y fácil de leer para tareas específicas como consultas de bases de datos.
 - Ejemplo con Java y SQL

En Java, el uso de SQL en aplicaciones es un ejemplo de un enfoque declarativo.

```
SELECT nombre, edad FROM estudiantes WHERE edad > 18;
```

-

3.3 Paradigma Orientado a Objetos

La programación orientada a objetos (POO) es ampliamente utilizada en Java. Este paradigma organiza el código en “objetos”, que representan entidades del mundo real.

- Características:
 - Usa conceptos de **clases** y **objetos**.
 - Principios de **abstracción**, **encapsulación**, **herencia** y **polimorfismo**.
- Ejemplo en Java

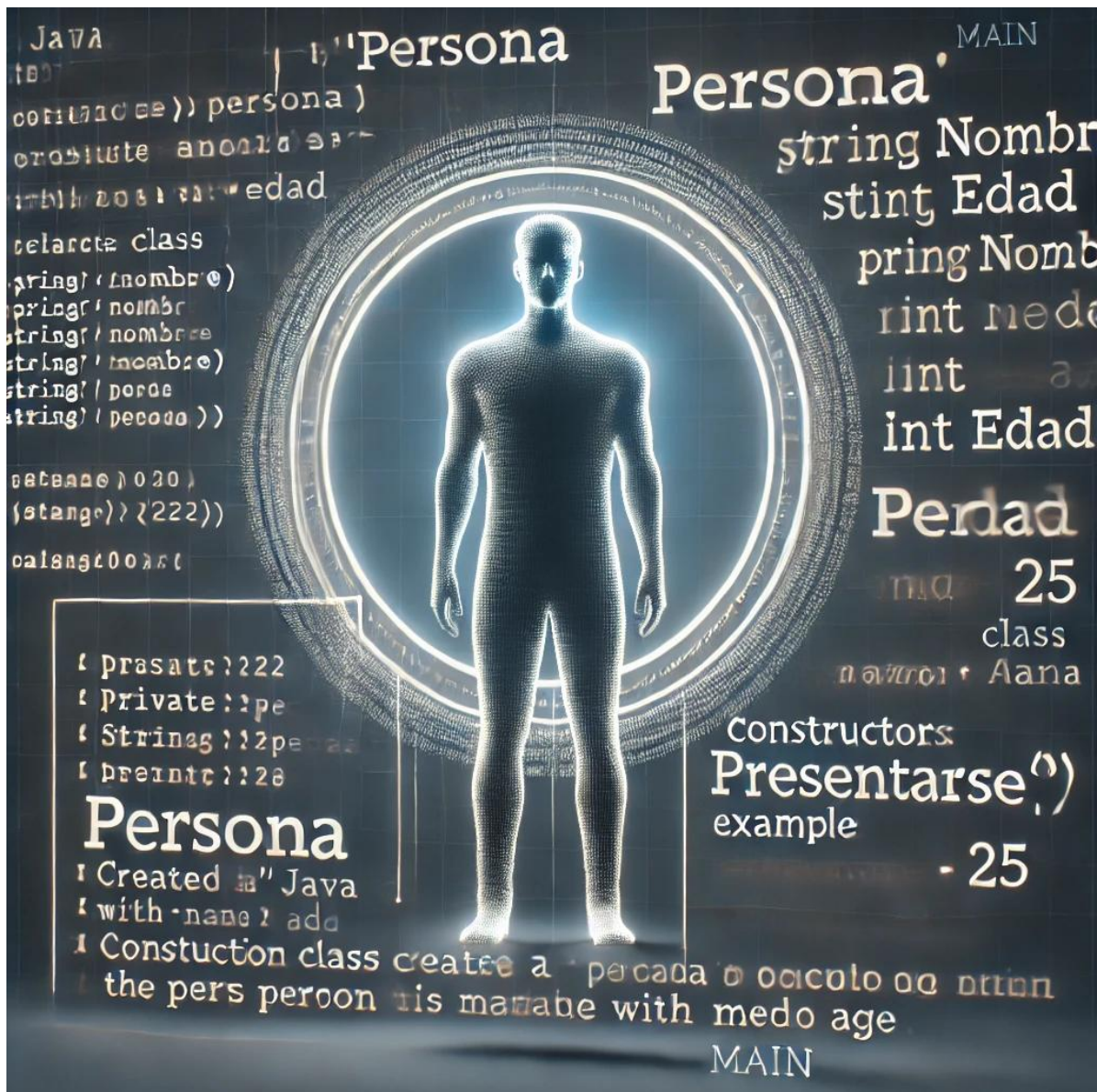
Aquí un ejemplo básico de POO en Java:

```
class Persona {
    String nombre;
    int edad;

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

    public void presentarse() {
        System.out.println("Hola, soy " + nombre + " y tengo " + edad + " años.");
    }
}
```

```
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Persona persona = new Persona("Ana", 25);  
        persona.presentarse();  
    }  
}
```



•

3.4 Paradigma Funcional.

El paradigma funcional trata las funciones como ciudadanos de primera clase. En este estilo, se evita el cambio de estado y las variables mutables.

- Características:

- Uso de **funciones puras** (sin efectos secundarios).
- Uso de **funciones de orden superior** y técnicas como el **mapeo** y el **filtrado**.
- Ejemplo en Java.

imagina que tienes una lista de números y quieres obtener solo los números pares y calcular el doble de cada uno de ellos. En el **paradigma funcional**, no describimos cada paso de cómo lograr esto con bucles o variables que cambien, sino que aplicamos funciones que transforman la lista de manera declarativa. 1. Primero, aplicamos una **función de filtro** para seleccionar solo los números pares. 2. Luego, aplicamos una **función de mapeo** para multiplicar cada número par por dos. 3. Finalmente, **obtenemos el resultado** como una nueva lista.

[Test](#)

Part IV

Unidad 4: Conceptos Generales de la POO

4 Conceptos Generales de la POO

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el código en torno a “objetos” que representan entidades del mundo real. La POO permite una estructura modular, reutilizable y escalable, haciendo el código más fácil de entender y mantener.

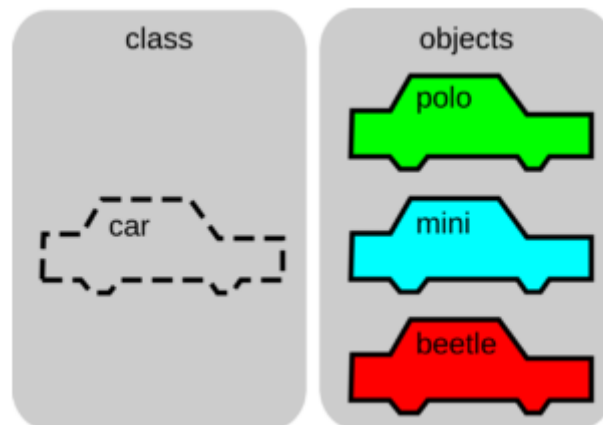
4.1 Principios Fundamentales de la POO

La POO se basa en cuatro principios clave: **Clases y Objetos**, **Encapsulación**, **Herencia** y **Polimorfismo**. A continuación, veremos cada uno en detalle.

4.2 1 Clases y Objetos

4.2.1 Clases

Una **clase** es una plantilla o modelo que define las propiedades y comportamientos de un tipo de objeto. Es como un plano que describe cómo debe ser un objeto.



```

public class Persona { // NOMBRE DE LA CLASE
    String nombre;      // ATRIBUTO
    int edad;

    public void presentarse() { // METODO
        System.out.println("Hola, soy " + nombre + " y tengo " + edad + " años.");
    }
}

```

4.2.2 Objetos

Un **objeto** es una instancia concreta de una clase. Cuando se crea un objeto, se asigna un espacio en memoria y se puede interactuar con sus atributos y métodos.

```

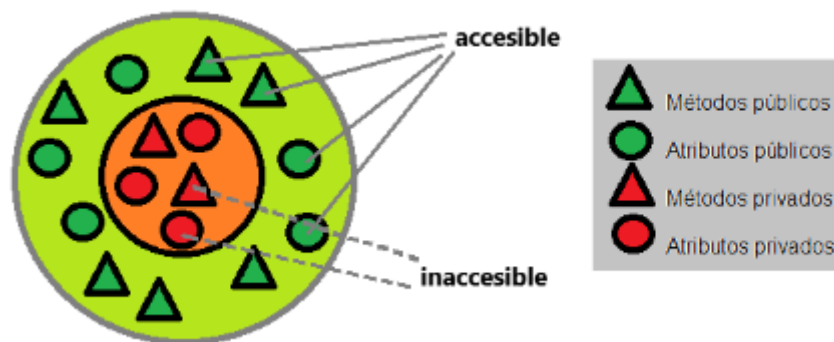
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.nombre = "Ana";
        persona.edad = 25;
        persona.presentarse();
    }
}

```

En este ejemplo, `persona` es un objeto de la clase `Persona` con atributos `nombre` y `edad`.

4.3 2. Encapsulación

La **encapsulación** consiste en restringir el acceso directo a ciertos datos de un objeto, protegiendo sus atributos para que solo se modifiquen a través de métodos específicos.



4.3.1 Ejemplo de Encapsulación

```
class Persona {  
    public String nombre;  
    private int edad;  
  
    public Persona(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getEdad() {  
        return edad;  
    }  
  
    public void setEdad(int edad) {  
        this.edad = edad;  
    }  
}
```

En este caso, el atributo `nombre` es público y `edad` es privado, y solo se pueden acceder mediante los métodos `getEdad()` y `setEdad()`.

4.4 3 Herencia

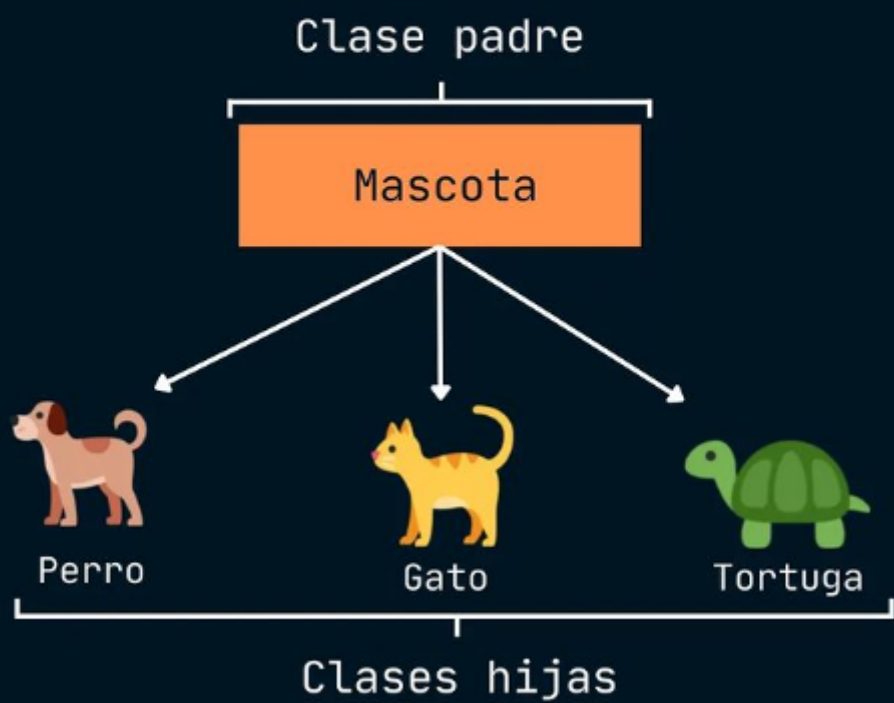
La **herencia** permite que una clase (subclase) herede atributos y métodos de otra clase (superclase), permitiendo reutilizar código y extender funcionalidades.

