FUNDAMENTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

Andrés Marcelo Salinas Copo
Luis Tarquino Vinueza Rodríguez
Jorge Giovanni Valdiviezo Rodríguez
René Omar Villa López
David Leonardo Guevara Aulestia
Cecilia Estefanía Urquizo Alvarez
Deysi Emilene Moya Ibarra
Irma Victoria Espín Mendoza

CIDE EDITORIAL

FUNDAMENTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

FUNDAMENTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

Autores:

Andrés Marcelo Salinas Copo
Luis Tarquino Vinueza Rodríguez
Jorge Giovanni Valdiviezo Rodríguez
René Omar Villa López
David Leonardo Guevara Aulestia
Cecilia Estefania Urquizo Alvarez
Deysi Emilene Moya Ibarra
Irma Victoria Espín Mendoza

Fundamentos de Programación Orientada a Objetos (POO)

Centro de Investigación y Desarrollo Ecuador
Tel.: + (593) 04 2037524
http.:/www.cidecuador.org

ISBN: 978-9942-679-46-8
https://doi.org/10.33996/cide.ecuador.DH2679260

Dirección editorial: Lic. Pedro Misacc Naranjo, Msc.
Coordinación técnica: Lic. María J. Delgado
Diseño gráfico: Lic. Danissa Colmenares
Diagramación: Lic. Alba Gil
Fecha de publicación: febrero, 2025

Guayaquil – Ecuador

La presente obra fue evaluada por pares académicos experimentados en el área

Catalogación en la Fuente

Fundamentos de Programación Orientada a Objetos (POO) / Andrés Marcelo Salinas Copo, Luis Tarquino Vinueza Rodríguez, Jorge Giovanni Valdiviezo Rodríguez, René Omar Villa López, David Leonardo Guevara Aulestia, Cecilia Estefania Urquizo Alvarez, Deysi Emilene Moya Ibarra, Irma Victoria Espín Mendoza. -Ecuador: Editorial CIDE, 2025.

156 p.: incluye tablas, figuras; 21,6 x 29,7 cm.
ISBN: 978-9942-679-46-8
1. Programación 2. Software

Dedicatoria

A los docentes que han contribuido a la realización de este libro, por su dedicación, conocimiento y compromiso con la educación. Este trabajo es el resultado de su pasión por enseñar y su constante esfuerzo por formar a las futuras generaciones de programadores.

Gracias por su visión, su entrega, y por ser una fuente de inspiración para sus estudiantes y colegas. Sin su colaboración, este proyecto no habría sido posible.

Agradecimiento

Quiero expresar mi más profundo agradecimiento a todas las personas que hicieron posible la realización de este libro.

A los docentes que aportaron su tiempo, conocimiento y experiencia, y que con su dedicación ayudaron a dar forma a este proyecto. Su pasión por la enseñanza y su compromiso con la excelencia son fundamentales para el éxito de esta obra.

Agradezco también a los estudiantes, cuyas preguntas, curiosidad y deseo de aprender han sido una constante fuente de inspiración. Este libro es, en parte, fruto de su esfuerzo y de las lecciones aprendidas en el camino.

A mis colegas y amigos, por su apoyo y retroalimentación en cada etapa del desarrollo de este proyecto. Su contribución ha sido invaluable.

Y, por último, a nuestras familias, por su paciencia, apoyo incondicional y comprensión a lo largo de este proceso. Sin ustedes, este libro no habría sido posible.

Semblanza de autores

Andrés Marcelo Salinas Copo

0000-0002-0465-7638

amscc1@gmail.com

Profesional con una amplia trayectoria en el ámbito académico y laboral. Es Técnico Mecánico Automotriz, Profesor de Segunda Enseñanza en Informática Licenciado en Informática y Computación, Ingeniero en Sistemas, títulos que consolidan su formación en el área de las Ciencias Computacionales. En el ámbito académico, ha impartido clases en diferentes unidades educativas tanto fiscales como particulares, en la Universidad de las Fuerzas Armadas ESPE y desde hace 13 años se desempeña como docente en el Instituto Superior Tecnológico Bolívar en diferentes carreras. A lo largo de su carrera, ha sido reconocido por diversos méritos académicos por su puntualidad en su asistencia como en su entrega de documentación.

Luis Tarquino Vinueza Rodríguez

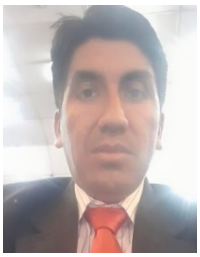
https://orcid.org/0009-0003-1430-8612

http://fantasma_vinuez@hotmail.com

Profesional con una amplia trayectoria en el ámbito académico y laboral. Ingeniero en Sistemas, actualmente egresado de la maestría en Pedagogía en Entornos Digitales, títulos que consolidan su formación en el área de Ciencias de la Educación. En el ámbito académico, ha impartido clases en institutos superiores y unidades educativas particulares y desde hace 7 años se desempeña como docente en el Instituto Superior Tecnológico Bolívar en diferentes carreras. A lo largo de su carrera, ha sido reconocido por diversos méritos académicos tanto como líder de unidades, por su puntualidad en su asistencia como en su entrega de documentación.

Jorge Giovanni Valdiviezo Rodríguez
https://orcid.org/0009-0001-0612-072X
jorge.valdiviezo@educacion.gob.ec

Profesional en el ámbito educativo con una trayectoria de 14 años con nombramiento del Ministerio de Educación en el Distrito 18Do1 Ambato 1 de Educación en la Unidad Educativa "Bolívar". Obtiene un título de tercer nivel como Ingeniero en Sistemas, actualmente egresado de la maestría en Pedagogía en Entornos Digitales, títulos que consolidan su formación en el área de Ciencias de la Educación. Se desempeña como docente de bachillerato en la figura profesional de informática, compartiendo su conocimiento en los diferentes módulos formativos que requieren los diferentes cursos del bachillerato. Por su labor profesional ha sido responsable por 8 años el cargo de docente CEL – TIC,s, coordinador provincial de INEVAL en el proceso de evaluación docente y otros cargos que los desempeño con responsabilidad.

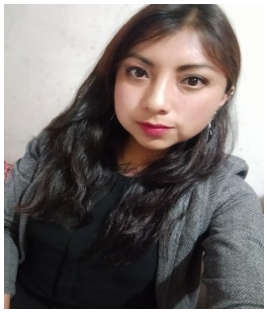


René Omar Villa López
ORCID 0009-0009-4659-4637
Memo_re7@yahoo.es

Profesional en el campo de las Ciencias Computacionales. Es Ingeniero en Sistemas Informáticos y Magíster en Sistemas de Telecomunicaciones, lo que respalda su sólida formación en tecnología y telecomunicaciones. A lo largo de su trayectoria, ha combinado la práctica profesional con la docencia. Ha impartido clases en unidades educativas fiscales y, durante los últimos seis años, se ha desempeñado como docente en el Instituto Superior Tecnológico Bolívar, donde ha contribuido a la formación de estudiantes en diversas carreras. Su compromiso con la educación y la excelencia le ha valido múltiples reconocimientos académicos, destacándose por su puntualidad, responsabilidad y rigurosidad en la entrega de documentación.

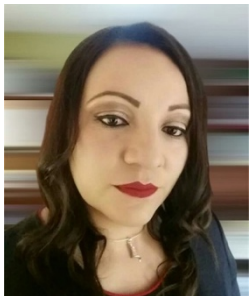
David Leonardo Guevara Aulestia
0009-0000-4449-2068
http://david@softeduc.com

Licenciado en Ciencias de la Educación con especialidad en Informática y Computación, y Magíster en Tecnología de la Información y Multimedia Educativa. Además, es experto en procesos e-learning. Con una trayectoria de 20 años como docente en posgrados y bachillerato, ha dedicado su carrera a la enseñanza y la investigación en diversas áreas. Entre sus intereses se encuentran los entornos virtuales de aprendizaje, el diseño y desarrollo web, el Internet de las cosas, la lógica de programación, los usos y aplicaciones ofimáticas, los paquetes contables, así como las redes y telecomunicaciones. Su experiencia en el ámbito del bachillerato técnico ha sido fundamental para la formación de futuros profesionales en estas disciplinas, especialmente en los procesos de planificación y organización. Además de su pasión por la tecnología y la educación, disfruta de la poesía y el arte, y es un apasionado de la investigación y la familia.



Cecilia Estefania Urquizo Alvarez
https://orcid.org/0009-0001-3559-9274
Stefis.urquizo93@gmail.com

Destacado profesional con una amplia trayectoria en el ámbito académico y laboral. Es Tecnóloga en redes, ensamblaje y mantenimiento de computadoras. Con más de 4 años de experiencia, ha desempeñado funciones clave en La Secretaría de Educación Superior, Ciencia, Tecnología e Innovación, donde ha aportado significativamente al desarrollo de Docencia. Su carrera ha estado marcada por la investigación, la docencia.



Deysi Emilene Moya Ibarra
0009-0001-7080-0009
http://emoya695@gmail.com

Profesional destacada en el campo de la informática y la educación. Posee un sólido bagaje académico, siendo Ingeniera en Sistemas, Tecnóloga en Informática y Técnica Superior en Programación de Sistemas, lo que respalda su formación en el ámbito de las Ciencias Computacionales. En su trayectoria académica, ha impartido clases en diversas instituciones del cantón Santiago de Píllaro, trabajando con estudiantes desde educación inicial hasta tercero de bachillerato. Su dedicación y compromiso con la enseñanza la llevaron a desempeñarse como docente en el Instituto Superior Tecnológico Bolívar durante siete años, donde contribuyó a la formación de numerosos estudiantes, demostrando su pasión por la educación y su compromiso con el desarrollo académico de sus estudiantes. Su trayectoria se caracteriza por un enfoque en la excelencia educativa y un deseo constante de mejorar la calidad de la enseñanza en su comunidad.

Irma Victoria Espín Mendoza
0009-0006-6968-8204

http://Isemiv@outlook.com

Ingeniera en Sistemas Computacionales e Informáticos por la Universidad Técnica de Ambato. Posteriormente, obtuvo una Maestría en Pedagogía con mención en Educación Técnica y Tecnológica por la Pontificia Universidad Católica del Ecuador, donde fue reconocida con honores Cum Laude. A lo largo de su trayectoria profesional, Victoria ha combinado su experiencia en ingeniería de sistemas con su pasión por la educación, participando en diversos proyectos que integran la tecnología en entornos educativos. Su enfoque se centra en desarrollar soluciones innovadoras que mejoren los procesos de enseñanza y aprendizaje, adaptándose a las necesidades del

contexto ecuatoriano. Además de su labor profesional, Victoria se ha destacado por su compromiso con la formación continua, asistiendo a conferencias y talleres relacionados con la tecnología educativa y la pedagogía moderna. Su dedicación y habilidades la han posicionado como una referente en la implementación de estrategias tecnológicas en el ámbito educativo.

Índice

Capítulo 3
Manejo de errores y excepciones

Capítulo 4
Interfaz gráfica

Capítulo 5
Conexión a BD

Capítulo 6
Entornos de desarrollo en
Lenguaje JAVA

Prólogo

La programación orientada a objetos (POO) no es solo un paradigma más en el mundo del desarrollo de software; es una forma de pensar, de estructurar ideas y de dar solución a problemas complejos mediante la organización del código en módulos más comprensibles, reutilizables y escalables. Desde su concepción, ha demostrado ser una herramienta poderosa para aquellos que buscan desarrollar aplicaciones de calidad, eficientes y mantenibles.

El presente libro nace con el objetivo de servir como una guía práctica y completa para quienes desean dominar la POO, con especial énfasis en su aplicación utilizando el lenguaje Java. A lo largo de mi carrera profesional y académica, he observado cómo los estudiantes y desarrolladores pueden verse abrumados por la cantidad de conceptos que engloba la POO, pero he sido testigo también de cómo, al dominar estos conceptos, se alcanza una mayor capacidad para abordar proyectos más complejos y desafiantes.

Este libro busca ser accesible, pero a la vez profundo, abarcando desde los conceptos más fundamentales, como la definición de clases, objetos y métodos, hasta temas avanzados como el polimorfismo, la herencia y el manejo de excepciones. Todo ello, respaldado por ejemplos prácticos y ejercicios diseñados para aplicar los conocimientos adquiridos de manera inmediata.

Una de las principales motivaciones detrás de este proyecto ha sido la necesidad de un recurso que no solo explique los aspectos teóricos, sino que ofrezca una visión práctica, enfocada en la resolución de problemas reales. En este sentido, se ha puesto especial atención a la creación de interfaces gráficas y a la integración de aplicaciones con bases de datos, dos componentes esenciales en el desarrollo de software moderno.

Además, el libro incluye una introducción sólida a Java, un lenguaje que ha sido clave en la evolución de la POO y que sigue siendo uno de los más utilizados y demandados en la industria tecnológica. Con un enfoque práctico y didáctico, el lector aprenderá a sacar el máximo provecho de este lenguaje y a utilizar las herramientas que ofrece para implementar de manera efectiva los principios de la POO.

Confío en que este libro será una valiosa herramienta tanto para quienes inician su camino en el mundo de la programación orientada a objetos, como para aquellos que desean perfeccionar sus habilidades en Java. Espero que cada lector encuentre en estas páginas no solo respuestas a sus preguntas, sino también nuevas ideas y soluciones para los retos que enfrentan en su día a día como desarrolladores.

Introducción

La programación orientada a objetos (POO) ha transformado la forma en que los desarrolladores abordan la creación de software. Desde su introducción, este paradigma ha facilitado el desarrollo de aplicaciones más modulares, mantenibles y escalables. Este libro está diseñado para ofrecer una comprensión integral de los principios de la POO y su aplicación en el lenguaje Java, uno de los más populares y versátiles para la programación orientada a objetos.

En las primeras secciones del libro, se abordarán los conceptos básicos de la POO, incluyendo su historia, las características que la distinguen, como la abstracción, encapsulamiento, herencia y polimorfismo, así como la estructura básica de clases, objetos, atributos y métodos. La inclusión de diagramas de flujo ayudará a visualizar el flujo lógico de los programas y su estructura.

Posteriormente, se profundizará en el manejo de excepciones, un aspecto clave para el desarrollo de aplicaciones robustas. El manejo adecuado de errores y excepciones permite prevenir y solucionar problemas que pueden surgir durante la ejecución del código, mejorando así la estabilidad del software.

La creación de interfaces gráficas es otro componente central en el desarrollo de aplicaciones modernas. En este libro, exploraremos cómo implementar interfaces gráficas interactivas utilizando Java, abordando temas como la creación de botones, listas, checkboxes, y la

gestión de eventos. Además, se cubrirán aspectos relacionados con el manejo de mensajes y excepciones en el contexto de aplicaciones gráficas.

Un aspecto vital en el desarrollo de software actual es la capacidad de interactuar con bases de datos. Este libro explicará cómo conectar aplicaciones Java con bases de datos utilizando JDBC, además de proporcionar una introducción al desarrollo de operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para gestionar la información de manera eficiente.

Finalmente, nos adentraremos en el lenguaje Java, mostrando cómo este lenguaje implementa los principios de la POO y cómo aprovechar sus características para crear software sofisticado y eficiente. A lo largo de este recorrido, los ejemplos y ejercicios proporcionarán una base sólida para que el lector pueda poner en práctica lo aprendido y desarrollar sus propias aplicaciones.

Este libro está dirigido tanto a estudiantes que se inician en la programación orientada a objetos como a profesionales que buscan profundizar su conocimiento en el uso de Java para crear soluciones robustas y eficientes.

CAPÍTULO 1

Introducción a la Programación Orientada a Objetos (POO)

1

Capítulo

1

Introducción a la Programación Orientada a Objetos (POO)

1.1. Introducción a la Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) representa un enfoque distintivo en el desarrollo de software, proporcionando un estilo y directrices específicas para su implementación. Este paradigma se fundamenta en la idea de clases y objetos, permitiendo descomponer un software en componentes simples y reutilizables (clases) que facilitan la creación de objetos únicos.

A través del tiempo, han emergido diversos paradigmas de programación. Lenguajes como COBOL, que pertenecen al tipo secuencial, y otros como Basic o C, de enfoque procedimental, priorizaron la lógica en lugar de los datos en sí. En contraste, lenguajes contemporáneos como Java, C# y Python han adoptado paradigmas distintos para la formulación de programas, destacando la Programación Orientada a Objetos como el más prevalente.

El enfoque de Programación Orientada a Objetos nos permite desplazar nuestra atención de la lógica programática hacia el concepto de objetos, que es fundamental en este paradigma. Esta perspectiva es especialmente valiosa en el desarrollo de sistemas grandes, ya que, en lugar de concentrarnos en funciones individuales, consideramos las interacciones y relaciones entre los diferentes elementos del sistema.

En este contexto, un programador estructura un programa organizando información y comportamientos relacionados dentro de una plantilla conocida como clase. Posteriormente, se generan objetos específicos a partir de dicha clase. Así, el software en su totalidad funciona mediante la interacción de múltiples objetos, formando así un sistema más complejo.

La Programación Orientada a Objetos (POO) se basa en cómo interpretamos y damos un significado a lo que nos rodea. Supongamos que deseas desarrollar un sistema para gestionar un zoológico. En lugar de enfocarte solo en algoritmos y estructuras de datos, la POO te sugiere que pienses en las entidades pertinentes al zoológico, como animales, cuidadores y visitantes.

En este modelo, cada entidad se transforma en un objeto que posee características (datos) y acciones (funcionalidades). Por ejemplo, un objeto llamado "Animal" podría tener atributos como especie, hábitat y edad, así como métodos para realizar acciones como alimentarse, moverse o interactuar con los cuidadores.

El aspecto central de la POO es la colaboración entre estos objetos. Tienen la capacidad de intercambiar información y trabajar

juntos para alcanzar un propósito. Por ejemplo, un objeto "Visitante" podría enviar una solicitud al objeto "Animal" para observarlo, y este respondería mostrando su comportamiento correspondiente.

La Programación Orientada a Objetos facilita la creación de código que no solo es estructurado sino también propenso a ser reutilizado, lo que simplifica su mantenimiento. Este enfoque ayuda a prevenir la redundancia en el código, contribuyendo así a la eficiencia del software. Asimismo, protege la integridad de los datos y oculta información confidencial a través de estrategias como la encapsulación y la abstracción, temas que examinaremos con más profundidad más adelante.

1.2. Uso de diagramas de flujo

La programación orientada a objetos (POO) es un paradigma que organiza el software como una colección de objetos que combinan datos y comportamientos. En este enfoque, los objetos se comunican entre sí mediante métodos, facilitando la modularidad y la reutilización del código. Los diagramas de flujo son herramientas visuales que permiten representar de manera clara y concisa los procesos y la lógica del programa. En este documento, exploramos el uso de diagramas de flujo en POO, cómo se integran en el proceso de desarrollo y sus beneficios.

1.2.1. Conceptos fundamentales de diagramas de flujo

Los diagramas de flujo son representaciones gráficas que utilizan símbolos estandarizados para ilustrar un algoritmo o un proceso. Los elementos principales incluyen:

- Ovalados: Indican el inicio y el final del proceso.
- Rectángulos: Representan instrucciones o acciones a seguir.
- Rombos: Indican decisiones que deben tomarse (condicionales).
- Flechas: Muestran la dirección del flujo del proceso.

Este tipo de diagramas proporciona una visión general de cómo se ejecuta un sistema y ayuda a clarificar procesos complejos. No solo son útiles para programadores, sino también para analistas y diseñadores, permitiendo una comunicación efectiva entre diferentes partes interesadas en un proyecto.

1.2.2. Diagramas de flujo en el contexto de la POO

En programación orientada a objetos, los diagramas de flujo se pueden utilizar en varias etapas del desarrollo de software:

- Diseño del Sistema: Antes de comenzar a codificar, es esencial tener un diseño claro del sistema. Los diagramas de flujo pueden ayudar a definir cómo interactúan los diferentes objetos, qué métodos poseen y cómo se pasan los datos entre ellos. Esto es especialmente útil en sistemas grandes y complejos.

- Documentación: A medida que el software se desarrolla, la documentación se vuelve vital. Los diagramas de flujo pueden servir como una forma de documentar cómo funciona el código, lo que facilita la comprensión y el mantenimiento futuros. Al incluir diagramas de flujo en la documentación, otros desarrolladores pueden comprender rápidamente la lógica subyacente.

- Análisis y Depuración: Durante la etapa de pruebas, los diagramas de flujo pueden utilizarse para analizar el flujo del programa y detectar posibles errores lógicos. Al visualizar el proceso en forma de diagrama, es más fácil identificar puntos donde la lógica puede fallar o donde no se está alcanzando el resultado esperado.

1.2.3. Beneficios de usar diagramas de flujo en POO

- Claridad Visual: Los diagramas de flujo proporcionan una representación visual de la lógica del programa, lo que facilita la comprensión. Esto es útil tanto para desarrolladores experimentados como para nuevos integrantes del equipo.
- Facilita la Comunicación: Al ser gráficos, los diagramas de flujo son una herramienta eficaz para comunicar ideas y procesos a personas no técnicas, como gerentes de proyecto o clientes. Esto ayuda a asegurar que todas las partes involucradas tengan una comprensión similar del sistema.
- Detección de Problemas: Visualizar el flujo de un programa puede hacer que los problemas en la lógica sean más evidentes. Los diagramas permiten a los desarrolladores seguir pasos específicos y verificar que se estén llevando a cabo las acciones correctas.
- Reutilización de Código: Al organizar el código en objetos, es más fácil identificar qué partes son susceptibles de reutilización. Un diagrama de flujo puede ayudar a resaltar estas áreas y facilitar la creación de bibliotecas de código reutilizables.

1.2.4. Ejemplo práctico de un diagrama de flujo en POO

Consideremos un ejemplo sencillo: un sistema de gestión de biblioteca donde se utilizan objetos como "Libro", "Usuario" y "Préstamo". Al crear un diagrama de flujo para el proceso de préstamo:

- **Inicio:** El usuario solicita un libro.
- **Decisión**: Se verifica si el libro está disponible.
- **S**i está disponible:
- **Crear un nuevo** objeto de tipo "Préstamo".
- **Asignar el libro y el usuario** al préstamo.
- **Actualizar** la disponibilidad del libro.
- **Mostrar** un mensaje de éxito.
- **Si no está** disponible:
- **Mostrar un mensaje** de error.
- **Fin**: Terminar el proceso.

Este diagrama ilustra el flujo lógico detrás de la funcionalidad de préstamo, y puede ser utilizado como referencia tanto en el diseño como en la implementación del código.

Es importante recordar que, aunque los diagramas de flujo son herramientas poderosas, deben utilizarse en combinación con otras técnicas de modelado y diseño. Algunas consideraciones y buenas prácticas incluyen:

Mantener Simplicidad: un diagrama de flujo debe ser claro y no sobrecargado de información. Cada diagrama debe centrarse en un área específica para evitar confusiones.

Actualización Constante: a medida que el código evoluciona, también lo hará la lógica del programa. Asegúrate de actualizar los diagramas de flujo para reflejar los cambios.

Interacción con otros Diagramas: completa los diagramas de flujo con otros tipos de diagramas, como diagramas de clases, diagramas de secuencia o diagramas de caso de uso, para una representación más completa del sistema.

Los diagramas de flujo son herramientas valiosas en el desarrollo de software orientado a objetos. Facilitan el diseño, la comunicación y el mantenimiento del sistema al proporcionar una forma visual de entender el flujo del proceso. Al utilizarlos de manera efectiva, los desarrolladores pueden mejorar la calidad de su trabajo y contribuir a la creación de software más eficiente y comprensible.