Ejemplo de Herencia

```
class Animal {  
    public String nombre;  
    public Animal (String nombre){  
        this.nombre = nombre;  
    }  
    public void sonido() {  
        System.out.println("El animal hace un sonido");  
    }  
}  
  
class Perro extends Animal {  
    @Override
```




P00 Herencia

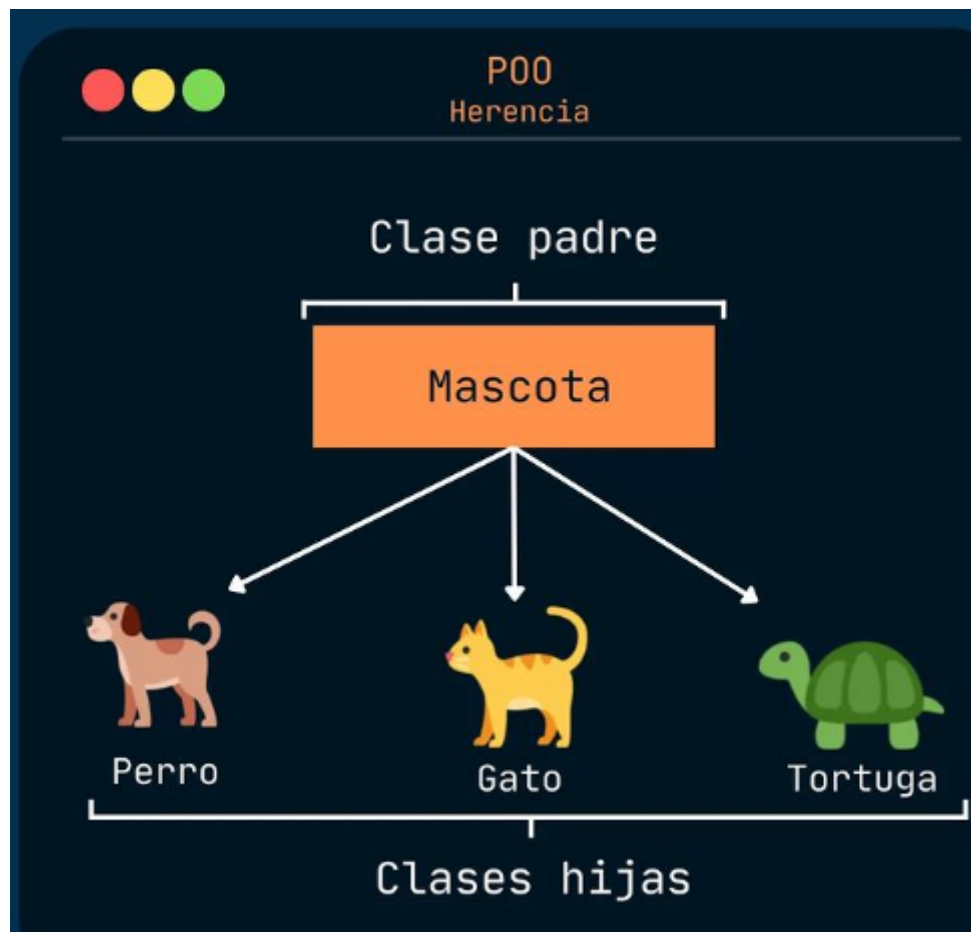


```
public void sonido() {  
    System.out.println("El perro ladra");  
}  
}
```

Aquí, `Perro` es una subclase de `Animal` y hereda el atributo `nombre` y su método `sonido()`, que luego sobrescribimos (`@override`) para proporcionar un comportamiento específico.

5 Polimorfismo

El **polimorfismo** permite que objetos de diferentes clases respondan al mismo método de formas distintas, aumentando la flexibilidad del código.



5.0.1 Ejemplo de Polimorfismo

```
class Animal {
    public void sonido() {
        System.out.println("El animal hace un sonido");
    }
}

class Perro extends Animal {
    public void sonido() {
        System.out.println("El perro ladra");
    }
}

class Gato extends Animal {
    public void sonido() {
        System.out.println("El gato maúlla");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Perro();
        miAnimal.sonido();
    }
}
```

En este ejemplo, el método `sonido()` se comporta de manera distinta dependiendo del tipo de objeto (Perro o Gato) que lo invoque.

[TEST](#)

Part V

Unidad 5: Modelamiento de clases y objetos

6 Modelamiento de clases y objetos.

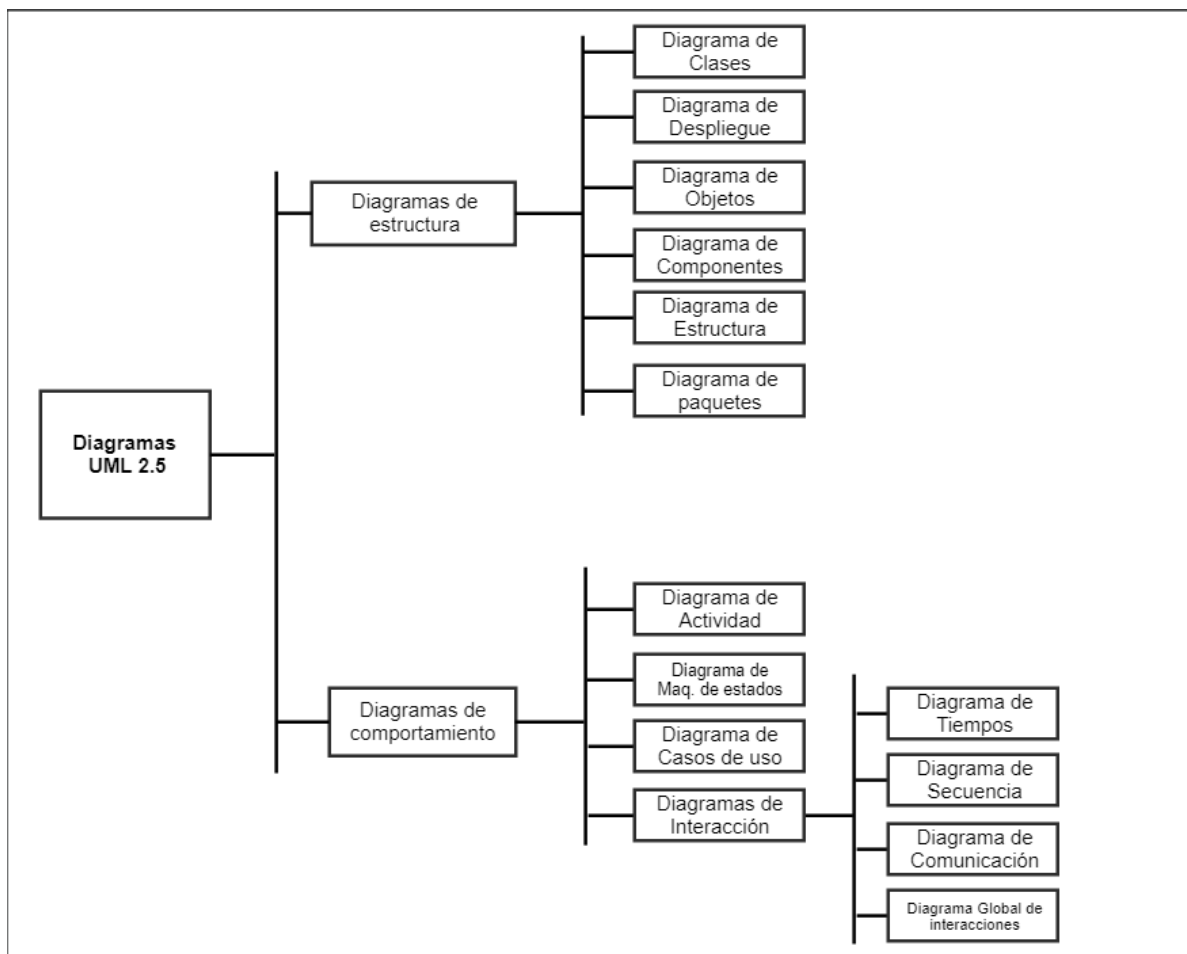
El modelamiento de clases y objetos es un tema fundamental en la programación orientada a objetos, un paradigma de programación que se basa en la idea de que los programas pueden estar formados por entidades que representan conceptos o cosas del mundo real. Estas entidades se llaman objetos, y tienen características (atributos) y comportamientos (métodos) que los definen y los diferencian de otros objetos. Los objetos se agrupan en clases, que son modelos o plantillas que definen las propiedades y las operaciones comunes de un conjunto de objetos. Por ejemplo, una clase puede ser Persona, y un objeto puede ser Juan, que es una instancia de la clase Persona.

El modelamiento de clases y objetos consiste en diseñar y representar gráficamente las clases y los objetos que forman parte de un sistema de software, así como sus relaciones y sus interacciones. Para ello, se utiliza un lenguaje estándar llamado UML (Lenguaje Unificado de Modelado), que tiene una notación gráfica y textual para describir los diferentes aspectos de un sistema. UML tiene varios tipos de diagramas, pero el más importante para el modelamiento de clases y objetos es el diagrama de clases, que muestra las clases, sus atributos, sus métodos y sus relaciones con otras clases.

Otros diagramas que se puede realizar es aquellos que muestran diferentes aspectos de un sistema. Por ejemplo, un diagrama de clases muestra las clases de un sistema y sus atributos y operaciones, un diagrama de secuencia muestra cómo se comunican los objetos entre sí a lo largo del tiempo, un diagrama de casos de uso muestra los requisitos funcionales de un sistema y los actores involucrados, etc.

El modelamiento de clases y objetos tiene muchos beneficios para el desarrollo de software, como facilitar la comprensión del problema, mejorar la comunicación entre los desarrolladores y los clientes, aumentar la reutilización y la modularidad del código, y reducir los errores y los costos de mantenimiento. El modelamiento de clases y objetos es una habilidad esencial para cualquier programador que quiera crear sistemas de software de calidad, eficientes y adaptables. En este tema, aprenderás los conceptos básicos del modelamiento de clases y objetos, cómo usar UML para representarlos, y cómo implementarlos en un lenguaje de programación como Java.

Los tipos de diagramas UML se pueden observar en la siguiente imagen.



6.1 TIPOS DE RELACIONES | DIAGRAMAS DE CASOS DE USOS.

Los **diagramas de casos de uso** muestran los casos de uso, los actores y las relaciones entre ellos.

Es, con total seguridad, el diagrama más conocido y es utilizado para representar los actores externos que interactúan con el sistema de información y a través de que funcionalidades (casos de uso o requisitos funcionales) se relacionan. Dicho de otra manera, muestra de manera visual las distintas funciones que puede realizar un usuario (más bien un tipo de usuario) de un Sistema de Información.

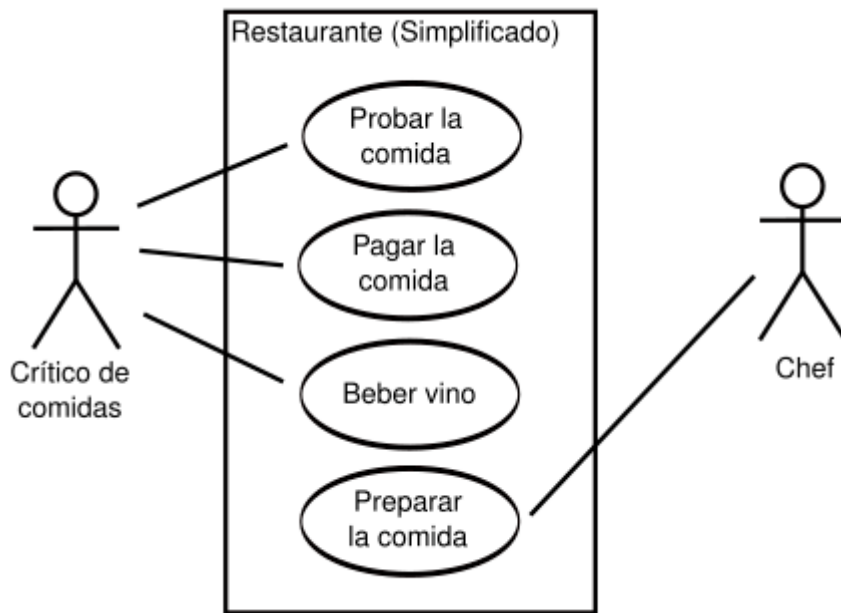
Lo primero es saber cual es su finalidad. El diagrama de casos de uso, dependiendo de la profundidad que le demos, puede ser utilizado para muchos fines, entre ellos podemos encontrar los siguientes:

- **Representar los requisitos funcionales.**
- **Representar los actores** que se comunican con el sistema. Normalmente los actores del sistema son los usuarios y otros sistemas externos que se relacionan con el sistema. En el caso de los usuarios hay que entender el actor como un “perfil”, pudiendo existir varios usuarios que actúan como el mismo actor.
- **Representar las relaciones** entre requisitos funcionales y actores.
- **Guiar el desarrollo** del sistema. Crear un punto de partida sobre el que empezar a desarrollar el sistema.
- **Comunicarse de forma precisa entre cliente y desarrollador.** Simplifica la forma en que todos los participantes del desarrollo, incluyendo el cliente, perciben como el sistema funcionará y ofrecerá una visión general común del mismo.

6.2 Elementos de un diagrama de casos de uso

Un diagrama de casos de uso está compuesto, principalmente, de 3 elementos: **Actores, Casos de uso y Relaciones.**

La interacción entre actores no se ve en el **diagrama de casos de uso**. Si esta interacción es esencial para una descripción coherente del comportamiento deseado, quizás los límites del sistema o del caso de uso deban de ser re-examinados. Alternativamente, la interacción entre actores puede ser parte de suposiciones usadas en el caso de uso. Sin embargo, los actores son una especie de rol, un usuario humano u otra entidad externa puede jugar varios papeles o roles. Así el Chef y el Cajero podrían ser realmente la misma persona.



6.2.1 Actores

Como ya hemos comentado en la presentación, un actor es algo o alguien externo al sistema que interactúa de forma directa con el sistema. Cuando decimos que interactúa nos referimos a que aporta información, recibe información, inicia una acción...

Se representan con una imagen de un “muñeco de palo” con el nombre del actor debajo

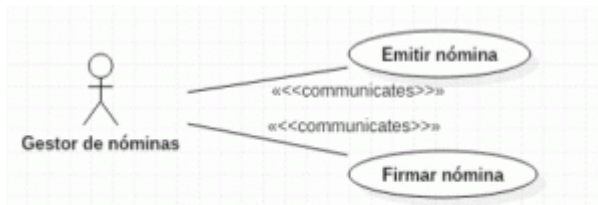


Representación de un actor

Existen dos tipos de actores: Los usuarios y los sistemas.

No hay que entender los usuarios como personas singulares, sino como “perfiles o roles” que identifican a un tipo de usuario, pero no al usuario en sí. Por ejemplo, en una aplicación de gestión de nóminas, un actor de este tipo podría ser “gestor de nóminas” que se encarga de emitir y firmar nóminas. Este rol podría ser tomado, por ejemplo, por cualquier individuo del

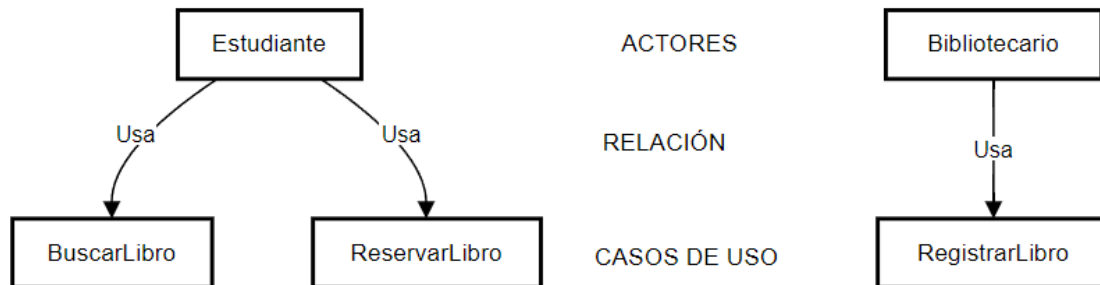
personal de recursos humanos y, además, por el jefe de la empresa. Es un ejemplo muy sencillo, pero como puedes ver, un actor no representa a una única persona o a un único usuario.



Ejemplo de actor

Por otro lado, los actores pueden ser otros sistemas que también interactúan con nuestro propio sistema. Un ejemplo podría ser, en nuestra aplicación de nóminas, un sistema que almacene las nóminas firmadas a modo de archivo. En este caso cuando se firma la nómina se recibe la misma por el sistema de archivo, por tanto el caso de uso se relaciona con el actor.

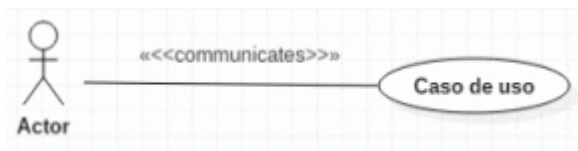
En ocasiones este tipo de actores no se representa con un “hombre de palo” porque puede dar la sensación de que es un usuario y queda poco intuitivo.



6.3 Relaciones

Las relaciones **conectan los casos de uso** con los actores o los casos de uso entre sí.

Cuando conectan un actor con un caso de uso representa que ese actor **interactúa** de alguna manera con ese caso de uso y se representa con una línea continua con la identificación «*communicates*».



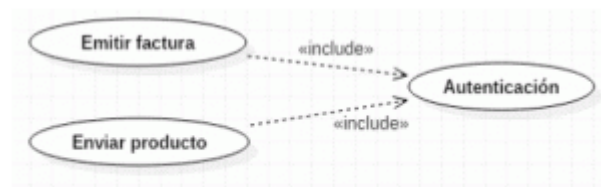
Cuando conectan casos de uso entre sí se pueden diferenciar dos tipos de relaciones: «**include**» y «**extends**». En español a veces se usa la nomenclatura «usa» y «extiende»:

- «**include**»: Se utiliza para representar que un caso de uso **utiliza siempre** a otro caso de uso. Es decir, un caso de uso se ejecutará obligatoriamente (lo incluye, lo usa). Se representa con una flecha discontinua que va desde el caso de uso de origen al caso de uso que se incluye.



Relación include entre dos casos de uso

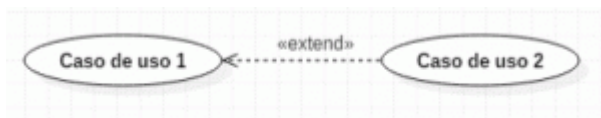
Un uso típico de este tipo de relaciones se produce cuando dos casos de uso **comparten una funcionalidad**. Esa funcionalidad es extraída de los dos y se crea un caso de uso nuevo que se relaciona con los anteriores con un include.



Ejemplo de uso de include

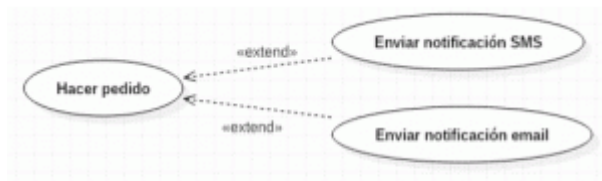
En este ejemplo, los casos de uso emitir factura y enviar producto ejecutarán ambos el caso de uso autenticación.

- «**extend**»: Este tipo de relaciones se utilizan cuando un caso de uso tiene un comportamiento **opcional**, reflejado en otro caso de uso. Es decir, un caso de uso puede ejecutar, normalmente dependiendo de alguna condición o flujo del programa, otro caso de uso. Se representa con una flecha discontinua que va desde el caso de uso opcional al original.



Relación extend entre dos casos de uso

Un ejemplo de esta relación podría ser la siguiente:

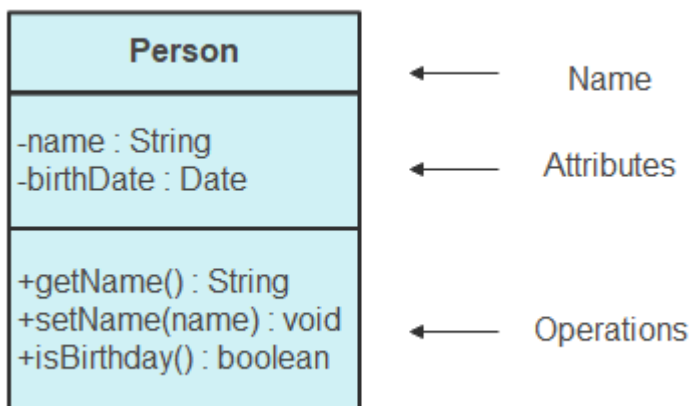


Ejemplo de relaciones extend

En este supuesto el caso de uso Hacer pedido puede dar lugar (o no) a otros dos casos de uso: Enviar notificación SMS y Enviar notificación email. Se supone que, cuando un usuario hace un pedido, el sistema le permite elegir si quiere que se envíe una notificación de ese pedido por SMS o por email

6.4 Tipos de relaciones.

Para una mejor explicación del tipo de relaciones hagamos un ejemplo teniendo presente el siguiente diagrama.



En el grafico anterior podemos observar que consta de lo siguiente: nombre | atributos | métodos.

Con esta información realicemos ejemplos para un mejor entendimiento.

Si tenemos una clase denominada Animales, cuyos atributos son nombre, ID, edad y sus métodos son definirNombre y comer.

Para esto tener presente la información de la imagen siguiente.

Símbolo	Mnemónico	Resultado
-	Private	Atributos privados solo pueden ser vistos y utilizados por métodos de la misma clase. Por omisión los atributos son privados.
+	Public	Atributos públicos pueden ser vistos y utilizados por métodos de la misma clase y de otras clases.
#	Protected*	Atributos protegidos pueden ser vistos y utilizados por métodos de la misma clase y por métodos de clases derivadas o subclases.
—	Static*	Atributos estáticos son únicos para toda la clase, es decir, no pertenecen a ningún ejemplar u objeto de la clase, pero pueden ser vistos por todos los objetos de la clase.

Es decir para representar atributos privados sería de la siguiente manera:

-nombre | Atributo privado

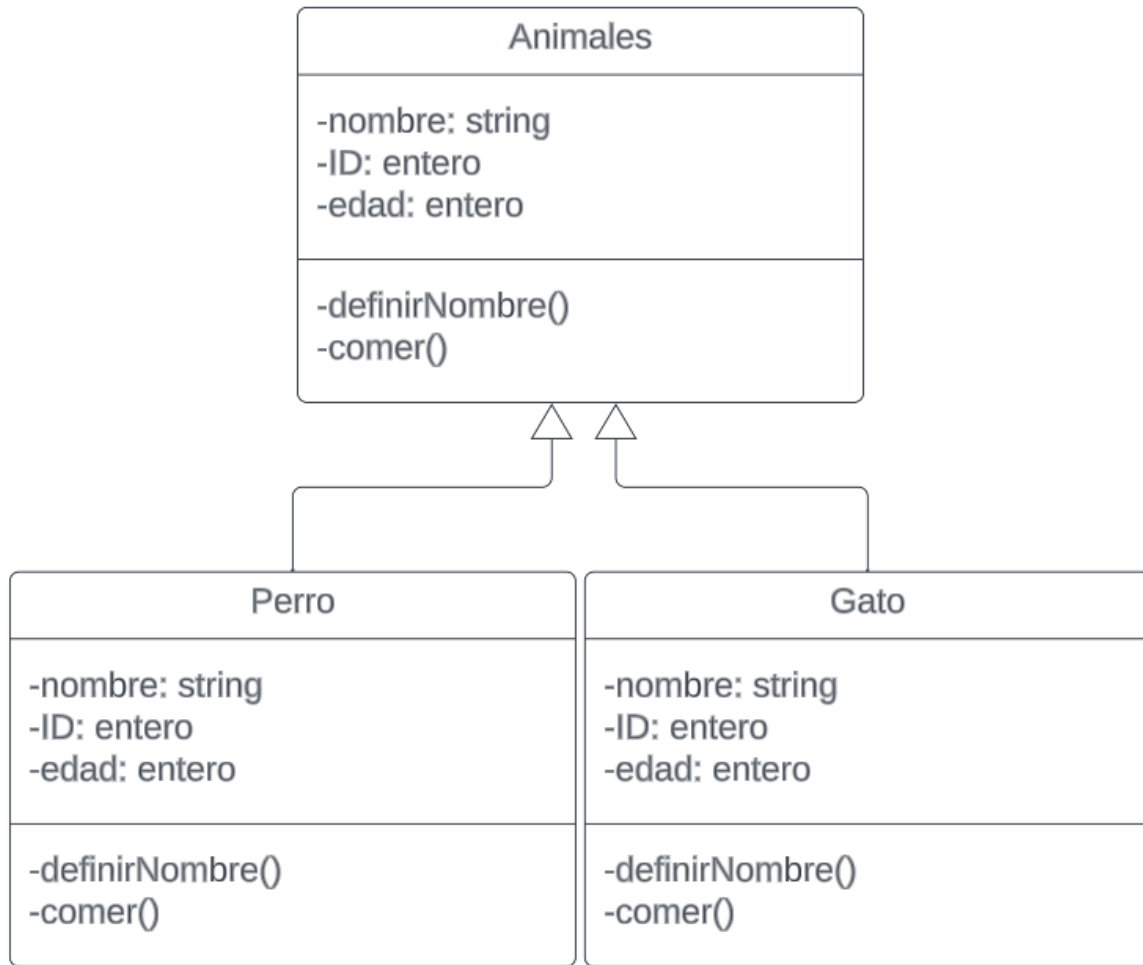
+ID | Atributo público

#edad | Atributo protegido.