1.3. Características de la programación

La Programación Orientada a Objetos (POO) es un paradigma que utiliza "objetos" para representar datos y métodos. Este enfoque se centra en la agrupación de datos y comportamientos en una entidad cohesiva. Entre las principales características de la POO, encontramos:

- **Encapsulamiento:** permite ocultar los detalles internos de un objeto, exponiendo solo lo necesario. Esto se logra mediante modificadores de acceso, como public, private y protected. El encapsulamiento fomenta la protección de la integridad del objeto, evitando que partes externas del programa manipulen sus propiedades de forma inadecuada.

- **Abstracción:** simplifica la complejidad al representar las características esenciales de un objeto, ignorando detalles no relevantes. La abstracción permite a los programadores enfocarse en las interacciones de alto nivel sin preocuparse por la implementación interna.
- **Herencia:** facilita la creación de nuevas clases basadas en clases existentes, permitiendo la reutilización de código y la creación de jerarquías de clases. Una clase derivada hereda atributos y métodos de una clase base, lo que minimiza la redundancia.
- **Polimorfismo:** permite que diferentes clases respondan a la misma operación de distintas maneras. Esto se implementa mediante la sobrecarga de métodos y la redefinición de métodos, aumentando así la versatilidad del código.

1.4. Identificadores, separadores, operadores, tipos de datos, variables, expresiones

Antes de diseñar y desarrollar programas orientados a objetos, es crucial entender algunos componentes fundamentales que constituyen un lenguaje de programación.

1.4.1. Separadores

Los separadores son símbolos utilizados para organizar el código y diferenciar entre sus diferentes componentes. Los más comunes son:

- Llaves {}: Delimitan el inicio y el fin de una clase o un bloque de código.

- Punto y coma; : Indica el final de una instrucción.
- Paréntesis (): Usualmente se utilizan para delimitar los parámetros de un método o función.
- Corchetes []: Se emplean comúnmente para definir arreglos.

Ejemplo en Java:

Figura 1

Ejemplo separadores

```
public class Ejemplo {
    public static void main(String[] args) {
        System.out.println("Hola, POO"); // punto y coma se usa para terminar la sentencia
    }
}
```

Nota. Elaboración propia

1.4.2. Operadores

Los operadores son símbolos que permiten realizar operaciones sobre variables y valores. Se clasifican en:

- **Aritméticos**: +, -, *, / (para realizar operaciones matemáticas).
- **Relacionales**: ==, !=, >, <, >=, <= (para comparar valores).
- **Lógicos**: &&, ||, ! (para operaciones lógicas).

Figura 1

Ejemplo operadores

```
a = 15
b = 25
resultado = a + b # operador aritmético
if a < b:  # operador relacional
   print ("a es menor que b")
```

Nota. Elaboración propia

1.4.3. Tipos de datos

Los tipos de datos definen el tipo de valor que puede almacenar una variable.

Los tipos de datos más comunes son:

- **Primitivos**: int, float, char, boolean en Java.
- **Compuestos**: Array, String, List, Map, entre otros.

Ejemplo en C#:

Figura 2

Ejemplo tipo de datos

```
int edad = 25; // tipo de dato entero
string nombre = "Juan"; // tipo de dato cadena
bool esEstudiante = true; // tipo de dato booleano
```

Nota. Elaboración propia

1.4.4. Variables

Las variables son espacios en memoria que se utilizan para almacenar datos. Se definen mediante un tipo de dato y un identificador.

Ejemplo en Java:

Figura 3

Ejemplo declaración variables

```
int edad; // declaración de variable
edad = 25; // asignación de valor
```

Nota. Elaboración propia

1.4.5. Expresiones

Las expresiones son combinaciones de variables, operadores y valores que producen un resultado. Pueden ser simples (como x + y) o complejas.

Figura 4

Ejemplo expresiones

```
# Python
x = 10
y = 5
suma = x + y  # expresión que resulta en 15
```

Nota. Elaboración propia

1.5. Tipos de estructuras, lazos repetitivos, secuenciales

Las estructuras de control permiten controlar el flujo de un programa y se dividen en tres tipos:

1.5.1. Estructuras lazos repetitivos(bucles)

Permiten ejecutar un bloque de código varias veces. Los tipos más comunes son for, while, do-while.(y for each, conceptualice cada bucle)

Ejemplo en Python:

```
# Python
for i in range(5):
print(i)  # imprime números del 0 al 4
```

1.5.2. Estructuras secuenciales

Las instrucciones se ejecutan una tras otra en orden.

Ejemplo:

```
// Java
System.out.println("Inicio");
System.out.println("Fin");
```

1.5.3. Estructuras de selección

Permitidas mediante if, switch, donde se evalúa una condición.

```
// C#
int numero = 10;
if (numero > 0) {
    Console.WriteLine("Positivo");
}
```

1.6. Definición de una clase: atributos y métodos Get, Set

Una clase es una estructura que agrupa datos (atributos) y comportamientos (métodos) relacionados.

Atributos: Son características de la clase que suelen declararse al inicio.

Métodos Get y Set: Se utilizan para acceder y modificar atributos privados.

Ejemplo en Java:

```
public class Persona {
        private String nombre; // atributo privado
        // método Getter
        public String getNombre() {
        return nombre;
        }
        // método Setter
        public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

1.7. Constructores, Objetos

Un constructor es un método especial que se llama cuando se crea un objeto de una clase. Se utiliza para inicializar atributos.

(

Ejemplo en Java:

```
 public class Vehiculo {
        private String marca;
        // Constructor
        public Vehiculo(String marca) {
        this.marca = marca;
        }
}
```

Para crear un objeto, se utiliza la palabra clave new:

```
// Creación de un objeto
Vehiculo miVehiculo = new Vehiculo("Toyota");
```

CAPÍTULO 2

Características de la Programación Orientada a Objetos (POO)

2

Capítulo

2

Características de la Programación Orientada a Objetos (POO)

2. Características de la POO

El paradigma de la programación orientada a objetos consta de representaciones de la realidad. Cubre conceptos fundamentales como clases, objetos, propiedades y métodos, y se caracteriza por el uso de abstracción, herencia, encapsulación y polimorfismo de datos. Estas propiedades deben estudiarse y comprenderse antes de poder utilizarlas en la programación orientada a objetos.

2.1. Abstracción

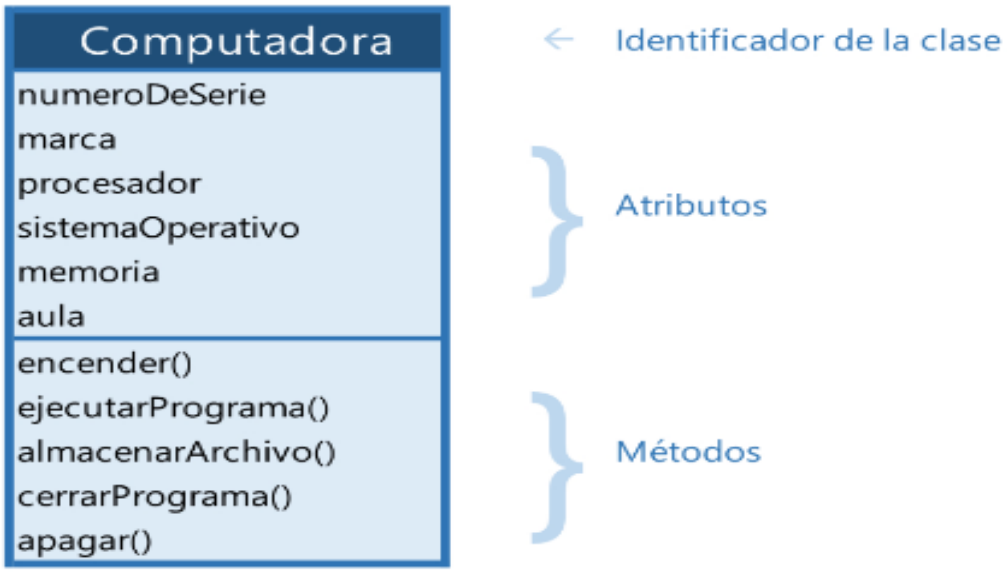
La abstracción es un proceso que permite seleccionar una entidad de realidad específica, sus propiedades y las funciones que realiza, representada por una clase que contiene las propiedades y métodos de esa clase.

En la programación orientada a objetos, es importante abstraer los métodos y propiedades que son comunes a un grupo de objetos

agrupados en una clase. El hardware de la computadora tiene características como marca, color, número de serie, cantidad de memoria instalada, capacidad del disco duro y la tecnología utilizada para el almacenamiento secundario. Además, también se pueden integrar otros objetos como placas base, procesadores, discos duros, módulos de memoria y monitores. En esta abstracción, consideramos la funcionalidad de un dispositivo informático en lugar de la forma en que se construyen internamente los componentes anteriores, pero son relevantes para los diseñadores e ingenieros de dispositivos informáticos. Es posible que no sea necesario modelar todo el inventario. La abstracción indica que solo se incluyen vocabularios o lenguajes, funciones y comportamientos específicos relevantes para el dominio del problema que se aborda. Sólo se deben modelar los detalles necesarios.

Figura 5

Esquematización de abstracción



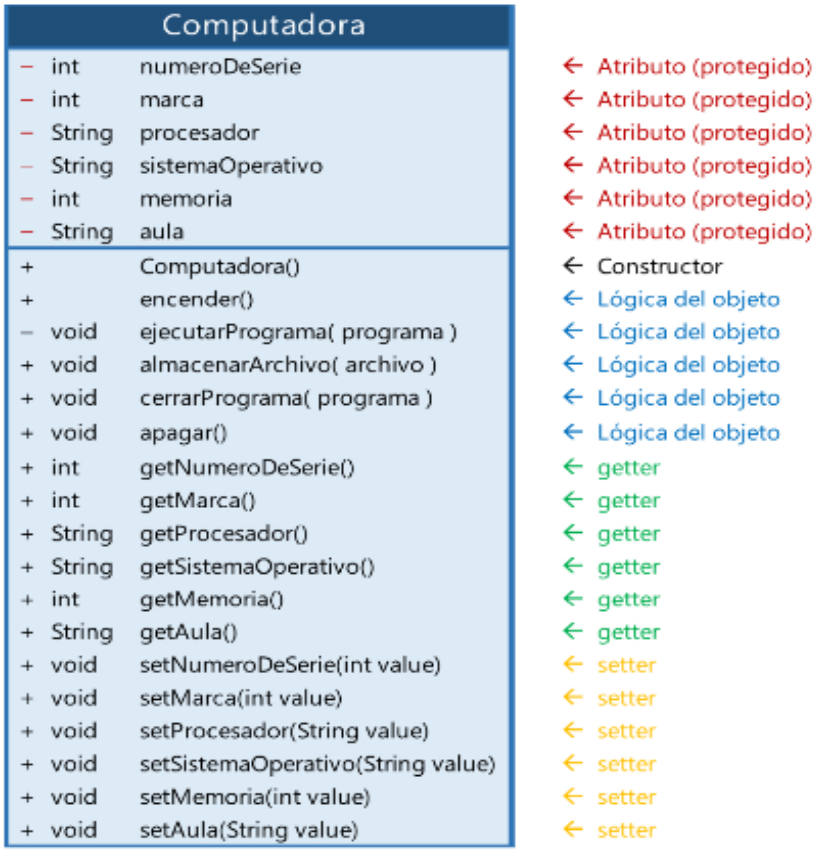
Nota Adaptado de (Bailón & Baltazar, 2021)

2.2. Encapsulamiento

En programación, la encapsulación de objetos principales normalmente protege la información o el estado de las propiedades para que la información del objeto no pueda verse ni modificarse sin los mecanismos adecuados. Para hacer esto, debe utilizar métodos para recuperar la información (recuperada) y asegurarse de que la información especificada corresponda al objeto. Por ejemplo, nuestra clase de informática se ve así:

Figura 6

Esquematización de encapsulamiento

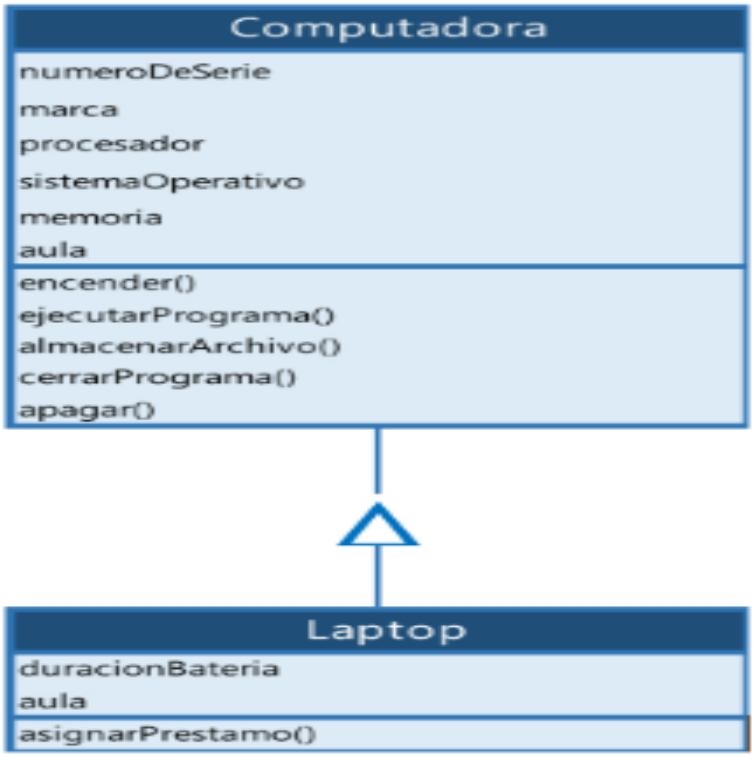


Nota Adaptado de (Bailón & Baltazar, 2021)

2.3. Herencia

Figura 7

Esquematización de herencia



La herencia le permite reutilizar el código escrito en cada clase "heredando" o extendiendo las propiedades de un objeto a sus "descendientes" o clases derivadas. Cuando se habla de una computadora o una computadora portátil, se sabe que sigue siendo una computadora y se puede tratarla como tal, pero se puede agregar características como la duración de la batería y a qué está asignada porque no estará en el aula. En este caso, la computadora portátil sigue siendo una computadora, tiene todas las propiedades y métodos, pero agrega dos propiedades y un método a la definición original, la llamada superclase (la superclase en la cima de la jerarquía).

Nota Adaptado de (Bailón & Baltazar, 2021)

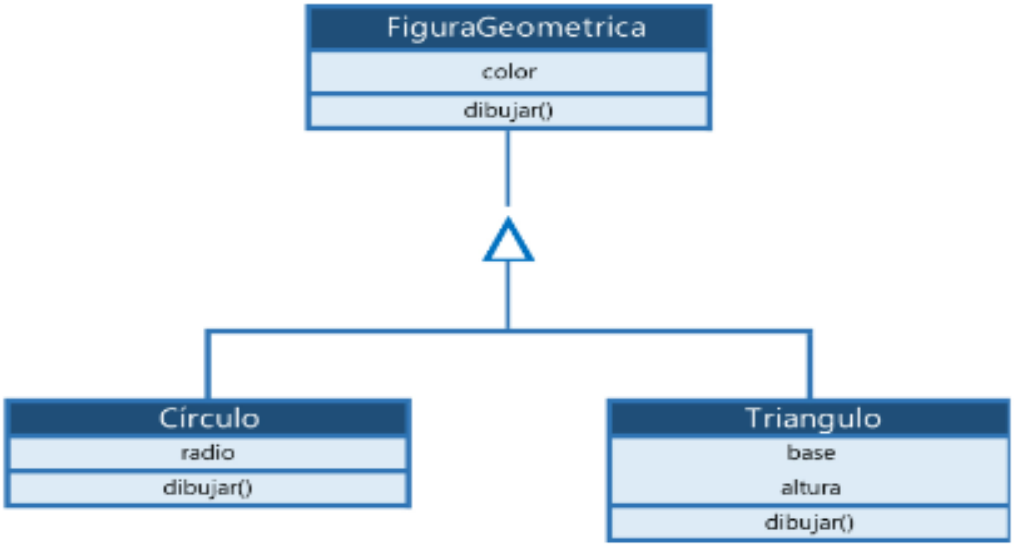
2.4. Polimorfismo

Polimorfismo en una colección de objetos con herencia, si las superclases especializadas tienen métodos con la misma definición o firma, responderán en consecuencia al recibir el mismo mensaje.

Por ejemplo, si tenemos una superclase FiguraGeometrica que se especializa en círculos y triángulos, y si el programa asume que todos son formas geométricas, entonces cuando se envía un mensaje draw() a los elementos de la colección, los triángulos se dibujarán usando . regla y se dibujarán usando un compás. Hacer lo mismo. Ambas son formas geométricas, pero están dibujadas de diferentes maneras.

Figura 8

Esquematización de polimorfismo



Nota Adaptado de (Bailón & Baltazar, 2021)

2.5. Implementación de métodos y clases

Los módulos de integración se pueden implementar utilizando archivos de clase Java. El uso de archivos de clase Java elimina la necesidad de registrar y configurar componentes de integración. Además, la implementación de todos los módulos de integración subyacentes es transparente para el producto de gestión de procesos.

Los módulos de integración de clases Java deben implementar la interfaz psdi.iface.mic.ServiceInvoker. Llamadas de servicio La interfaz Java se incluye en el archivo businessobjects.jar. Incluya las clases Java del módulo de integración en la ruta de clases del sistema en tiempo de ejecución.

Las clases son los tipos o patrones utilizados para crear objetos.

La sintaxis para declarar una clase es:

```
[modificador] class <nombre> {
        // Campos de la clase
        // Métodos de la clase
 }
```

Modificador de acceso

Los modificadores de acceso se utilizan para restringir el acceso a campos o métodos de una clase; es decir nos introducen en el concepto de encapsulación. La encapsulación intenta controlar de alguna manera el acceso a los datos que componen un objeto o instancia, y de esta manera podemos decir que una clase y sus objetos, usando modificadores de acceso (especialmente privados), son objetos

encapsulados. Los modificadores de acceso nos permiten brindar un mayor nivel de seguridad a nuestras aplicaciones al restringir el acceso a varias propiedades, métodos y constructores, garantizando que los usuarios deben seguir una "ruta" que especificamos para acceder a la información.

Es probable que nuestra aplicación sea utilizada por otros programadores o usuarios con cierto nivel de experiencia, al utilizar modificadores de acceso podemos asegurar que otros programadores o usuarios no cambien valores de forma incorrecta. Normalmente, se puede acceder a las propiedades mediante métodos getter y setter, ya que las propiedades de la clase deben ser estrictamente privadas.

Nota. Siempre es una buena idea mantener privadas las propiedades de la clase, por lo que cada propiedad debe tener sus propios métodos getter y setter para obtener y establecer el valor de la propiedad en consecuencia. Nota 2. Cuando utilice una clase de otro paquete, utilice import para importarla. Si dos clases están en el mismo paquete, no es necesaria ninguna importación, pero esto no significa que se pueda acceder a sus componentes directamente. Echemos un vistazo más de cerca a cada modificador de acceso.

Modificador de acceso private

El modificador private en Java es el más restrictivo de todos los modificadores, básicamente, cualquier elemento que sea privado en una clase solo puede ser accedido por la clase misma y no puede ser accedido por ninguna otra cosa. Es decir, por ejemplo, si una propiedad es privada, sólo podrán acceder a ella métodos o constructores de la misma clase. Ninguna otra clase puede acceder a ellos independientemente de su relación.

```
package aap.ejemplo1;
public class Ejemplo1{
	private int atributo1;//Este atributo es privado
	private int contador = 0; //Contador de registro

//Si un atributo es privado podemos crear método get y set ...
	//... para éste y permitir el acceso a él desde otras instancias

 public void setAtributo1(int valor) {
		contador++;//Contador que lleva el registro de ediciones del atributo1
		atributo1 = valor;//Establecemos el valor del atributo
	}

	public int getAtributo1() {
		return atributo1;//Retornamos el valor actual del atributo
	}

	//Get para el contador
	public int getContador(){
		return contador;
	}

	//Notar que no ponemos un set, pues no nos interesa que el contador pueda ser cambiado.
}
```

En el ejemplo anterior se observó, que se tiene una propiedad privada y solo se permite el acceso a ella a través de los métodos get y set. Tenga en cuenta que estos métodos son públicos, por lo que cualquiera puede acceder a ellos. Lo realmente interesante de los

métodos getter y setter es que permiten hacer cualquier cosa, como realizar un seguimiento de cuántas veces se establece el valor de una propiedad, lo que permite mantener el sistema sin ningún problema. También hay que tener en cuenta que, dado que los métodos getter y setter son específicos de la clase, no hay ningún problema para acceder a las propiedades directamente.

Modificador por defecto (default)

Java nos da la opción de no usar modificadores de acceso, si no lo hacemos, el elemento tendrá un derecho de acceso llamado acceso predeterminado o estándar, que permite que la clase misma y las clases del mismo paquete accedan a dicho elemento (por eso es importante declarar siempre un paquete para la clase).

```
package aap.ejemplo2;
public class Ejemplo2{
    private static int atributo1;//Este atributo es privado
    static int contador = 0; //Contador con acceso por defecto

    public static void setAtributo1(int valor){
        contador++;//Contador que lleva el registro de ediciones del atributo1
        atributo1 = valor;// Se establece el valor del atributo
    }

    public static int getAtributo1(){
        return atributo1;//Retorna el valor actual del atributo
    }
}
```

```
package aap.ejemplo2;
public class Ejemplo2_1{
	public static int getContador(){
		return Ejemplo2.contador;//Accede directamente al contador desde otra clase
	}
}
```

Modificador de acceso protected

El modificador de acceso permite acceso a los componentes con dicho modificador desde la misma clase, clases del mismo paquete y clases que hereden de ella (incluso en diferentes paquetes).

El modificador de acceso protected da acceso a componentes con ese modificador de la misma clase, clases en el mismo paquete y clases que heredan del paquete (incluso en paquetes diferentes).

```
package aap.ejemplo3;
public class Ejemplo3{
	protected static int atributo1;//Atributo protected
	private static int atributo2; //Atributo privado
	int atributo3;//Atributo por default

 public static int getAtributo2(){
return atributo2;
	}
}
package aap.ejemplo3_1;
import aap.ejemplo3.Ejemplo3;//Es necesario importar la clase del ejemplo 3
```

```
public class Ejemplo3_1 extends Ejemplo3{
	public static void main(String[] args){
		//Las siguientes dos líneas generan error, pues atributo2 es privado y atributo 3 es default
//System.out.println(atributo2);
		//System.out.println(atributo3);
		System.out.println(atributo1);//Sí se tiene acceso a atributo1
	}
}
```

Modificador public

El modificador de acceso public es el más permisivo de todos los modificadores de acceso. En esencia, public es (lógicamente) lo opuesto a private en todos los aspectos. Esto significa que, si los miembros de una clase son public, obteniendo datos o acceder a cualquier clase sin importar el paquete o su fuente.