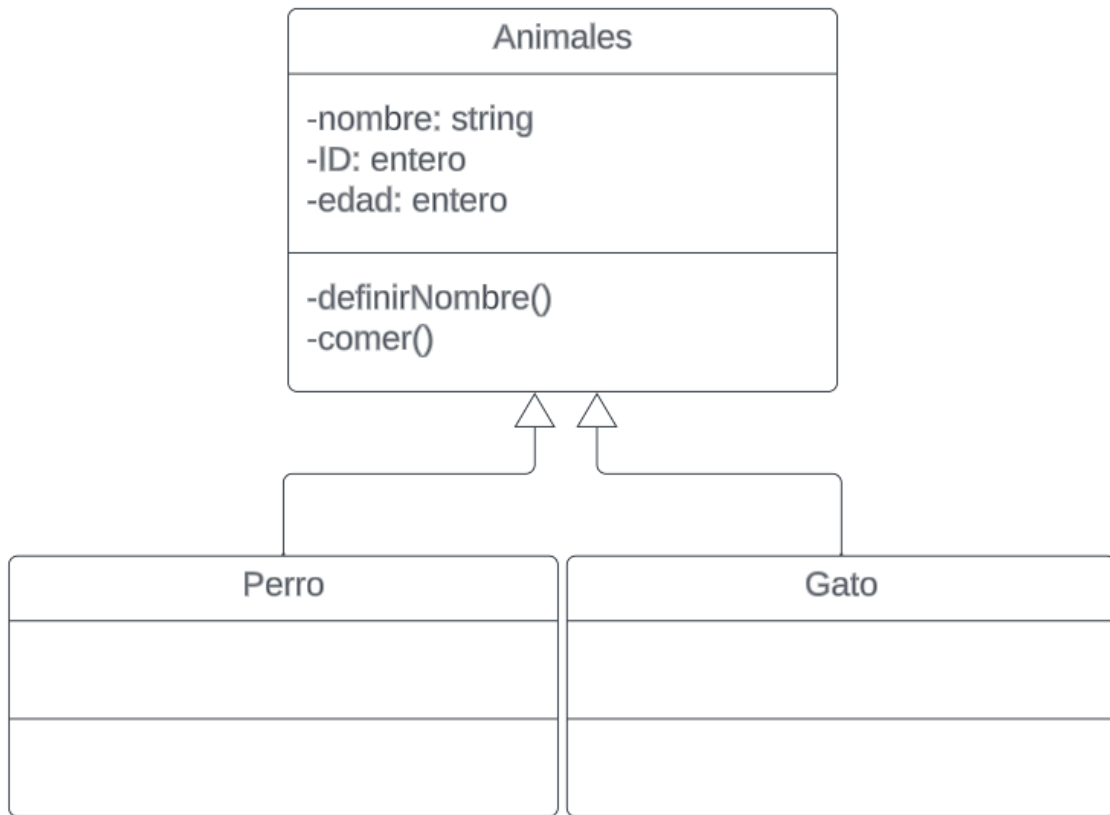
Teniendo la representación de la clase como se observa en la siguiente imagen.

Animales
-nombre: string -ID: entero -edad: entero
-definirNombre() -comer()

Que sucedería si se quisiera crear 2 animales diferentes un perro y un gato.



Tendríamos el siguiente diagrama, sin embargo, aplicando el principio de herencia, podemos re utilizar el código, como se observa en la imagen siguiente.

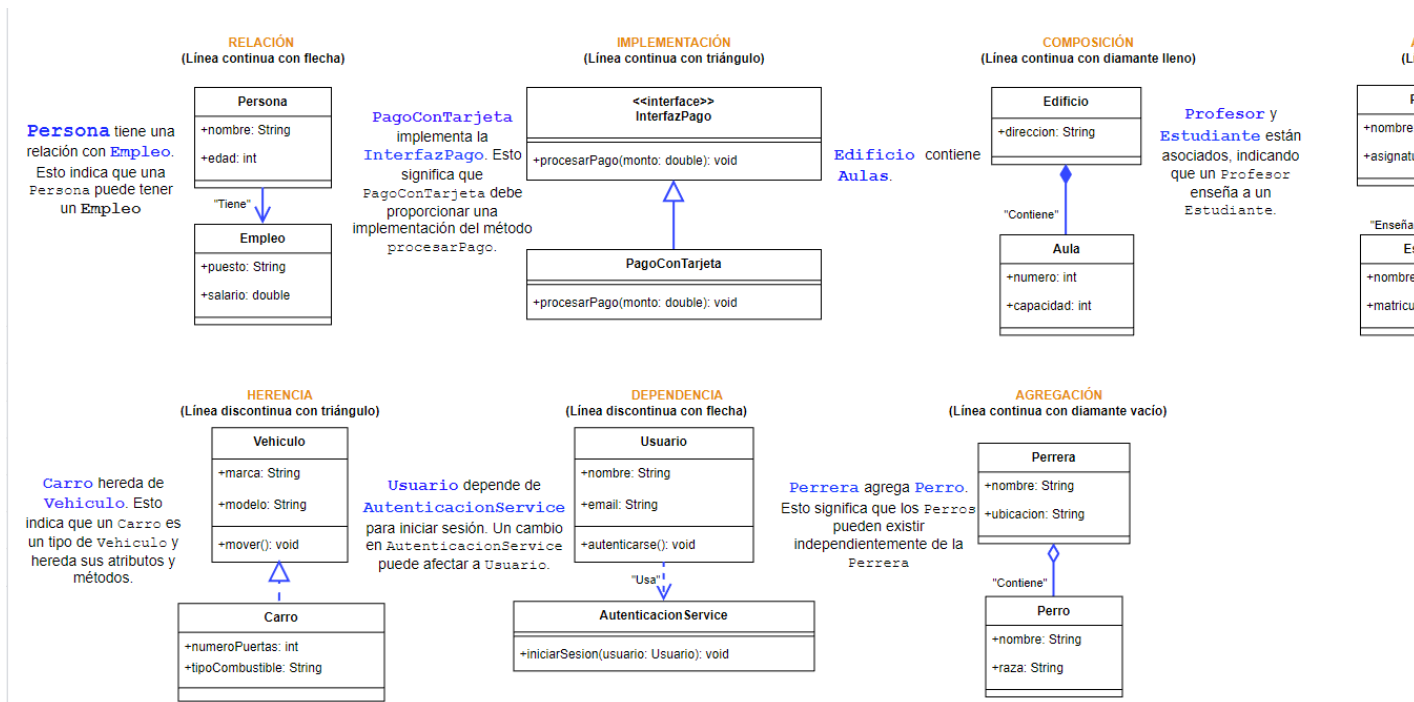


La clase animales en la imagen anterior toma el nombre de clase padre o superclase y las clases de perro y gato toman el nombre de subclase o clase hija.

6.5 Tipos de relaciones en diagramas de clases UML:

6.5.1 Relación (Línea continua con flecha)

- **Ejemplo:** *Persona* tiene una relación con *Empleo*.
- **Descripción:** Esta relación indica que una clase (*Persona*) está asociada a otra clase (*Empleo*). En este caso, una persona puede tener un empleo, lo que representa una asociación básica entre estas dos entidades.
- **Símbolo:** Línea continua con flecha.



6.5.2 Implementación (Línea continua con triángulo)

- **Ejemplo:** **PagoConTarjeta** implementa la **InterfazPago**.
- **Descripción:** En este caso, **PagoConTarjeta** debe proporcionar una implementación del método **procesarPago()** definido en la interfaz **InterfazPago**. Esto significa que **PagoConTarjeta** se compromete a cumplir con el contrato especificado por la interfaz.
- **Símbolo:** Línea continua con triángulo en la dirección de la interfaz.

6.5.3 Composición (Línea continua con diamante lleno)

- **Ejemplo:** **Edificio** contiene **Aulas**.
- **Descripción:** La composición representa una relación fuerte de “parte de”. En este caso, las aulas forman parte del edificio, y si el edificio se destruye, las aulas también desaparecen. **Aula** no puede existir sin **Edificio**.
- **Símbolo:** Línea continua con diamante lleno en el lado del todo (**Edificio**).

6.5.4 Asociación (Línea continua)

- **Ejemplo:** Profesor y Estudiante están asociados.
- **Descripción:** Indica una relación entre dos clases en la que ambas pueden interactuar entre sí. En este caso, un Profesor enseña a un Estudiante. Es una relación básica, sin dependencia fuerte ni implicaciones de pertenencia.
- **Símbolo:** Línea continua sin ningún otro símbolo.

6.5.5 Herencia (Línea discontinua con triángulo)

- **Ejemplo:** Carro hereda de Vehiculo.
- **Descripción:** La herencia indica una relación “es un tipo de”. Carro hereda de Vehiculo, lo que significa que Carro es una especialización de Vehiculo y hereda sus atributos y métodos. Este es un caso de polimorfismo donde Carro puede ser tratado como Vehiculo.
- **Símbolo:** Línea discontinua con triángulo apuntando hacia la superclase (Vehiculo).

6.5.6 Dependencia (Línea discontinua con flecha)

- **Ejemplo:** Usuario depende de AutenticacionService.
- **Descripción:** La dependencia indica que una clase (Usuario) utiliza otra clase (AutenticacionService) temporalmente para realizar una acción. En este ejemplo, Usuario depende de AutenticacionService para iniciar sesión. La relación es débil, lo que significa que Usuario solo necesita AutenticacionService temporalmente.
- **Símbolo:** Línea discontinua con flecha apuntando hacia la clase de la que depende (AutenticacionService).

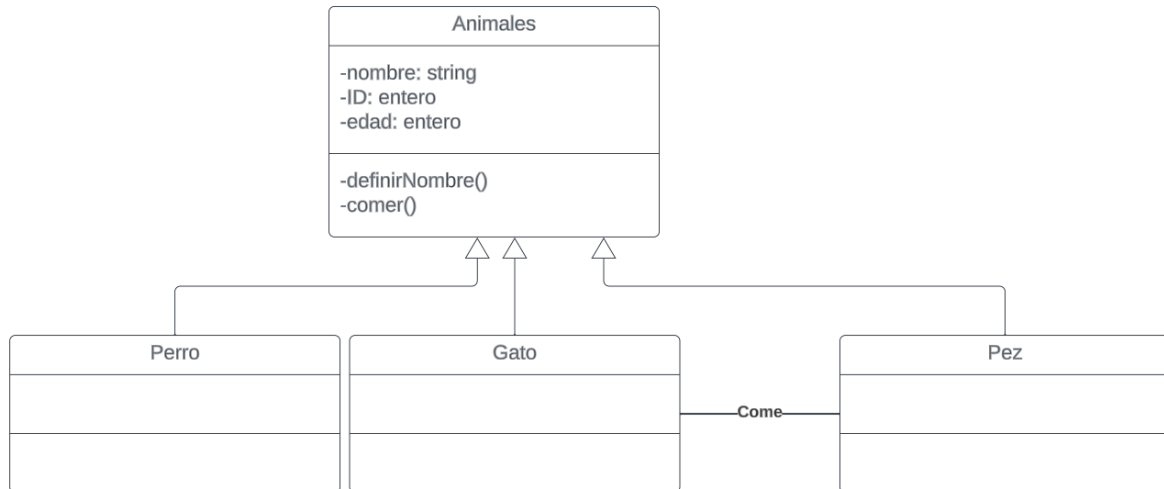
6.5.7 Agregación (Línea continua con diamante vacío)

- **Ejemplo:** Perrera agrega Perro.
- **Descripción:** La agregación representa una relación de “parte de” débil. Perro es una parte de Perrera, pero puede existir independientemente de ella. Es decir, si la Perrera deja de existir, los Perros pueden seguir existiendo.
- **Símbolo:** Línea continua con un diamante vacío en el lado del todo (Perrera).

6.6 Asociación:

Utilizando una línea simple la asociación indica una relación entre dos clases sin dependencia.