```
package aap.ejemplo4;

public class Ejemplo4{
	public static int atributo1; //Atributo público

	public static void metodo1(){
		System.out.println("Método público");
	}
}

package paquete.externo;
import aap.ejemplo4.Ejemplo4; // se importa la clase del ejemplo4
```

```
public class ClaseExterna{
        public static void main(String[] args){
                System.out.println(Ejemplo4.atributo1);
                //Se tuvo acceso directo por ser publico

Ejemplo4.metodo1(); //Metodo1 también es publico
        }
}
```

Tabla 1

Resume cómo funcionan los modificadores de acceso en Java.

Modificador	La misma clase	Mismo paquete	Subclase	Otro paquete
private	Sí	No	No	No
default	Sí	Sí	No	No
protected	Sí	Sí	Sí/No	No
public	Sí	Sí	Sí	Sí

Nota. Elaboración propia

Especifica el contexto de los modificadores de acceso protegido y el acceso desde sus clases. Un error común es pensar que se puede crear un objeto de la clase padre y luego usar el acceso protegido para acceder a las propiedades sin ningún problema, pero este no es el caso porque el modificador protected nos permite acceder a las propiedades de la clase padre. Acceda a activos heredados.

Métodos

En Java, los métodos son bloques de código que agrupan un conjunto de instrucciones pertinentes. Sin embargo, los parámetros son variables que se utilizan en los métodos de recopilación de información. Los argumentos son los valores reales que se transmiten a los métodos

cuando se les llama. En resumen, son componentes esenciales para crear aplicaciones Java efectivas.

Los métodos son bloques de código que se utilizan para completar tareas particulares. Cada método tiene un nombre y puede recibir una serie de parámetros, que son variables que brindan información necesaria para que el método realice su tarea. Estos parámetros sirven como entrada para el método, lo que le permite manipular los valores que recibe.

Se pueden realizar varias operaciones dentro del cuerpo del método, como cálculos matemáticos, manipulación de datos o incluso llamar a otros métodos. Puede devolver un resultado utilizando la palabra clave return seguida del valor que desea devolver una vez que el método haya completado sus operaciones.

Una de las principales ventajas de los métodos en Java es la capacidad de reutilizar el código. Al definir un método, podemos evitar repetir código innecesariamente al llamarlo a diferentes secciones del programa. Esto permite un mayor modularidad en la aplicación y simplifica el código.

Por otro lado, es importante tener en cuenta el término "recursión", que se refiere a un método de resolución de problemas que se utiliza repetidamente hasta llegar a una condición de terminación.

Una ilustración de la estructura de un método en Java es la siguiente:

```
public int sumar(int a, int b) {
    int resultado = a + b;
    return resultado;
}
```

Se tiene un método llamado sumar en este ejemplo que recibe dos parámetros de tipo int, a y b. El cuerpo del método incluye la operación de suma entre los dos parámetros y el resultado se almacena en una variable conocida como resultado. Luego, para devolver el resultado de la suma, se usa la palabra clave return después del valor de resultado. El resultado de esta técnica es un valor entero (int).

Tipos de métodos en Java

De acuerdo con la estructura y funcionalidad, se puede encontrar varios tipos de métodos en Java:

Métodos sin retorno: Estos métodos no generan valor. Se utilizan principalmente para modificar el estado de un objeto o ejecutar un conjunto de instrucciones.

```
public void saludar() {
    System.out.println("¡Hola, mundo!");
}
```

Métodos con retorno: Estos métodos generan un valor al ejecutarlo. Este valor puede ser de cualquier tipo de dato que Java admite, como enteros, cadenas de texto y booleanos. Por ejemplo, el resultado de una operación aritmética es el siguiente:

```
public int sumar(int a, int b) {
    return a + b;
}
```

Métodos estáticos: En lugar de estar relacionados con una instancia de clase, los métodos estáticos están relacionados con la clase. Estos métodos se pueden utilizar directamente sin crear un objeto.

```
public static void mostrarMensaje() {
    System.out.println("¡Hola, soy un ejemplo de método estático!");
}
```

Declaración de métodos

La declaración de un método en Java sigue una estructura determinada. En Java, este es un ejemplo de cómo se declara un método:

```
public tipo_de_dato_devuelto nombre_metodo(tipo_de_dato parametro1, tipo_de_dato parametro2) {
    // código a ejecutar
    return resultado;
}
```

Para declarar un método, se debe especificar el tipo de dato que devolverá, seguido del nombre del método y los parámetros que recibe, si los tiene. El cuerpo del método contiene el código a ejecutar y, en aquellos métodos que tienen valores de retorno, se utiliza la palabra clave return para devolver un resultado.

En el caso de métodos que no devuelven nada y no admiten parámetros, se debe incluir la clase reservada vacía al comienzo de la declaración del método, la estructura es la siguiente:

```
public void metodoSinRetorno() {
    // Código del método aquí
    System.out.println("Este es un método que no devuelve nada.");
    // Se puede realizar acciones dentro del método, pero no retornará un valor.
}
```

La llamada de los métodos

Se puede llamar a un método desde otro lugar de nuestro código una vez que se lo ha declarado. El nombre de un método se llama seguido de paréntesis, y si es necesario, se pueden pasar los argumentos correspondientes.

```
saludar(); // Llamada a un método sin retorno
int resultado = sumar(5, 3); // Llamada a un método con retorno
```

Retorno de los métodos

Los métodos con retorno de Java utilizan la palabra clave "retorno" para devolver un valor donde se llaman. El tipo de datos que se devuelve debe cumplir con las especificaciones del método.

```
public int sumar(int a, int b) {
    return a + b;
}
```

En este caso, se indica en la descripción del método que el tipo de valor que se retorna es de tipo entero (int).

Ejemplos

Estos son algunos ejemplos útiles de cómo usar los diferentes tipos de métodos en Java:
Ejemplo de métodos sin valor devuelto

```
public void saludar(String nombre) {
    System.out.println("¡Hola, " + nombre + "!");
}

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

```
saludar("Juan");
}
```

Ejemplo de métodos con retorno

```
public int multiplicar(int a, int b) {
        return a * b;
}

public static void main(String[] args) {
        int resultado = multiplicar(4, 5);
        System.out.println(resultado);
}
```

Parámetros en Java

Los elementos esenciales de Java que permiten recibir información en un método son los parámetros. Al llamar a un método, funcionan como recipientes para los datos que se deben proporcionar. La declaración del método define estos parámetros, que especifican el tipo de dato y el nombre que se espera recibir.

El uso de parámetros en los métodos de Java hace que sean más adaptables y reutilizables. Se puede transferir la información relevante al método al establecer parámetros, lo que le permite operar con datos específicos según las necesidades del usuario. Por ejemplo, se establecen dos parámetros de tipo entero para recibir los valores que se desean sumar si tiene un método para calcular la suma de dos números.

Es importante destacar que los parámetros solo son válidos y accesibles dentro del método porque actúan como variables locales. Esto indica que su alcance se limita a la estructura del método que los define. Además, se debe proporcionar argumentos que coincidan en tipo

y cantidad con los parámetros declarados al llamar al método, de lo contrario, se producirá un error.

Este ejemplo básico ilustra cómo funcionan los parámetros definidos en el cuerpo del método:

```
public void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "! ¿Cómo estás?");
}
```

Un parámetro de tipo String llamado "*nombre*" se recibe en el método "*saludar*". Se imprime una cadena de saludo con el valor del parámetro "*nombre*" dentro del cuerpo del método. Se debe proporcionar un argumento de tipo String que se asignará al parámetro "*nombre*" al usar este método.

En resumen, los parámetros son cruciales en Java porque nos permiten recibir datos específicos en los métodos. Haciendo que los métodos sean más dinámicos y adaptables a diversas situaciones a través de ellos.

Parámetros de valor

Los parámetros de valor se utilizan en Java para proporcionar información a un método. Se crea una copia del valor pasado a un método cuando se pasa un parámetro de valor a ese método. Esto significa que la variable original no se verá afectada por cualquier cambio en un parámetro dentro del método.

```
public void duplicar(int numero) {
    numero = numero * 2;
    System.out.println("Número dentro del método: " + numero);
}
```

```
public static void main(String[] args) {
    int numero = 5;
    duplicar(numero);
    System.out.println("Número fuera del método: " + numero);
}
```

En el ejemplo anterior, el valor de la variable número dentro del método duplicar() se duplica; sin embargo, el valor de la variable número fuera del método no se ve afectado por esta modificación. Esto se debe al hecho de que se está utilizando una copia del valor original.

Un ejemplo más complejo de este tipo de método sería este caso, en el que se pide el nombre de usuario en la consola y luego se devuelve por pantalla. Como particularidad, se importa la clase Scanner, que permite tomar los datos que escriba el usuario desde una fuente de entrada.

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // Se llama al método para obtener el nombre del usuario
        String nombreUsuario = obtenerNombreUsuario();

        // Muestra el nombre obtenido
        System.out.println("¡Hola, " + nombreUsuario + "!");
    }

    // Método que obtiene el nombre del usuario
    public static String obtenerNombreUsuario() {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Por favor, ingresa tu nombre: ");
        String nombre = scanner.nextLine();

        // Cerrando el scanner para liberar recursos
        scanner.close();
```

```
return nombre;
    }
}
```

Parámetros de referencia

Los parámetros de referencia se transmiten a un método de referencia en Java. La referencia o la dirección de memoria del objeto se pasan al objeto en sí mismo cuando se pasa un parámetro de referencia a un método. Esto da al método la capacidad de acceder y cambiar el estado del objeto original.

```
public void cambiarNombre(Persona persona, String nuevoNombre) {
    persona.setNombre(nuevoNombre);
    System.out.println("Nombre cambiado: " +
persona.getNombre());
}

public static void main(String[] args) {
    Persona persona = new Persona("Juan");
    cambiarNombre(persona, "Pedro");
    System.out.println("Nuevo nombre: " + persona.getNombre());
}
```

En este ejemplo, el método *cambiarNombre*() recibe un nuevo nombre y una referencia al objeto persona como parámetros. El método cambia el nombre de la persona y luego muestra el cambio. Al imprimir el nombre fuera del método, se muestra el nuevo valor modificado.

Pasar parámetros a un método en Java

En Java, simplemente se deben proporcionar los valores correspondientes al llamar a un método. Los parámetros deben ser del mismo tipo y cantidad que los especificados en la declaración del método.

```
public void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "!");
}

public static void main(String[] args) {
    String nombre = "María";
    saludar(nombre);
}
```

El método *saludar*(), en el ejemplo anterior, recibe un parámetro nombre de tipo String. Se proporciona el valor correspondiente al llamar al método; imprimirá "*Hola, María!*" el método en este caso.

Ejemplos

Se ilustra cómo se utilizan los parámetros en Java con algunos ejemplos prácticos:

- **Ejemplo 1**

```
public void sumar(int a, int b) {
    int suma = a + b;
    System.out.println("La suma de " + a + " y " + b + " es: " +
suma);
}

public static void main(String[] args) {
    int num1 = 10;
int num2 = 5;
    sumar(num1, num2);
}
```

- **Ejemplo 2**

```
public void imprimirDatos(String nombre, int edad) {
        System.out.println("Nombre: " + nombre);
        System.out.println("Edad: " + edad);
}

public static void main(String[] args) {
        String nombre = "Juan";
        int edad = 25;artitmética,artitmética,
imprimirDatos(nombre, edad);
}
```

Atributos en Java

Los atributos en Java son propiedades, también conocidas como campos o variables de instancia, son variables que pertenecen a una clase y representan atributos o propiedades que describen el estado de un objeto. Estos atributos definen las características únicas de cada objeto creado por esta clase.

Cada objeto tiene un conjunto único de atributos que determinan su estado en un momento dado. Estas propiedades pueden contener diferentes tipos de datos, como números, textos, valores booleanos u otros objetos. Los atributos de una clase representan las propiedades que la clase quiere modelar en sus instancias.

Por ejemplo, una clase que representa un automóvil podría tener atributos como modelo, color, velocidad actual y nivel de combustible. Estas características describen las características básicas del automóvil y varían de un automóvil a otro.

Tipos de atributos

En Java, pueden clasificarse los atributos según sus propiedades y comportamientos en varios tipos.

- **Atributos de instancia**: Cada instancia de una clase son específicos. Estos atributos y sus valores pueden variar de un objeto a otro cada objeto tiene su propia copia.

```
public class Persona {
    private String nombre; // Atributo de instancia
    private int edad; // Atributo de instancia
}
```

- **Atributos estáticos:** Son compartidos en una clase por todas las instancias; su valor es el mismo para todas las instancias de esa clase, pertenecen a la clase en lugar de a una instancia particular.

```
public class Contador {
    public static int count; // Atributo estático
}
```

- **Atributos finales**: Después de su inicialización no pueden ser modificados. Pueden ser de instancia o estáticos.

```
public class Constantes {
public final double PI = 3.14; // Atributo final de instancia
    public static final int MAX_VALUE = 100; // Atributo final estático
}
```

- **Atributos de clase**: Son compartidos por todas las instancias de una clase, siendo similares a los atributos estáticos. Sin embargo, se pueden heredar y sobrescribir en subclases.

```
public class Animal {
    protected static String tipo = "Mamífero"; // Atributo de clase
}
```

Son estos los tipos de atributos más frecuentes en Java. Es importante entender sus diferencias para crear clases y objetos efectivos y cómo se utilizan en la programación.

Declarar atributos en Java

Definir atributos en Java, dentro de la clase se utiliza la sintaxis de declaración de variables, pero fuera de cualquier método. A menudo, se definen estos atributos con un modificador de acceso (como public, private o protected) y se les asigna un valor inicial.

```
public class Vehiculo {
    private String modelo;
private String color;
    private double velocidadActual;
    private int cantidadCombustible;
    // Otros atributos y métodos aquí...
}
```

La clase "*Vehiculo*" en este ejemplo, tiene varios atributos (modelo, color, velocidadActual, cantidadCombustible) que describen las propiedades de un vehículo. Ayudan estos atributos a modelar la información y el comportamiento de un vehículo en el programa Java.

Inicializar atributos

Pueden inicializarse los atributos de una clase en Java de varias formas, ya sea dentro de un constructor, en un bloque de inicialización o directamente en la declaración de la variable.

- Inicialización directa en la declaración de la variable

```
public class Persona {
    private String nombre = "John Doe"; // Inicialización directa
    private int edad = 30; // Inicialización directa
}
```

- **Inicialización en el constructor**

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

- **Bloque de inicialización**

```
public class Persona {
    private int edad; {
        // Bloque de inicialización
        edad = 30;
    }
    // Constructor
    public Persona() {
        // ...
    }
}
```

Ejemplos

En el primero de ellos se crea una clase Persona con atributos de instancia nombre y edad, después se crean objetos de esta clase y se inicializan estos atributos.

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

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

	// Método para mostrar información de la persona
	public void mostrarInformacion() {
		System.out.println("Nombre: " + nombre);
		System.out.println("Edad: " + edad);
	}

	public static void main(String[] args) {
		// Crear objetos de tipo Persona
		Persona persona1 = new Persona("Alice", 25);
		Persona persona2 = new Persona("Bob", 30);

		// Mostrar información de las personas
		persona1.mostrarInformacion();
		System.out.println(); // Línea en blanco
		persona2.mostrarInformacion();
	}
}
```

Este otro ejemplo, se crea una clase Contador con un atributo estático count para contar el número de objetos de la clase.

```
public class Contador {
	private static int count; // Atributo estático

// Constructor
	public Contador() {
	count++;
}
```

```
    public static void main(String[] args) {
        // Crear objetos de tipo Contador
        Contador contador1 = new Contador();
        Contador contador2 = new Contador();

        // Mostrar el número de objetos creados
        System.out.println("Número de objetos creados: " +
Contador.count);
    }
}
```

Constructores en Java

Cuando se crea una instancia u objeto de una clase, se llama automáticamente un constructor. Antes de que el objeto esté listo para su uso, su objetivo principal es inicializar los atributos de la clase y realizar cualquier otra inicialización necesaria. Los constructores y las clases tienen el mismo nombre y no tienen tipo de retorno.

Los constructores son esenciales para inicializar y configurar correctamente un objeto. Durante el proceso de creación, pueden recibir parámetros que permiten transferir valores iniciales al objeto.

Tipos de constructores en Java

Java, tiene varios tipos de constructores que se pueden utilizar según tus necesidades.

- **Constructor por defecto**

No toma ningún parámetro. En la clase si no se define ningún constructor, Java proporciona por defecto un constructor automáticamente sin argumentos.

```
public class Persona {
// Constructor por defecto (sin argumentos)
	public Persona() {
		// Inicialización por defecto
	}
}
```

- **Constructor con parámetros**

Este tipo de constructor, toma uno o más parámetros; permitiendo durante la creación del objeto inicializar los atributos de la clase con valores específicos.

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

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

- **Constructor copia**

Crea un nuevo objeto con los mismos valores de atributos que el objeto pasado como argumento, tomados de un objeto del mismo tipo como parámetro.

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

// Constructor copia
	public Persona(Persona otraPersona) {
		this.nombre = otraPersona.nombre;
		this.edad = otraPersona.edad;
	}
}
```

- **Constructor privado**

Generalmente se usa para evitar que se instancie la clase directamente y forzar la creación de objetos a través de métodos de fábrica u otros mecanismos de creación controlada.

```
public class Singleton {
    private static Singleton instance;
    // Constructor privado
    private Singleton() {
        // Inicialización
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Estos son tipos de constructores básicos en Java; para crear constructores más complejos o adaptados a la aplicación; se puede combinar estas técnicas.

Crear constructores en Java

Para en Java crear constructores, los pasos básicos a seguir son:

1. Se decide qué atributos inicializará el constructor.
2. Se define la clase y los atributos
3. Se escribe la firma del constructor (public Persona(String nombre, int edad))
4. Se inicializa los atributos dentro del constructor

Ejemplos

Un ejemplo simple del uso de un constructor:

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

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

La clase Persona del ejemplo, tiene un constructor que acepta dos parámetros: nombre y edad; parámetros que se utilizan para inicializar los atributos nombre y edad de la instancia de la clase.

Usando este constructor se puede crear objetos de la clase Persona de la siguiente manera:

```
Persona persona1 = new Persona("Alice", 25);
Persona persona2 = new Persona("Bob", 30);
```

Se ha creado de esta forma, dos objetos Persona utilizando el constructor con diferentes valores de nombre y edad.

Ventajas de usar atributos y constructores

Definir atributos como privados, puede controlar el acceso a ellos asegurando que solo puedan modificarse a través de métodos específicos, lo que mejora la encapsulación y el control sobre el estado interno del objeto.

La reutilización de código esto permite que los atributos definen las propiedades de un objeto de forma modular y clara.

Con diferentes listas de argumentos se puede tener múltiples constructores. Esto permite la creación de objetos de la misma clase con diferentes formas de inicialización.

Para inicializar atributos los constructores pueden contener lógica avanzada, configurar valores predeterminados, realizar validaciones, o incluso interactuar con otros objetos antes de que el objeto esté completamente inicializado.

Destructores

Un destructor, en contraste con un constructor, destruye un objeto al liberar la memoria de la computadora para que pueda ser utilizada por otra variable u objeto.

Los destructores no existen en Java gracias al recolector de basura de la máquina virtual. El recolector de basura, como su nombre lo indica, recopila todas las variables u objetos que no se utilizan y no se encuentran referenciados por una clase en ejecución, liberando automáticamente la memoria de nuestra computadora.

Aunque Java maneja el recolector de basura de manera automática, el usuario también puede especificar en qué momento Java lo pase mediante la instrucción.

System.gc();

A pesar de que la instrucción anterior es poco o casi nunca utilizada, es importante tener en cuenta que se puede llamar en cualquier momento.

2.6. Upcasting y Downcasting

La conversión de tipos es uno de los conceptos más importantes que básicamente se refiere a la conversión de un tipo de datos en otro, ya sea directa o indirectamente.

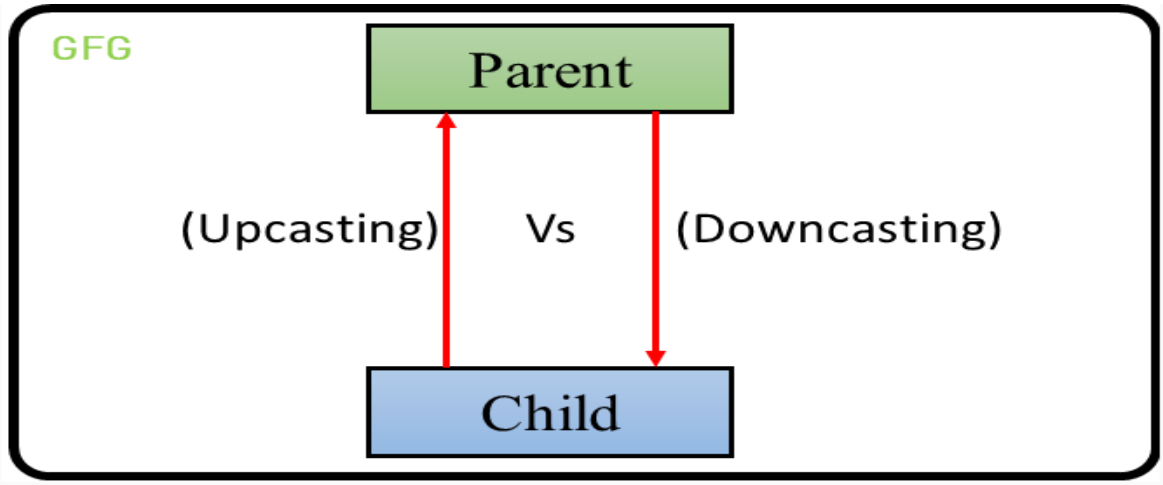
Analiza el concepto de clasificación de objetos. Al igual que los tipos de datos, los objetos también se pueden convertir por tipo. Pero entre los objetos sólo hay dos tipos de objetos: objetos principales y objetos secundarios. Entonces, la conversión de tipo de objeto básicamente significa que un objeto de un tipo es un hijo o padre de otro tipo. Hay dos tipos de conversión, sí:

Upcasting: Convierte objetos secundarios en objetos principales. La transmisión se puede realizar de forma indirecta. Upcasting brinda la flexibilidad de acceder a miembros de una categoría principal, pero el uso de esta función impide el acceso a todos los miembros de una subcategoría. Podemos acceder a ciertos miembros de la subcategoría, pero no a todos los miembros. Por ejemplo, podemos acceder a métodos anulados.

Downcasting: Del mismo modo, downcasting significa clasificar un objeto padre como hijo. La caída no puede ser indirecta.

Figura 9

Diagramación que ilustra los conceptos de transición hacia arriba y hacia abajo



Nota. Adaptado de (geeksforgeeks, 2022)

Clase padre. Un padre puede tener muchos hijos. Consideremos a uno de los niños. Las propiedades del padre se heredan al hijo. Entonces, el niño y el padre tienen una relación "es-un". Por lo tanto, el hijo puede ser el padre implícitamente. Pero un padre puede o no heredar los bienes de su hijo. Sin embargo, podemos hacer que un padre se convierte en un hijo, lo que se conoce como downcasting. Después de definir explícitamente este tipo de conversión, el compilador verifica en segundo plano si es posible. Si esto no es factible, el compilador crea una excepción. ClasCastException

Para comprender la diferencia, aprendamos el siguiente código:

```
Java
// Java program to demonstrate
// Upcasting Vs Downcasting

// Parent class
class Parent {
```