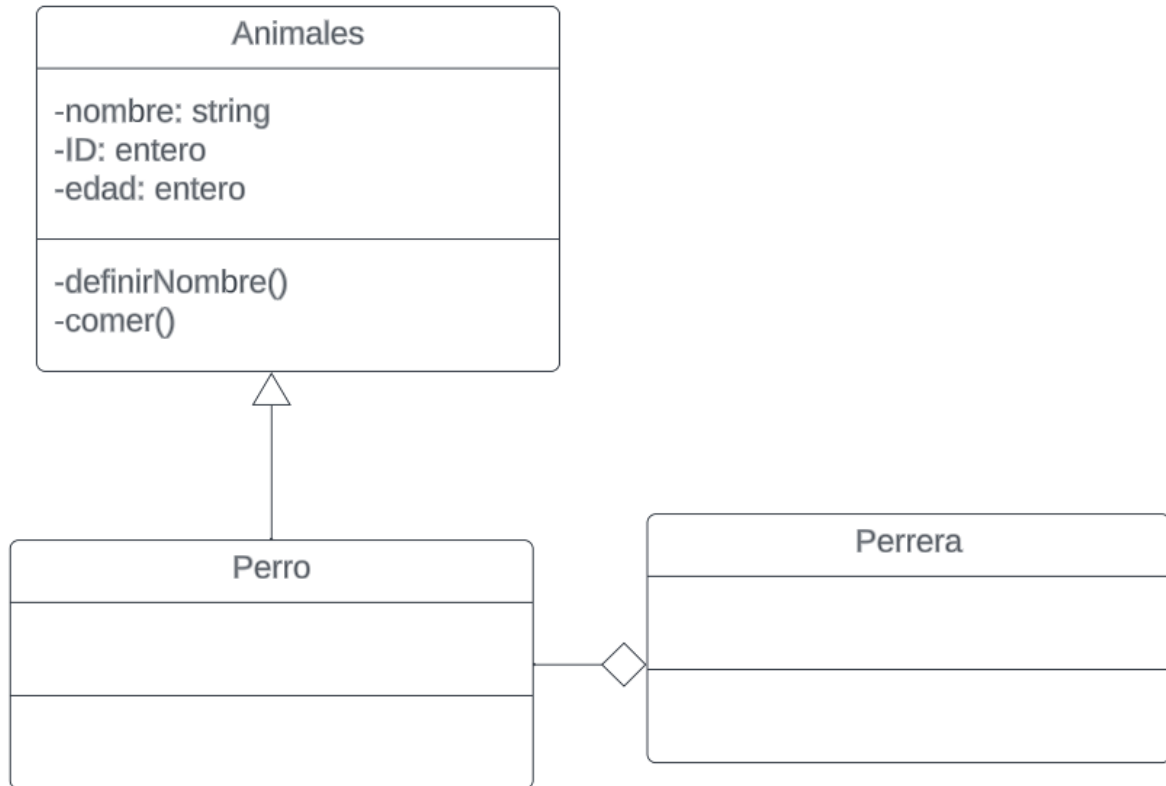
Ejemplo si ahora creamos un animal (PEZ), podemos decir que los gatos pueden comer peces utilizando el diagrama que se observa en la siguiente imagen.



6.7 Agregación:

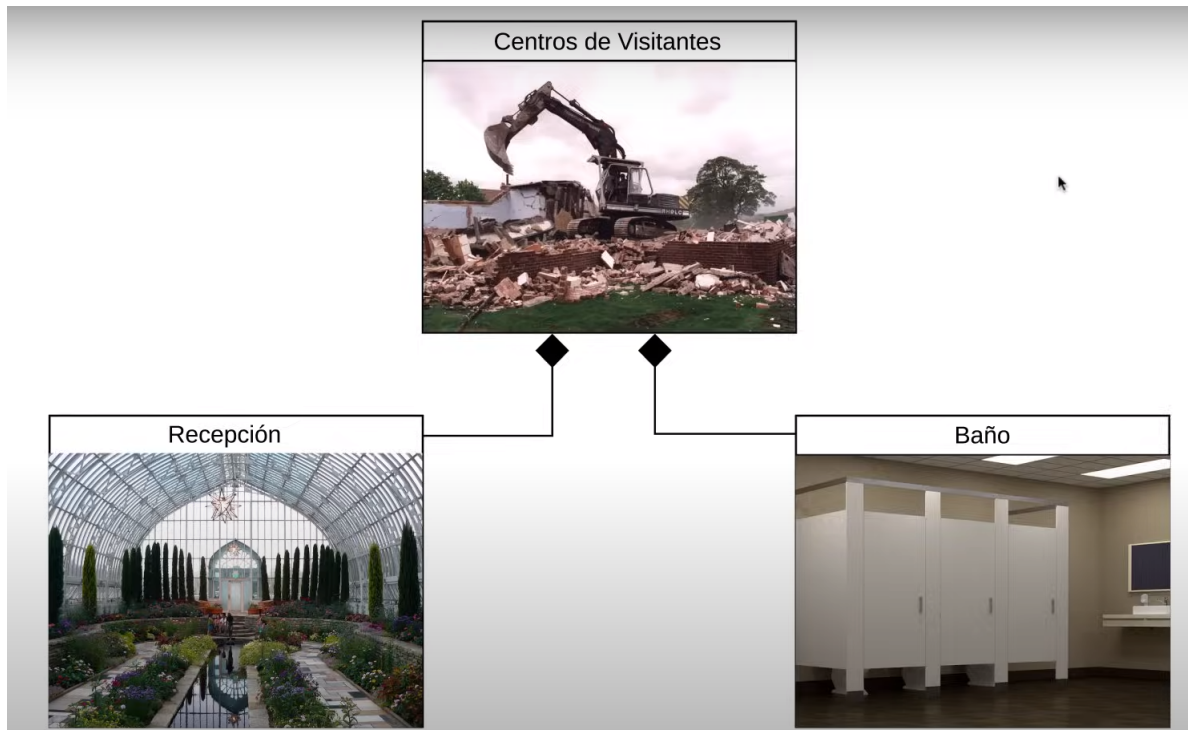
Es un tipo especial de asociación que especifica un todo y sus partes, esta relación indica que los elementos pueden existir fuera del todo. Ejemplo:

Si creamos una clase **Perro**, y una clase **Perrera**, el perro podría pertenecer a la clase **Perrera**, pero no depende necesariamente la clase **perro** de la **perrera** y se representa como se observa en la siguiente imagen.



6.8 Composición:

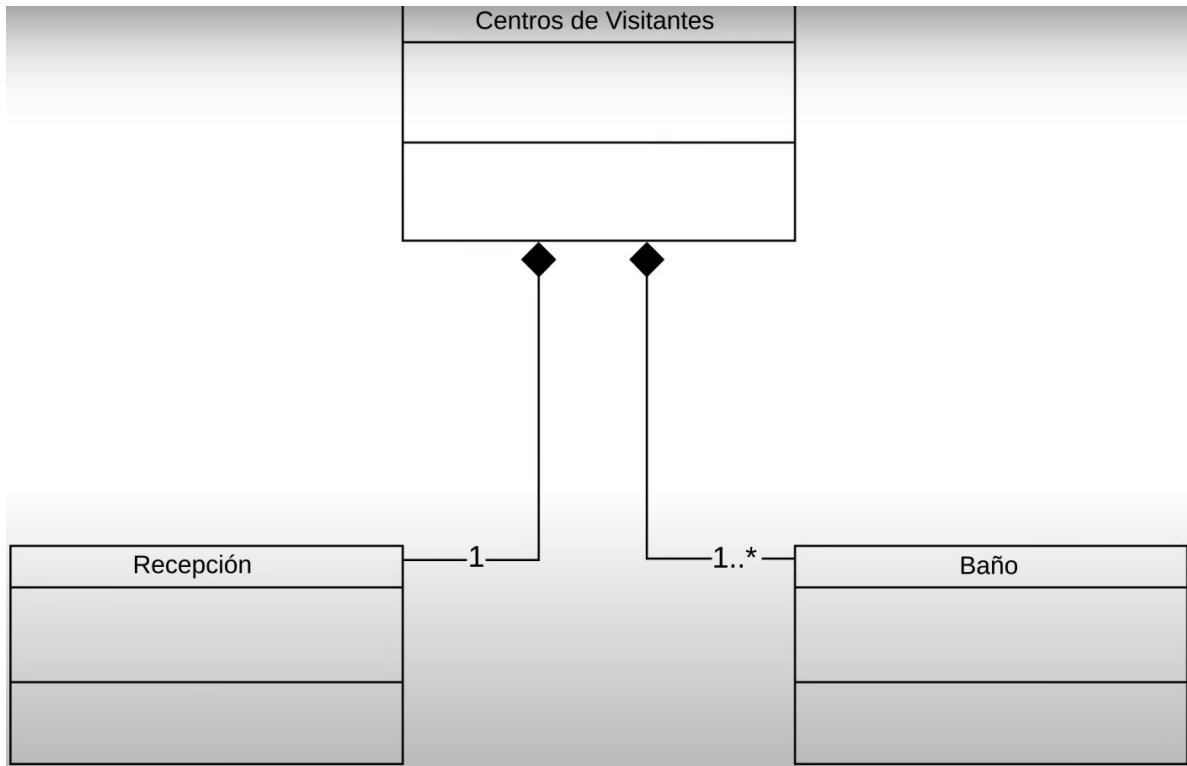
Es la relación en la que la parte no puede existir fuera del todo sin la otra clase. Ejemplo, si tenemos un centro de visitantes en la cual tenga varios espacios físicos como recepción y baños, Si el centro de visitantes se derrumbara sus espacios físicos como recepción y baño no podrían existir separados.



6.9 Multiplicidad.

Permite restringir las restricciones numéricas de dependencia.

Ejemplo si queremos especificar que el centro de visitantes tiene un solo centro de recepción, y que puede tener uno o varios baños, se puede realizar la siguiente manera.



Si tienes más dudas, te sugiero revisar el siguiente video:

6.10 PlantiumUML herramienta

```

@startuml
class Persona {
    + nombre : String
    + correr() : void
}
@enduml
  
```

```

@startuml
class Persona {
    + nombre : String
    + correr() : void
}

class Estudiante extends Persona {
  
```

```
+ matricula : String  
+ estudiar() : void  
}  
@enduml
```

6.11 PlantiumUML Video

6.12 Video

6.13 TEST

Part VI

Unidad 6: Código Limpio

7 Nombres Claros y Descriptivos

Introducción a Código Limpio

Escribir **código limpio** significa crear un código que sea fácil de leer, entender y mantener. Según Robert C. Martin, también conocido como “Uncle Bob”, un código limpio es aquel que “parece que fue escrito por alguien que se preocupa”. A continuación, se presentan los principios clave de código limpio que todo desarrollador debe seguir para mejorar la calidad de sus programas.

- Variables

Los nombres de las variables deben ser descriptivos y reflejar claramente su propósito. Esto mejora la legibilidad y hace que el código sea más fácil de entender.

- Ejemplo de un mal nombre:**

```
int n; // ¿Qué representa "n"?
```

- Ejemplo de un buen nombre:

```
int numeroDeEstudiantes;
```

7.1 Funciones y Métodos Los nombres de funciones y métodos deben describir lo que hacen. Es una buena práctica que los nombres sean verbos o frases verbales.

- Ejemplo de un mal nombre:

```
void d(); // ¿Qué significa "d"?
```

- Ejemplo de un buen nombre:


```
void dibujarRectangulo();
```

7.2 Funciones Simples y de Responsabilidad Única

Las funciones deben ser cortas y realizar solo una tarea específica. Una función que intenta hacer muchas cosas es difícil de entender y mantener.

- Ejemplo En lugar de hacer todo en una sola función, divide las tareas en funciones separadas.
- Ejemplo de función compleja:

```
void procesarDatosYGuardarEnBD() { // Procesa datos // Guarda en la base de datos  
}
```

Ejemplo de funciones separadas:

```
void procesarDatos() { // Procesa datos  
}  
void guardarEnBD() { // Guarda en la base de datos  
}
```

7.3 Evitar Comentarios Innecesarios

Un código limpio debería ser lo suficientemente claro para no necesitar comentarios. Si sientes que necesitas comentar para explicar el código, es probable que el código en sí necesite ser refactorizado.

- Ejemplo

Código con comentarios innecesarios:

```
int edad = 25; // Declara la edad del usuario
```

Código claro sin comentarios:

```
int edadUsuario = 25;
```

7.4 Evitar Código Duplicado

El código duplicado genera inconsistencias y es difícil de mantener. Si necesitas hacer algo similar en múltiples lugares, utiliza funciones o métodos para reducir la duplicación.

Ejemplo de código duplicado:

```
int calcularAreaRectangulo (int ancho, int alto) {  
    return ancho * alto;  
}  
  
int calcularAreaCuadrado(int lado) {  
    return lado * lado;  
}
```

Ejemplo de código refactorizado:

```
int calcularArea(int ancho, int alto) {  
    return ancho * alto;  
}
```

7.5 Control de Excepciones y Errores

El manejo de excepciones debe ser claro y específico. No uses excepciones generales como `catch (Exception e)` y evita el abuso de excepciones para el control de flujo.

- Ejemplo

Mal manejo de excepciones:

```
try {  
    // Código  
} catch (Exception e) {  
    System.out.println("Ocurrió un error");  
}
```

Buen manejo de excepciones:

```

try {

    // Código

} catch (IOException e) {

    System.out.println("Error de entrada/salida");

} catch (NullPointerException e) {

    System.out.println("Error: valor nulo encontrado");

}

```

7.6 Uso de Formato Consistente

El código debe tener un formato consistente para mejorar su legibilidad. Esto incluye el uso adecuado de sangría, espacios y estilo de llaves. Además, el equipo de desarrollo debe seguir una convención de formato común.

7.6.1 Ejemplo

Mal formato:

```

if(condicion){
    doSomething();
}

```

Buen formato:

```

if (condicion) {
    doSomething();
}

```

7.7 Reducir el Número de Parámetros

Las funciones con muchos parámetros son difíciles de entender y usar correctamente. Intenta reducir el número de parámetros pasando objetos cuando sea posible.