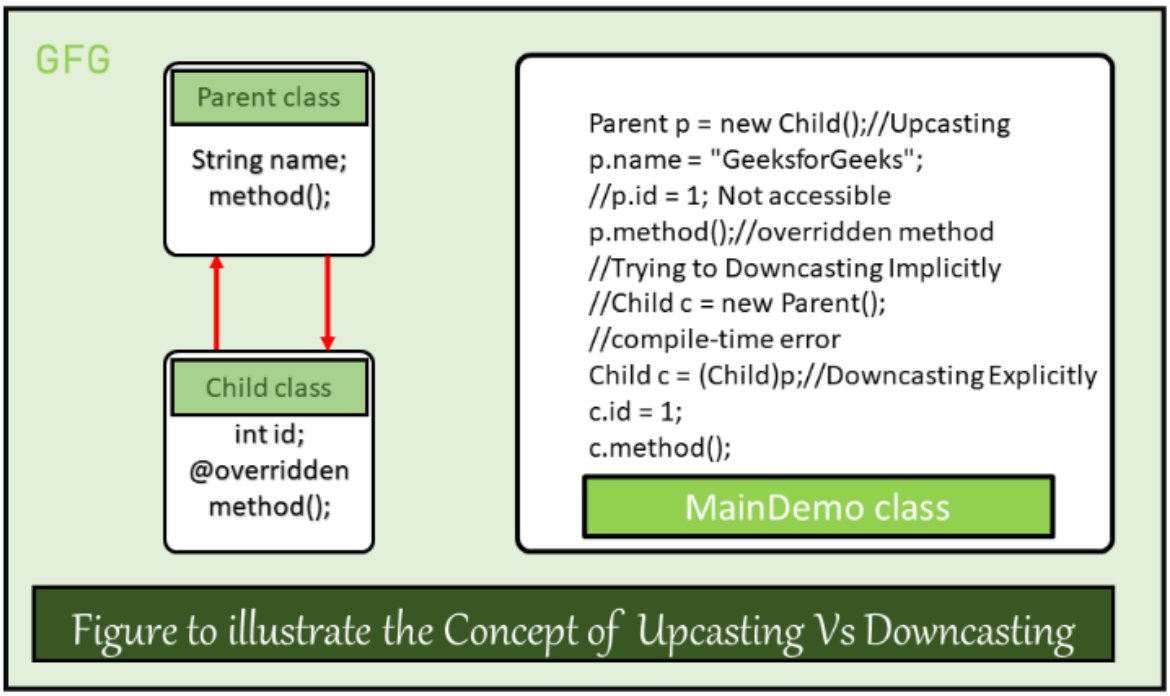
```
	String name;

	// A method which prints the
	// signature of the parent class
	void method(){
		System.out.println("Method from Parent");
	}
}

// Child class
class Child extends Parent {
	int id;

	// Overriding the parent method
 // to print the signature of the
	// child class
	@Override void method(){
		System.out.println("Method from Child");
	}
}

// Demo class to see the difference
// between upcasting and downcasting
public class GFG {

	// Driver code
	public static void main(String[] args){
		// Upcasting
		Parent p = new Child();
		p.name = "GeeksforGeeks";

		//Printing the parentclass name
		System.out.println(p.name);
		//parent class method is overriden method hence this will
be executed
		p.method();

		// Trying to Downcasting Implicitly
		// Child c = new Parent(); - > compile time error

		// Downcasting Explicitly
```

```
            Child c = (Child)p;

            c.id = 1;
            System.out.println(c.name);
            System.out.println(c.id);
            c.method();
        }
}
```

Producción
GeeksforGeeks
Method from Child
GeeksforGeeks
1
Method from Child

Figura 10

Ilustración del programa anterior



Nota Adaptado de (geeksforgeeks, 2022)

Sintaxis de Upcasting

Parent p = new Child();

La actualización se llevará a cabo internamente y, como resultado, el objeto solo puede acceder a los miembros de la clase principal y a los miembros de la clase secundaria especificados (métodos anulados, etc.). No todos los miembros pueden acceder al objeto.

// This variable is not
// accessible
p.id = 1;
Sintaxis de Downcasting:
Child c = (Child)p;

El downcasting debe realizarse desde una fuente externa y permite que un objeto secundario adquiera las propiedades del objeto principal.
c.name = p.name;
i.e., c.name = "GeeksforGeeks"

CAPÍTULO 3

Manejo de errores y excepciones

3

Capítulo

3

Manejo de errores y excepciones

3. Manejo de errores y excepciones

El manejo de errores y excepciones en Java es una parte fundamental de la programación para garantizar que las aplicaciones se comporten de manera predecible y manejen adecuadamente las situaciones imprevistas.

3.1. Definición de una excepción

Una excepción en Java es un evento anómalo o inesperado que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. Cuando se produce una excepción, el programa debe manejarla para evitar fallos inesperados o comportamientos incorrectos.

Definición técnica

Una excepción es un objeto que representa un error o una condición inusual que surge durante la ejecución del programa. En Java, todas las excepciones son instancias de una subclase de

java.lang.Throwable. Las excepciones se pueden generar de manera explícita utilizando la palabra clave throw, o pueden ser lanzadas por la propia máquina virtual Java (JVM) cuando se detecta un problema.

Ejemplo de una excepción común

Si intentas dividir un número por cero, el sistema generará una excepción de tipo ArithmeticException:

```
int resultado = 10 / 0;  // Lanza ArithmeticException: / by zero
```

Jerarquía de excepciones

- **Throwable**: Es la clase base para todos los errores y excepciones en Java.
- **Exception**: Es la clase base para las condiciones que un programa podría querer atrapar. Incluye tanto excepciones verificadas (`Checked exceptions`) como no verificadas (`Unchecked exceptions`).
- **RuntimeException**: Es una subclase de Exception y representa excepciones que pueden evitarse mediante la corrección de errores lógicos en el código.
- **Error**: Representa problemas graves que normalmente no deberían ser manejados por el código del programa, como `OutOfMemoryError`.

1. Tipos de errores en Java

- **Errores de Sintaxis**: son problemas en la estructura del código que el compilador detecta. Por ejemplo, no cerrar un paréntesis o una llave. Estos errores deben corregirse antes de que el código pueda compilarse.

- **Errores de tiempo de ejecución**: ocurren durante la ejecución del programa, como intentar dividir por cero o acceder a un índice de array que no existe.

- **Errores lógicos**: son problemas en la lógica del programa que no son detectados por el compilador, pero hacen que el programa funcione incorrectamente.

2. Excepciones:

- **Checked exceptions**: son excepciones que el compilador fuerza a manejar, ya sea mediante un bloque try-catch o declarando la excepción en la firma del método con throws. Ejemplos incluyen IOException, SQLException.

- **Unchecked exceptions**: también conocidas como excepciones de tiempo de ejecución (RuntimeException y sus subclases). No es obligatorio manejarlas o declararlas. Ejemplos incluyen NullPointerException, ArrayIndexOutOfBoundsException.

- **Errores (Errors)**: son problemas graves que generalmente no deberían ser manejados por el código del programa, como OutOfMemoryError. Indican problemas con la máquina virtual Java (JVM).

3. Manejo de excepciones

- **Bloque try-catch:**
  - El bloque try contiene el código que puede lanzar una excepción.
  - El bloque catch se usa para manejar la excepción. Se puede tener múltiples bloques catch para manejar diferentes tipos de excepciones.

Ejemplo

```
try {
    int division = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error: División por cero.");
}
```

Bloque finally:

- Es opcional y se usa para ejecutar un bloque de código, independientemente de si se lanza o no una excepción. Se utiliza comúnmente para liberar recursos, como cerrar conexiones de base de datos.

Ejemplo

```
try {
    // código que puede lanzar una excepción
} catch (Exception e) {
    // manejo de la excepción
} finally {
    // código que siempre se ejecuta
}
```

Propagación de excepciones con throws:

- Si un método no maneja una excepción, puede declararla con la palabra clave throws para que el método que lo llame se encargue de manejarla.

Ejemplo

```
public void miMetodo() throws IOException {
    // código que puede lanzar IOException
}
```

Creación de excepciones personalizadas:

- Puedes definir tus propias excepciones heredando de Exception o RuntimeException.

Ejemplo

```
public class MiExcepcion extends Exception {
    public MiExcepcion(String mensaje) {
        super(mensaje);
    }
}
```

Buenas prácticas

- Capturar excepciones específicas en lugar de usar catch (Exception e).
- No silenciar excepciones con bloques catch vacíos.
- Liberar recursos en el bloque finally o utilizar el bloque try-with-resources para manejar recursos como archivos o conexiones de bases de datos.
- Crear excepciones personalizadas solo cuando sea necesario y mantener un mensaje descriptivo en ellas.

El manejo adecuado de excepciones es crucial para crear programas robustos y fáciles de mantener.

3.2. Tipos de excepciones

En Java, las excepciones se pueden categorizar principalmente en dos tipos: Checked Exceptions (excepciones verificadas) y Unchecked Exceptions (excepciones no verificadas). Además, hay una categoría

separada llamada Errores. Aquí te explico cada uno de estos tipos:

1. Checked exceptions (Excepciones Verificadas)

Las Checked Exceptions son excepciones que el compilador verifica en tiempo de compilación. Esto significa que el desarrollador está obligado a manejar estas excepciones, ya sea capturándolas en un bloque try-catch o declarando que el método puede lanzar la excepción utilizando la palabra clave throws.

Ejemplos de Checked Exceptions

- **IOException:** Se lanza cuando ocurre un error de entrada/salida, como al intentar leer un archivo que no existe.
- **SQLException:** Se lanza cuando ocurre un error al interactuar con una base de datos.
- **FileNotFoundException:** Se lanza cuando se intenta abrir un archivo que no se encuentra.

Ejemplo:

```
import java.io.*;

public class EjemploCheckedException {
    public static void leerArchivo(String nombreArchivo) throws IOException {
        FileReader archivo = new FileReader(nombreArchivo);
        // Lógica para leer el archivo
    }
```

```
    public static void main(String[] args) {
        try {
            leerArchivo("miArchivo.txt");
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " +
e.getMessage());
        }
    }
}
```

2. Unchecked Exceptions (Excepciones No Verificadas)

Las Unchecked Exceptions son excepciones que no se verifican en tiempo de compilación, es decir, el compilador no obliga al desarrollador a manejarlas. Son subclases de RuntimeException y ocurren generalmente debido a errores lógicos en el código.

Ejemplos de Unchecked Exceptions:

- **NullPointerException**: Se lanza cuando se intenta utilizar un objeto que no ha sido inicializado (es null).
- **ArrayIndexOutOfBoundsException**: Se lanza cuando se intenta acceder a un índice fuera de los límites de un array.
- **ArithmeticException**: Se lanza cuando ocurre un error aritmético, como la división por cero.

Ejemplo:

```
public class EjemploUncheckedException {
    public static void main(String[] args) {
        String texto = null;
        System.out.println(texto.length());          //          Lanza
NullPointerException
    }
}
```

3. Errores (Errors)

Los errores no son excepciones, sino problemas graves que generalmente no deberían ser manejados por el código del programa. Son subclases de Error y representan problemas de los que el programa normalmente no puede recuperarse, como errores del sistema o de la máquina virtual Java (JVM).

Ejemplos de errores:

- **OutOfMemoryError**: Se lanza cuando la JVM se queda sin memoria.

- **StackOverflowError**: Se lanza cuando se desborda la pila, generalmente debido a una recursión infinita.

- **VirtualMachineError**: Se lanza cuando la JVM se encuentra en un estado inestable.

Ejemplo:

```
public class EjemploError {
    public static void metodoRecursivo() {
        metodoRecursivo();  // Lanza StackOverflowError
    }

    public static void main(String[] args) {
        metodoRecursivo();
    }
}
```

- **Checked Exceptions**: Deben ser manejadas o declaradas. Ejemplo: IOException.

- **Unchecked Exceptions**: No es obligatorio manejarlas. Ejemplo: NullPointerException.

- **Errores**: Problemas graves del sistema que generalmente no deben manejarse. Ejemplo: OutOfMemoryError.

3.3. Manejo de excepciones

El manejo de excepciones en Java es una técnica que permite controlar el flujo del programa cuando ocurre una situación inesperada o un error. Este mecanismo es fundamental para crear aplicaciones robustas, ya que permite a los desarrolladores anticipar y gestionar problemas que pueden surgir durante la ejecución del programa. A continuación, te explico cómo se manejan las excepciones en Java:

1. Bloques try-catch

El bloque try-catch es la estructura básica para manejar excepciones. El código que puede generar una excepción se coloca dentro del bloque try, y si ocurre una excepción, el control se transfiere al bloque catch, donde se maneja la excepción.
Estructura básica:

```
try {
    // Código que puede lanzar una excepción
} catch (TipoDeExcepcion e) {
    // Código para manejar la excepción
}
```

Ejemplo:

```
public class EjemploTryCatch {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0;  // Esto lanza ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        }
    }
}
```

2. Múltiples Bloques catch

Es posible manejar diferentes tipos de excepciones con múltiples bloques catch. Cada bloque catch maneja un tipo específico de excepción.

Ejemplo:

```
public class EjemploMultiplesCatch {
    public static void main(String[] args) {
        try {
            int[] numeros = {1, 2, 3};
            System.out.println(numeros[10]);          // Esto lanza
ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Índice fuera de los límites del
array.");
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        }
    }
}
```

3. Bloque finally

El bloque finally se usa para ejecutar un conjunto de instrucciones independientemente de si se lanza o no una excepción. Es útil para liberar recursos como archivos, conexiones de base de datos, etc.