Ejemplo de demasiados parámetros:

```
void crearUsuario(String nombre, String apellido, int edad, String direccion) {  
    // Crear usuario  
}
```

Ejemplo de uso de un objeto:

```
// Main.java  
  
// Definición de la clase Usuario  
class Usuario {  
    String nombre;  
    String apellido;  
    int edad;  
    String direccion;  
  
    // Constructor para inicializar los atributos de Usuario  
    public Usuario(String nombre, String apellido, int edad, String direccion) {  
        this.nombre = nombre;  
        this.apellido = apellido;  
        this.edad = edad;  
        this.direccion = direccion;  
    }  
  
    // Método para mostrar la información del usuario  
    public void mostrarInformacion() {  
        System.out.println("Nombre: " + nombre);  
        System.out.println("Apellido: " + apellido);  
        System.out.println("Edad: " + edad);  
        System.out.println("Dirección: " + direccion);  
    }  
}  
  
// Clase principal con el método crearUsuario  
public class Main {
```

```

// Método para "crear" un usuario (en este caso, solo muestra la información)
public static void crearUsuario(Usuario usuario) {
    System.out.println("Creando usuario...");
    usuario.mostrarInformacion();
}

public static void main(String[] args) {
    // Crear una instancia de Usuario
    Usuario usuario = new Usuario("Luis", "Jaramillo", 30, "Calle Falsa 123");

    // Llamar al método crearUsuario
    crearUsuario(usuario);
}
}

```

7.8 Pruebas Unitarias

Es importante escribir pruebas unitarias para el código. Las pruebas ayudan a detectar errores y aseguran que el código funcione correctamente a medida que evoluciona.

- Ejemplo de prueba unitaria en Java

```

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculadoraTest {

    @Test
    public void testSuma() {
        Calculadora calc = new Calculadora();
        assertEquals(5, calc.suma(2, 3));
    }
}

```

[TEST](https://forms.gle/pRTgrvN2GhVDRkqw7)

Part VII

Unidad 7: Estructura General de un Programa

8 Estructura General de un Programa en JAVA

En el mundo de la programación, la estructura del programa consiste en un conjunto de sentencias con su respectiva sintaxis que se aplica de manera general al momento de la creación de un programa en cualquier lenguaje de programación.

Una sentencia es una unidad ejecutable más pequeña de un programa. En otras palabras, una línea de código escrita es una sentencia. Especifica y controla el flujo y orden de ejecución del programa. Una sentencia consta de palabras clave o reservadas como *cout*, *cin*, *for*, *while*, *if ... else*, etc., expresiones, declaraciones o llamadas a funciones. Si no existen sentencias específicas de selección o salto, el programa se ejecuta de forma secuencial en el mismo orden en que se ha escrito el código fuente (es el que podríamos considerar orden “natural” de ejecución).

Las sentencias se ejecutan evaluando la expresión. Todos los efectos colaterales de la evaluación son tenidos en cuenta antes de ejecutar la próxima sentencia. La mayoría de sentencias son asignaciones o llamadas a funciones. Un caso especial es la sentencia nula; consiste en un punto y coma (;) aislado. Una sentencia nula no hace nada, pero puede ser necesaria en situaciones en que la sintaxis del lenguaje espere una sentencia pero nuestro programa no necesita hacer nada.

Pueden existir infinitas sentencias distintas e innumerables criterios para su clasificación. Las clases de sentencia son: - De expresión - Compuestas - De selección - De iteración - De salto - De declaración - Bloques de intento - Sentencias de pre proceso - Sentencias ensamblador

Para utilizar los siguientes ejemplos realiza los siguientes pasos previos.

- 1) crea una clase tipo main.
 - 2) copia lo que está dentro de las llaves de la línea “public static void main(String[] args) {”
 - 3) ejecuta el código en netbeans o vscode, o como hayas configurado.
- De expresión: Son sentencias que consisten en una expresión seguida de un punto y coma (;). La expresión puede ser una asignación, una llamada a un método, una operación aritmética, etc. Por ejemplo:

```
public class Clasevirtual {  
  
    public static void main(String[] args) {
```

```

    int x,y;
    x = 5; // Asigna el valor 5 a la variable x
    System.out.println("Hola"); // Llama al método println para mostrar un mensaje
    y = x + 3; // Suma el valor de x y 3 y lo guarda en la variable y
    System.out.println(y);
}
}

```

- Compuestas: Son sentencias que agrupan varias sentencias simples o compuestas entre llaves ({ y }). Se usan para formar bloques de código que se ejecutan como una unidad. Por ejemplo:

```

{ // Inicio del bloque
    int a = 10; // Sentencia simple
    int b = 20; // Sentencia simple
    if (a > b) { // Sentencia compuesta
        System.out.println("a es mayor que b"); // Sentencia simple
    }
    else { // Sentencia compuesta
        System.out.println("b es mayor que a"); // Sentencia simple
    }
} // Fin del bloque

```

- De selección: Son sentencias que permiten elegir entre diferentes opciones de ejecución según se cumpla o no una condición o expresión lógica. Hay dos tipos principales de sentencias de selección: if y switch. Por ejemplo:

```

if (x > 0) { // Si x es positivo
    System.out.println("x es positivo"); // Muestra un mensaje
}
else { // Si x no es positivo
    System.out.println("x es negativo o cero"); // Muestra otro mensaje
}

switch (y) { // Según el valor de y
    case 1: // Si y es 1
        System.out.println("y es uno"); // Muestra un mensaje
        break; // Sale del switch
    case 2: // Si y es 2
        System.out.println("y es dos"); // Muestra otro mensaje
        break; // Sale del switch
}

```



```

        default: // Si y no es ni 1 ni 2
            System.out.println("y es otro valor"); // Muestra otro mensaje
            break; // Sale del switch
    }

```

- De iteración: Son sentencias que permiten repetir una o más sentencias mientras se cumpla una condición o un número determinado de veces. Hay tres tipos principales de sentencias de iteración: while, do-while y for. Por ejemplo:

```

while (x < 10) { // Mientras x sea menor que 10
    System.out.println("x es " + x); // Muestra el valor de x
    x++; // Incrementa x en 1
}

do { // Haz
    System.out.println("y es " + y); // Muestra el valor de y
    y--; // Decrementa y en 1
} while (y > 0); // Mientras y sea mayor que 0

for (z = 0; z < 5; z++) { // Para z desde 0 hasta 4, incrementando z en 1
    System.out.println("z es " + z); // Muestra el valor de z
}

```

- De salto: Son sentencias que permiten alterar el flujo normal de ejecución de un programa, saltando a otra parte del código. Hay cuatro tipos principales de sentencias de salto: break, continue, return y throw. Por ejemplo:

```

for (i = 0; i < 10; i++) { // Para i desde 0 hasta 9
    if (i == 5) { // Si i es 5
        break; // Sale del bucle
    }
    System.out.println("");
}

```

- De declaración: Son sentencias que permiten declarar variables, constantes, métodos, clases, interfaces, etc. Se usan para definir los identificadores y sus características que se usarán en el programa. Por ejemplo:

```

int x; // Declara una variable de tipo entero llamada x
final double PI = 3.14; // Declara una constante de tipo real llamada PI con el valor
void imprimir(String s) { // Declara un método que recibe un String y no devuelve nada
    System.out.println(s); // Muestra el String
}

```

```

class Punto { // Declara una clase llamada Punto
    int x; // Con un campo de tipo entero llamado x
    int y; // Con otro campo de tipo entero llamado y
}
enum Color { // Declara una enumeración llamada Color
    ROJO, // Con un valor llamado ROJO
    VERDE, // Con otro valor llamado VERDE
    AZUL // Con otro valor llamado AZUL
}
interface Figura { // Declara una interfaz llamada Figura
    double area(); // Con un método abstracto llamado area que devuelve un double
}

```

- Bloques de intento: Son sentencias que permiten manejar las excepciones que pueden ocurrir durante la ejecución de un programa, como errores, fallos o situaciones anómalas. Se usan para evitar que el programa se cierre abruptamente y para tomar medidas correctivas o de recuperación. Hay tres tipos principales de bloques de intento: try, catch y finally. Por ejemplo:

```

try { // Intenta
    int a = 10; // Declara una variable de tipo entero con el valor 10
    int b = 0; // Declara otra variable de tipo entero con el valor 0
    int c = a / b; // Intenta dividir a entre b
    System.out.println("c es " + c); // Muestra el valor de c
}
catch (ArithmeticException e) { // Captura la excepción de tipo aritmético
    System.out.println("Error: división por cero"); // Muestra un mensaje de error
    e.printStackTrace(); // Muestra la traza de la excepción
}
finally { // Finalmente
    System.out.println("Fin del programa"); // Muestra un mensaje de fin
}

```

- Sentencias de preproceso: Son sentencias que se ejecutan antes de la compilación del programa, y que sirven para modificar el código fuente, incluir otros archivos, definir macros, establecer condiciones de compilación, etc. Java no tiene sentencias de preproceso como C o C++, pero tiene algunas directivas que se usan para importar otros paquetes o clases, definir el paquete al que pertenece una clase, o indicar que una clase o un método está obsoleto. Se reconocen por empezar con el símbolo @. Por ejemplo:

```
import java.util.*; // Importa el paquete java.util
package com.example; // Define el paquete com.example
@Deprecated // Indica que la clase está obsoleta
class MiClase {
    @Deprecated // Indica que el método está obsoleto
    void miMetodo() {
        // Código del método
    }
}
```

- Sentencias ensamblador: Son sentencias que se escriben en el lenguaje de bajo nivel del procesador, y que se usan para acceder directamente al hardware, optimizar el rendimiento, realizar operaciones específicas, etc. Java no tiene sentencias ensamblador como C o C++, pero tiene una forma de invocar código nativo escrito en otro lenguaje mediante la interfaz Java Native Interface (JNI). Se reconoce por usar la palabra reservada native. Por ejemplo:

```
class MiClase {
    // Declara un método nativo que recibe un int y devuelve un int
    public native int miMetodo(int x);
    // Carga la biblioteca que contiene el código nativo
    static {
        System.loadLibrary("miBiblioteca");
    }
}
```

8.1 ESTRUCTURA GENERAL DE UN PROGRAMA EN JAVA.

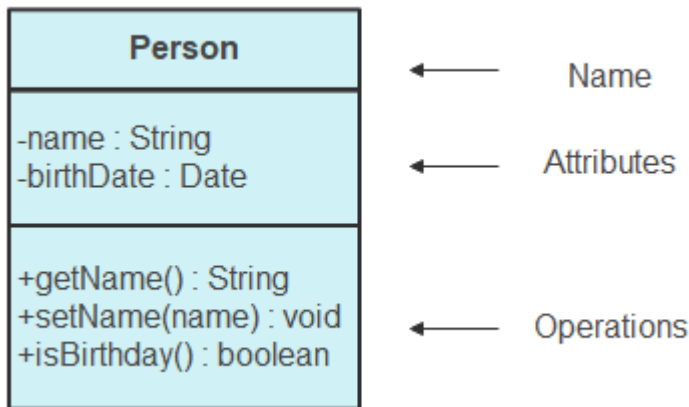
La estructura general de un programa orientado a objetos depende del lenguaje de programación que se utilice, pero en general se basa en los siguientes elementos:

- Clases: Son las plantillas o modelos que definen las propiedades y los métodos comunes de un conjunto de objetos. Las clases se pueden organizar en jerarquías mediante la herencia, que permite reutilizar y extender el código de las clases superiores a las clases inferiores.
- Objetos: Son las instancias o ejemplares de las clases, que tienen un estado (valores de los atributos) y un comportamiento (ejecución de los métodos) específicos. Los objetos se pueden comunicar entre sí mediante mensajes, que son las llamadas a los métodos de otros objetos.

- Relaciones: Son los vínculos o conexiones que existen entre las clases y los objetos, que indican cómo se relacionan o interactúan entre sí. Hay diferentes tipos de relaciones, como la asociación, la dependencia, la generalización, la realización, la agregación y la composición.

Veamos paso a paso un ejemplo de un programa orientado a objetos.

Antes de empezar, utilizaremos la imagen que utilizamos en modelamiento y se encuentra a continuación.



Para el ejemplo utilizaremos el nombre de clase “Person”, están indicados los métodos y atributos en privado para mejor entendimiento cambiaremos de privado a default, por lo que el código quedaría así.

```
public class Person{
    String name; // Atributo 1
    int birthDate; // Atributo 2

    public Person(String name, int birthDate) { //constructor
        this.name = name;
        this.birthDate = birthDate;
    }

    public void isBirthday(){ //método
        System.out.println("imprimir la fecha");
    }
}
```

Un ejemplo más completo es el siguiente:

```

// Clase Persona
public class Persona {
    // Atributos
    String nombre;
    int edad;
    String dni;

    // Constructor
    public Persona(String nombre, int edad, String dni) {
        this.nombre = nombre;
        this.edad = edad;
        this.dni = dni;
    }

    public void saludar() {
        System.out.println("Hola, me llamo " + nombre + " y tengo " + edad + " años.");
    }
}

// Clase Estudiante que hereda de Persona
public class Estudiante extends Persona {
    // Atributos
    public String carrera;
    public double notaMedia;

    // Constructor
    public Estudiante(String nombre, int edad, String dni, String carrera, double notaMedia) {
        // Llamada al constructor de la superclase
        super(nombre, edad, dni);
        this.carrera = carrera;
        this.notaMedia = notaMedia;
    }

    public void estudiar() {
        System.out.println("Estoy estudiando " + carrera + ".");
    }

    public void examinar() {
        System.out.println("Me voy a examinar de " + carrera + ".");
    }
}

```

```

// Sobrescritura del método saludar de la superclase
@Override
public void saludar() {
    System.out.println("Hola, soy " + nombre + ", estudiante de " + carrera + " y tengo un
}
}

// Clase Profesor que hereda de Persona
public class Profesor extends Persona {
    // Atributos
    String asignatura;
    double salario;

    // Constructor
    public Profesor(String nombre, int edad, String dni, String asignatura, double salario)
        // Llamada al constructor de la superclase
        super(nombre, edad, dni);
        this.asignatura = asignatura;
        this.salario = salario;
    }

    public void enseñar() {
        System.out.println("Estoy enseñando " + asignatura + ".");
    }

    public void evaluar() {
        System.out.println("Voy a evaluar a mis alumnos de " + asignatura + ".");
    }

    // Sobrescritura del método saludar de la superclase
    @Override
    public void saludar() {
        System.out.println("Hola, soy " + nombre + ", profesor de " + asignatura + " y cobro "
    }
}

// Clase principal con el método main
public class Main {
    public static void main(String[] args) {
        // Creación de objetos
        Persona p1 = new Persona("Ana", 25, "12345678A");
    }
}

```

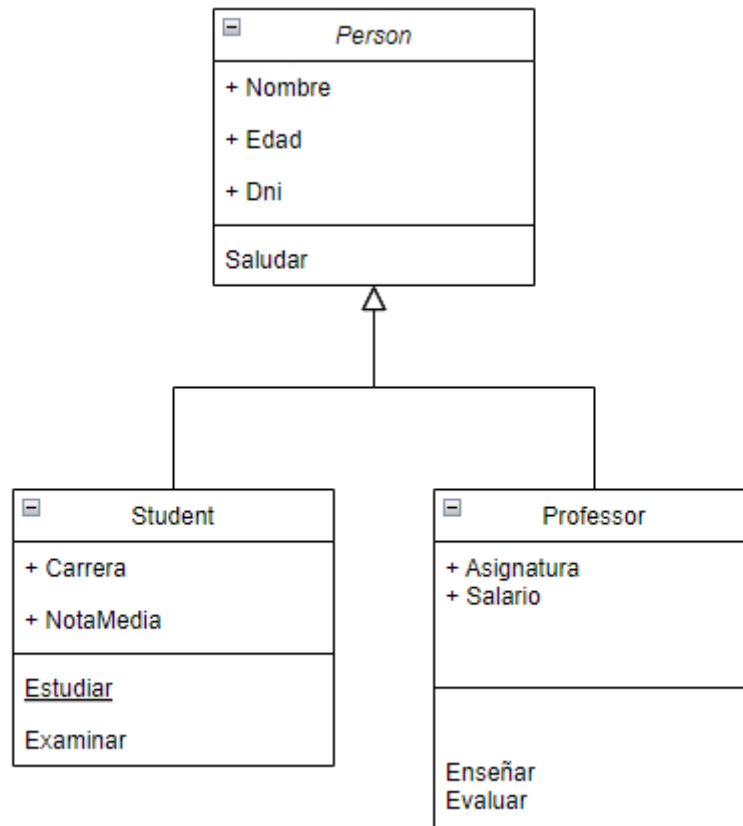
```

Estudiante e1 = new Estudiante("Luis", 20, "87654321B", "Informática", 8.5);
Profesor pr1 = new Profesor("Pedro", 35, "45678912C", "Matemáticas", 1500);

// Llamada a los métodos de los objetos
p1.saludar();
e1.saludar();
pr1.saludar();
e1.estudiar();
pr1.enseñar();
    }
}

```

Todo el código pertenece al siguiente esquema.



Este programa define tres clases: **Persona**, **Estudiante** y **Profesor**. La clase **Estudiante** y la clase **Profesor** heredan de la clase **Persona**, lo que significa que comparten sus atributos y métodos, y añaden algunos propios. El programa crea tres objetos: uno de cada clase, y llama

a sus métodos. El resultado es el siguiente:

Hola, me llamo Ana y tengo 25 años.

Hola, soy Luis, estudiante de Informática y tengo una nota media de 8.5.

Hola, soy Pedro, profesor de Matemáticas y cobro 1500 euros al mes.

Estoy estudiando Informática.

Estoy enseñando Matemáticas.

8.2 Video

Part VIII

Unidad 8: Lectura y Escritura de Datos por consola

9 Lectura y Escritura de Datos por Consola

El desarrollo de aplicaciones en Java, uno de los primeros aspectos fundamentales que todo programador debe dominar es la interacción entre el programa y el usuario a través de la consola. Esta capacidad de leer y escribir datos en la consola no solo es esencial para crear aplicaciones interactivas, sino también para facilitar la depuración y prueba de código, además de permitir el desarrollo de interfaces de usuario en programas de línea de comandos.

Este capítulo explora en profundidad los conceptos de lectura y escritura de datos por consola, desde las operaciones básicas hasta las técnicas avanzadas de captura y manipulación de datos. Iniciaremos con una explicación de los métodos más comunes en Java para obtener entradas del usuario, utilizando la clase `Scanner` de la biblioteca estándar, y abordaremos otros métodos útiles como `System.in` y `BufferedReader`. A continuación, profundizaremos en las técnicas de salida de datos utilizando `System.out`, desde el uso básico de `println` hasta el formato avanzado de mensajes con `printf`.

9.1 Introducción a la Entrada y Salida en Consola

La interacción por consola en Java permite a los desarrolladores capturar información ingresada por el usuario y devolver resultados o mensajes. En Java, la entrada y salida (E/S) de datos es fundamental para muchos programas, y este capítulo se centra en métodos prácticos para implementar E/S en aplicaciones de consola. A través de ejemplos prácticos, entenderás cómo utilizar las clases y métodos esenciales para obtener y mostrar información en el entorno de línea de comandos.

9.2 Entrada de datos

9.2.1 Clase Scanner

La clase `Scanner` es la herramienta más común para la entrada de datos en Java. Pertenecce al paquete `java.util` y permite leer diferentes tipos de datos: **cadenas**, **enteros**, **decimales** y **booleanos**. El uso de `Scanner` facilita la captura de datos con el método `next()` para cadenas o `nextInt()`, `nextDouble()`, entre otros, para tipos específicos.

- Ejemplo: Leer un número y un nombre

```
import java.util.Scanner;

public class EjemploScanner {
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Ingresa tu nombre: ");
        String nombre = entrada.nextLine();

        System.out.print("Ingresa tu edad: ");
        int edad = entrada.nextInt();

        System.out.println("Hola " + nombre + ", tienes " + edad + " años.");
    }
}
```

- `nextLine()`: captura una línea completa de texto, incluyendo espacios.
- `nextInt()`: captura un número entero.

Precaución al Leer Datos en Java

Es importante tener cuidado al alternar la lectura de datos de diferentes tipos en Java, especialmente cuando se combinan enteros o decimales (`int`, `float`, etc.) con cadenas (`String`). Después de leer un valor numérico con métodos como `nextInt()` o `nextFloat()`, el buffer de entrada puede contener un salto de línea (`\n`) residual. Esto puede causar que una llamada posterior a `nextLine()` lea dicho salto de línea en lugar de capturar la cadena esperada. Para evitar errores, se recomienda limpiar el buffer antes de leer una cadena, usando `nextLine()` adicionalmente.

```
import java.util.Scanner;

public class principal {
    public static void main(String[] args) {
        Scanner lectura = new Scanner(System.in); // 'in' significa entrada, 'out' significa salida
        // Ejemplo de lectura de datos
        System.out.print("Ingresa su nombre: ");
        String nombre = lectura.nextLine(); // Leemos un dato tipo cadena de caracteres
        System.out.print("Ingresa su edad: ");
        int edad = lectura.nextInt(); // Leemos un dato tipo entero
        System.out.print("Salario: ");
    }
}
```

```

float salario = lectura.nextFloat();
// Limpiar el buffer después de leer un entero
lectura.nextLine();
System.out.print("Ingrese su apellido: ");
String apellido = lectura.nextLine();

// ¿Qué pasará si no limpiamos el buffer?
System.out.println("¿Qué pasará si no limpiamos el buffer?");

System.out.print("Ingrese su nombre2: ");
String nombre2 = lectura.nextLine();
System.out.print("Ingrese su edad2: ");
int edad2 = lectura.nextInt();
System.out.print("Salario2: ");
float salario2 = lectura.nextFloat();
System.out.print("Ingrese su apellido: ");
String apellido2 = lectura.nextLine();
// Cerrar el scanner al finalizar
lectura.close();

// Ejemplo de escritura de datos
System.out.println("*****");
System.out.println("\nInformación ingresada1:");
System.out.println("Nombre: " + nombre);
System.out.println("Apellido: " + apellido);
System.out.println("Edad: " + edad);
System.out.println("Salario: " + salario);

System.out.println("*****");
System.out.println("\nInformación ingresada2:");
System.out.println("Nombre2: " + nombre2);
System.out.println("Apellido: " + apellido2);
System.out.println("Edad2: " + edad2);
System.out.println("Salario2: " + salario2);
}
}

```

- **Limpieza de Buffer:** Luego de leer int o float, se utiliza `lectura.nextLine()`; para limpiar el buffer antes de leer un String con `nextLine()`.
- **Evitar Errores:** Al no limpiar el buffer, el programa puede no capturar la entrada del

usuario como se espera y leer datos incorrectos o vacíos.

9.2.2 Uso de `BufferedReader`

`BufferedReader` es otra opción para la lectura de datos y es especialmente útil para aplicaciones que requieren un manejo eficiente de grandes volúmenes de datos. Utiliza `InputStreamReader` para leer desde `System.in`.

Ejemplo: Leer una cadena de texto

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class EjemploBufferedReader {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Ingresa un mensaje: ");
        String mensaje = reader.readLine();

        System.out.println("Mensaje recibido: " + mensaje);
    }
}
```

Explicación:

- `readLine()`: lee una línea de texto completa, ideal para capturar grandes bloques de datos.

9.3 Escritura de Datos en la Consola

9.3.1 Uso de `System.out.println`

`System.out.println` es el método más sencillo y común para mostrar datos en la consola. Acepta texto y datos variables, permitiendo mostrar mensajes, resultados, y más.

Ejemplo: Saludo Básico

```
public class EjemploPrintln {
    public static void main(String[] args) {
```

```

        System.out.println("Bienvenido al sistema.");
        System.out.println("Este mensaje aparecerá en una nueva línea.");
    }
}

```

Formateo de Salida con System.out.printf

El método `printf` permite controlar el formato de la salida, especialmente útil cuando se requiere una presentación organizada de datos, como tablas o valores numéricos con un número específico de decimales.

Ejemplo: Formateo con printf

```

public class EjemploPrintf {
    public static void main(String[] args) {
        String producto = "Laptop";
        double precio = 1234.56;

        System.out.printf("Producto: %s, Precio: %.2f\n", producto, precio);
    }
}

```

- `%s` es un marcador de posición para cadenas.
- `%.2f` indica un número decimal con dos dígitos después del punto.

Ejemplo 2 : Formateo con printf

```

public class Ejemplo2Printf {
    public static void main(String[] args) {
        String nombre = "Ana";
        int edad = 25;

        System.out.printf("Nombre: %s, Edad: %d años\n", nombre, edad);
    }
}

```

Salida esperada: Nombre: Ana, Edad: 25 años

Part IX

Unidad 9: Manejor de Exepciones

10 Manejo de excepciones.

10.1 Introducción

El manejo de excepciones en Java es una característica fundamental que permite gestionar errores durante la ejecución de un programa, evitando que se interrumpa abruptamente. Las excepciones representan condiciones inusuales o errores que pueden ocurrir, como intentos fallidos de abrir un archivo o la entrada de datos inválidos.

10.2 Conceptos Básicos

10.2.1 ¿Qué es una excepción?

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. En Java, todas las excepciones son objetos que heredan de la clase base `Throwable`.

Muchas clases de errores pueden provocar una excepción, desde un desbordamiento de memoria o un disco duro estropeado, un intento de dividir por cero, o intentar acceder a un vector fuera de sus límites. Cuando esto ocurre, Java **crea un objeto** de la clase excepción.

Ejemplo:

```
public class EjemploExcepciones{
    public static void main (String[] args) {
        int num1= 2, num2=0;
        int result=num1/num2;
        System.out.println("El resultado es: " + result);
    }
}
```


10.2.2 Excepciones Propias de java

Antes de continuar, se debe conocer que java **error** y **excepción** son términos relacionados con problemas que pueden ocurrir durante la ejecución de un programa, pero tienen diferencias importantes en su naturaleza y manejo:

10.2.2.1 Error:

- Un error es un problema grave que ocurre fuera del control del programa y generalmente indica un fallo crítico del sistema o del entorno de ejecución.

Características:

- Representado por la clase Error, que es parte del paquete java.lang.
- No está diseñado para ser manejado por el programa.

En la mayoría de los casos, intentar capturar o recuperar un error es inútil.

Ejemplos comunes incluyen:

- StackOverflowError: Cuando se desborda la pila debido a una recursión excesiva.
- OutOfMemoryError: Cuando la JVM no tiene suficiente memoria para continuar.

Son más relacionados con el entorno de ejecución o la configuración del sistema.

10.2.2.2 Excepción:

Una excepción es un problema que ocurre durante la ejecución del programa debido a errores de lógica o circunstancias excepcionales predecibles.

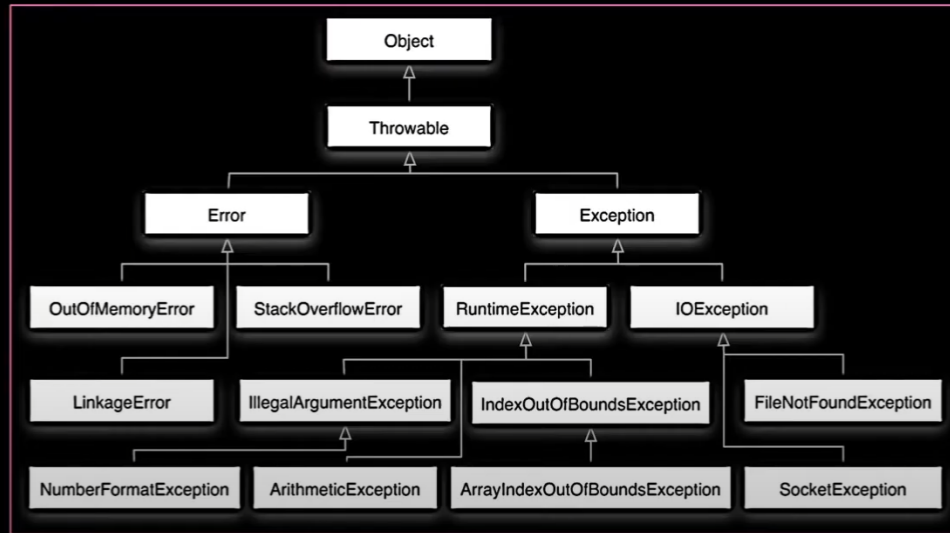
Representada por la clase Exception, que es una subclase de **Throwable**, Está diseñada para ser manejada por el programa mediante bloques try-catch o con la cláusula throws.

Se divide en dos tipos:

- Verificadas** en tiempo de compilación:
- No Verificadas** en tiempo de compilación y suelen deberse a errores de programación:

Todos vienen de la clase **throwable** como se observa en la siguiente imagen.

Excepciones Propias de Java



10.2.3 Jerarquía de Excepciones

- **Throwable:** Clase base de todas las excepciones y errores.
 - **Exception:** Errores que pueden ser manejados por el programa.
 - * **IOException:** Relacionadas con operaciones de entrada/salida.
 - * **SQLException:** Errores en bases de datos.
 - * **FileNotFoundException:** Archivo no encontrado.
 - * **ArithmeticException:** Errores matemáticos como división por cero.
 - **Error:** Condiciones críticas del sistema, como falta de memoria, que no deben manejarse directamente.

Ejemplos de Errores

- **StackOverflowError**: Este error ocurre cuando un programa utiliza demasiada memoria en la pila debido a una recursión infinita o excesiva.

```
public class StackOverflowExample {
    public static void main(String[] args) {
```

```

        recursiveMethod(); // Llamada recursiva infinita
    }

    public static void recursiveMethod() {
        recursiveMethod(); // Llamada a sí misma sin condición de salida
    }
}

```

- **OutOfMemoryError**: Este error ocurre cuando la Java virtual machine (JVM) no tiene suficiente memoria para asignar nuevos objetos en el heap.

```

import java.util.ArrayList;

public class OutOfMemoryExample {
    public static void main(String[] args) {
        ArrayList<int[]> memoryHog = new ArrayList<>();
        while (true) {
            memoryHog.add(new int[1_000_000]); // Intenta llenar el heap
        }
    }
}

```

10.2.4 Ejemplo de Excepciones

- Uso de throw

Ejemplo lanzando una excepción

```

import java.io.FileReader;

public class Throws {

    public static void main(String[] args) {

        FileReader archivo = new FileReader("archivo_inexistente.txt"); // Puede lanzar Fi
        System.out.println("Archivo abierto con éxito.");

    }

}

```

saldrá el siguiente mensaje: Exception in thread "main" java.lang.RuntimeException: Uncompilable code - unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown at exepciones.Exepciones.main(

Español: Excepción en el hilo "main" java.lang.RuntimeException: Código no compilable - excepción no reportada java.io.FileNotFoundException; debe ser capturada o declarada para lanzarse en exepciones.Exepciones.main(

Ejemplo: Si ahora sólo lanzáramos la excepción con la palabra throws obtendríamos el siguiente mensaje.

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Exepciones {

    public static void main(String[] args) throws FileNotFoundException {

        FileReader archivo = new FileReader("archivo_inexistente.txt"); // Puede lanzar Fi
        System.out.println("Archivo abierto con éxito.");
    }

}
```

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        // Intentar abrir el archivo
        FileReader archivo = new FileReader("archivo_inexistente.txt");
        BufferedReader bufferedReader = new BufferedReader(archivo);

        // Leer el archivo línea por línea
        String linea;
        while ((linea = bufferedReader.readLine()) != null) {
            System.out.println(linea); // Imprime cada línea
        }
    }
}
```

```

        System.out.println("Archivo abierto con éxito.");

        // Cerrar los recursos
        bufferedReader.close();
        archivo.close();
    }
}

```

Exception in thread "main" java.io.FileNotFoundException: archivo_inexistente.txt (El sistema no puede encontrar el archivo especificado).

10.2.5 Ejemplo de Excepciones Manejando TRY-CATCH-FINALLY

El manejo de excepciones en Java se realiza utilizando los bloques `try`, `catch`, `finally`, y `throw`. Estos permiten controlar el flujo del programa y tomar acciones específicas cuando ocurre una excepción.

- **try:** Se utiliza para envolver el código que podría generar una excepción.
- **catch:** Captura la excepción y permite tomar medidas para manejarla.
- **finally:** Es un bloque opcional que se ejecuta después del `try` y `catch`, sin importar si se lanzó una excepción o no. Se usa generalmente para liberar recursos.
- **throw:** Permite lanzar una excepción de manera manual.

Uso de `try` y `catch`

Ejemplo básico:

```

public class EjemploExcepciones {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0; // Esto genera una ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        } finally {
            System.out.println("Operación finalizada.");
        }
    }
}

```

Sin excepción

```

public class EjemploExcepciones{
    public static void main (String[] args) {
        int num1= 2, num2=0;
        int result=num1/num2;
        System.out.println("El resultado es: " + result);
    }
}

```

10.2.6 Captura de Múltiples Excepciones:

En Java, es posible capturar múltiples excepciones en un mismo bloque **try-catch**, lo que facilita el manejo de diferentes tipos de errores en una misma sección de código. Desde Java 7, también se permite combinar varias excepciones en un solo bloque **catch** usando el operador **|**.

```

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Ingrese el numerador: ");
            int numerador = scanner.nextInt();

            System.out.print("Ingrese el denominador: ");
            int denominador = scanner.nextInt();

            int resultado = numerador / denominador;
            System.out.println("Resultado: " + resultado);

        } catch (ArithmeticException e) {
            System.out.println("Error: El denominador no puede ser 0. Inténtalo de nuevo");
        } catch (java.util.InputMismatchException e) { // java.util.InputMismatchExceptio

            System.out.println("Error: Entrada inválida. Solo se permiten números enteros");
        }
    }
}

```

```
}  
  
}
```

Que pasaría ahora si se lanzara dentro de un bucle ejemplo while, para que se repita, si el usuario coloca mal el dato seguirá repitiendo la solicitud hasta tener datos correctos.

```
import java.util.Scanner;  
  
public class Main {  
  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        boolean entradaValida = false; // Bandera para controlar el bucle  
  
        while (!entradaValida) {  
            try {  
                System.out.print("Ingrese el numerador: ");  
                int numerador = scanner.nextInt();  
  
                System.out.print("Ingrese el denominador: ");  
                int denominador = scanner.nextInt();  
  
                if (denominador == 0) {  
                    throw new ArithmeticException("El denominador no puede ser 0.");  
                }  
  
                int resultado = numerador / denominador;  
                System.out.println("Resultado: " + resultado);  
  
                entradaValida = true; // Si llega aquí, las entradas son válidas  
  
            } catch (ArithmeticException e) {  
                System.out.println("Error: " + e.getMessage());  
                scanner.nextLine(); // Limpiar el buffer de entrada  
            } catch (java.util.InputMismatchException e) {  
                System.out.println("Error: Entrada inválida. Solo se permiten números enteros");  
                scanner.nextLine(); // Limpiar el buffer de entrada  
            }  
        }  
        scanner.close();  
    }  
}
```

```
}
```

Bloque Finally

El bloque **finally** se utiliza en Java para ejecutar código que debe correr sin importar si se lanzó o no una excepción. Es útil para liberar recursos, como cerrar un objeto **Scanner**.

En este caso, podemos usar **finally** para garantizar que el objeto **Scanner** se cierre después de que el programa haya terminado de procesar las entradas.

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // Crear el Scanner

        try {
            boolean entradaValida = false; // Bandera para controlar el bucle

            while (!entradaValida) {
                try {
                    System.out.print("Ingrese el numerador: ");
                    int numerador = scanner.nextInt();

                    System.out.print("Ingrese el denominador: ");
                    int denominador = scanner.nextInt();

                    if (denominador == 0) {
                        throw new ArithmeticException("El denominador no puede ser 0.");
                    }

                    int resultado = numerador / denominador;
                    System.out.println("Resultado: " + resultado);

                    entradaValida = true; // Entrada válida, salir del bucle

                } catch (ArithmeticException e) {
                    System.out.println("Error: " + e.getMessage());
                    scanner.nextLine(); // Limpiar el buffer de entrada
                } catch (java.util.InputMismatchException e) {
                    System.out.println("Error: Entrada inválida. Solo se permiten números");
                }
            }
        } finally {
            scanner.close();
        }
    }
}
```



```

        scanner.nextLine(); // Limpiar el buffer de entrada
    }
}
} finally {
    scanner.close(); // Garantiza que el recurso Scanner se cierra
    System.out.println("Recurso Scanner cerrado.");
}
}
}
}

```

10.2.7 Diferencias Clave entre los programas anteriores

Característica	Primer Código (con finally)	Segundo Código (sin finally)
Uso de finally	Utiliza el bloque finally para garantizar el cierre del recurso Scanner .	No usa finally , pero cierra el recurso manualmente al final.
Liberación de Recursos	El cierre del Scanner ocurre siempre , incluso si ocurre una excepción.	El cierre del Scanner depende de que el flujo salga del bucle correctamente.
Robustez del Código	Más robusto, ya que garantiza que el Scanner siempre se cierre.	Menos robusto, podría no cerrar el recurso si ocurre un error inesperado fuera del bucle.
Legibilidad	Separa claramente la lógica del cierre de recursos en el bloque finally .	Agrupar todo en el flujo principal, lo que puede ser más legible en casos simples.

Video de refuerzo

<https://www.youtube.com/watch?v=VHy6xFXJ1Rw>

11 Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. <https://doi.org/10.1093/comjnl/27.2.97>.