Estructura básica

```
try {
    // Código que puede lanzar una excepción
} catch (TipoDeExcepcion e) {
    // Código para manejar la excepción
} finally {
    // Código que siempre se ejecuta
}
```

Ejemplo:

```
public class EjemploFinally {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error: División por cero.");
        } finally {
            System.out.println("Este bloque se ejecuta siempre.");
        }
    }
}
```

4. Propagación de Excepciones

En algunos casos, es posible que no se desee manejar la excepción en el lugar donde ocurre, sino propagarla para que sea manejada en un nivel superior. Esto se logra utilizando la palabra clave throws en la declaración del método.

Ejemplo:

```
import java.io.*;

public class EjemploPropagacion {
    public static void leerArchivo(String nombreArchivo) throws
IOException {
        FileReader archivo = new FileReader(nombreArchivo);
        // Lógica para leer el archivo
    }
```

```
    public static void main(String[] args) {
        try {
            leerArchivo("miArchivo.txt");
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " +
e.getMessage());
        }
    }
}
```

6. Bloque try-with-resources

Introducido en Java 7, el bloque try-with-resources se utiliza para manejar recursos que deben cerrarse después de su uso, como archivos o conexiones de base de datos. Los recursos que se utilizan dentro del bloque try deben implementar la interfaz AutoCloseable.

Ejemplo:

```
import java.io.*;

public class EjemploTryWithResources {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("miArchivo.txt");
             BufferedReader br = new BufferedReader(fr)) {

            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
```

```
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " +
 e.getMessage());
        }
    }
}
```

El manejo de excepciones en Java permite que tu programa responda de manera controlada y predecible ante errores, manteniendo la robustez y la estabilidad del sistema.

CAPÍTULO 4

Interfaz gráfica

4

Capítulo

4

Interfaz gráfica

4. Interfaz gráfica

4.1. Introducción

Las interfaces en *Java* son una parte crucial de la Programación Orientada a Objetos (POO) que permite definir un contrato que las clases deben cumplir, en esencia define un conjunto de métodos que deben ser implementados por cualquier clase que implemente esa interfaz; es un acuerdo formal que garantiza que una clase tendrá ciertos comportamientos.

Las interfaces gráficas de usuarios explotan la capacidad de las computadoras para reproducir imágenes y gráficos en pantallas y brindan un ambiente amigable, simple e intuitivo, como su nombre lo indica los tres elementos que definen a un GUI son: Interfaz, gráfica, usuario.

4.2. Componentes GUI (botón, checkbox, list, etc.)

Ventanas (marcos, frames): son contenedores de primer nivel, está compuesto por:

- Una barra de título y botones de control de una ventana.
- Un panel de contenido, un contendor de segundo nivel. Su propósito es organizar los componentes (u otros paneles) que contengan.
- Componentes son los elementos con los que interactúa el usuario, entre los cuales se menciona:

Botones: es un elemento interactivo que permite a los usuarios realizar acciones específicas al hacer clic sobre él, son fundamentales en las GUI ya que facilitan la interacción del usuario con el sistema, pueden desencadenar una variedad de procesos, como enviar formularios, iniciar programas, confirmar operaciones o navegar entre diferentes partes de la aplicación.

Etiquetas: es un componente gráficos pasivo que se utiliza para mostrar textos o imágenes al usuario, a diferencia de otros elementos interactivos, las etiquetas no permiten la edición por parte del usuario, su función principal es proporcionar información descriptiva o indicativa sobre otros elementos de la interfaz, como formularios, botones o campos de texto.

Cuadros de texto: es un elemento interactivo que permite a los usuarios ingresar, mostrar o editar texto, son comunes en la GUI y se utilizan para recibir datos de entrada del usuario, como nombres, contraseñas, direcciones y otros tipos de información. Su diseño es simple, generalmente representado como una caja en blanco donde el usuario puede escribir texto.

Casilla de verificación: es un elemento interactivo que permite a los usuarios seleccionar que permite a los usuarios seleccionar una o varias

opciones de un conjunto. Las casillas de verificación, también conocidas como checkbox son representadas como pequeños cuadros que pueden ser marcados (activados) o desmarcados (desactivados) según la selección del usuario; se utilizan comúnmente para habilitar y deshabilitar características, aceptar términos y condiciones o seleccionar múltiples opciones de una lista.

Desplegables: son elementos interactivos que permiten a los usuarios seleccionar una opción de una lista de alternativas, también conocidos como listas desplegables o combos, estos elementos se presentan inicialmente como un campo de selección único que, al ser activado muestra una lista de opciones disponibles, son especialmente útiles para ahorrar espacio en la pantalla y para ofrecer al usuario un conjunto de opciones predefinidas asegurando una selección más controlada y precisa.

Iconos: imagen gráfica que se puede utilizar para representar visualmente elementos de la interfaz de usuario, como botones, menús, ventanas; los iconos mejoran la experiencia del usuario al proporcionar una representación visual de las acciones o el contenido asociado. En Java los íconos se pueden manejar utilizando clases como ImageIcon que forman parte del paquete javax.swing. Para agregar un ícono a un botón en una aplicación Swing, se puede utilizar el siguiente código:

```
JButton button = new JButton("Click Me");
ImageIcon icon = new ImageIcon("ruta/al/icono.png");
button.setIcon(icon);
```

4.3. Creación de la interfaz

La creación de una GUI requiere:

- Diseñar la interfaz de acuerdo a las especificaciones
- Implementar la interfaz usando las facilidades provistas por el lenguaje.

El diseño de una GUI abarca los siguientes aspectos:

- Definir los componentes
- Organizar los componentes estableciendo el diagramado de los componentes contendedoras.
- Decidir cómo reacciona cada componente ante las acciones realizadas por el usuario.

La implementación de una interfaz gráfica consiste en:

- Crear un objeto gráfico para cada componente de la GUI e insertarlo en otros componentes contenedoras.
- Definir el comportamiento de los componentes en respuesta a las acciones de los usuarios.

4.4. Manejo de eventos

Un evento es una señal, externa o interna a la aplicación, que produce la ejecución de un bloque de código que se escribe como un método.

Ejemplos: pulsar el ratón, llamar a un método, modificar el valor de una propiedad.

Un programador debe escribir el código adecuado para responder a cada evento (manejo de eventos)

Componentes clave

1. **EventObject:** todos los eventos en JAVA son objetos que se heredan de la clase EventObject en el paquete java.util

2. Tipos de eventos

ActionEvent: generado por la activación de componentes, como un botón.

AdjustmentEvent: generado por componentes ajustables, como barras de desplazamiento.

WindowEvent: generado por cambios en una ventana, como abrir, cerrar o minimizar.

3. Pasos para manejar eventos

- Registrar el oyente de eventos utilizando métodos como addActionListener

- Implementar la interfaz del oyente, sobrescribiendo métodos para definir la respuesta al evento.

Ejemplo práctico:

```
        JButton button = new JButton("Click Me");
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

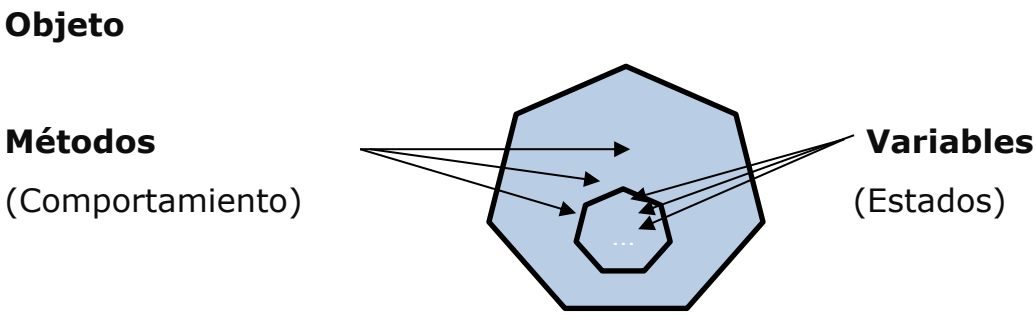
4.5. Manejo de mensajes y excepciones

Manejo de mensajes: un mensaje es una llamada a un método de un objeto, lo que desencadena una acción o el retorno de un valor. El manejo de mensajes es esencial para la comunicación entre objetos, este enfoque se basa en los siguientes conceptos:

1. **Objetos y clases:** los objetos son instancias de clases, que definen su estructura y comportamiento, cada objeto tiene atributos y métodos específicos.

Figura 11

Métodos y clases

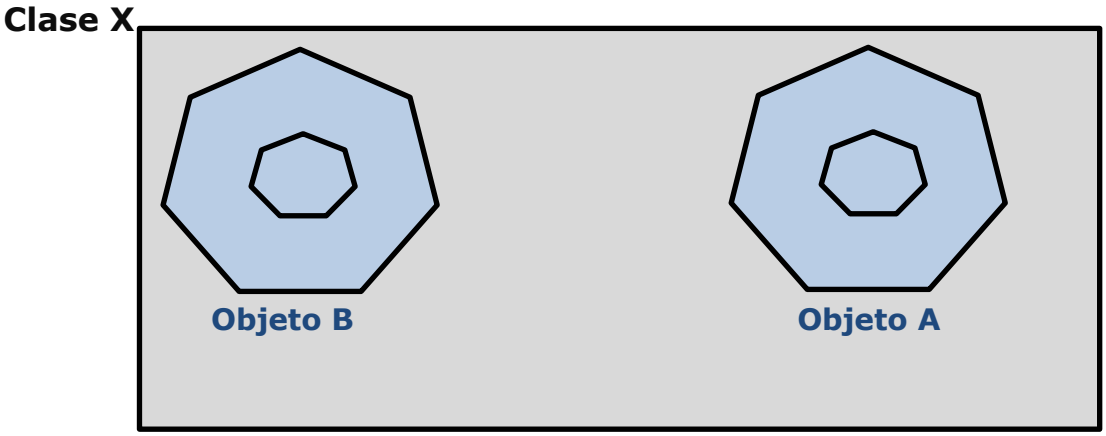


(Leobardo, 2018)

Como se observa todos los objetos tienen una parte pública (comportamiento) y una parte privada (estado).

Figura 12

Clases



(Leobardo, 2018)

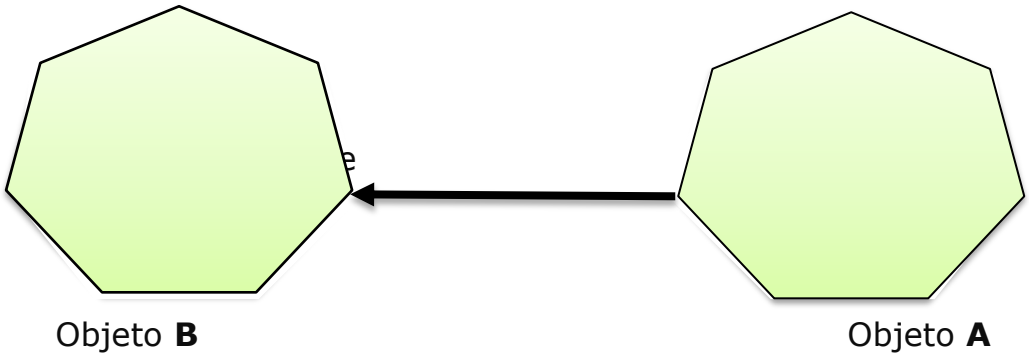
El objeto A y el objeto B son instancia de la clase X. Cada uno de los objetos tiene su propia copia de variables definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos.

2. **Métodos:** son funciones definidas dentro de las clases que operan sobre los datos del objeto, cuando un objeto recibe un mensaje, ejecuta el método correspondiente.

3. **Mensajes:** representan la solicitud de ejecución de un método en un objeto, un objeto envía un mensaje a otro para interactuar con él, lo que permite modularidad y reutilización del código.

Figura 13

Mensajes



(Leobardo, 2018)

Al recibir el mensaje del objeto A, el objeto B ejecutará el método adecuado para el mensaje recibido.

Este enfoque promueve el encapsulamiento, la herencia y el polimorfismo que son los pilares fundamentales de la Programación Orientada a Objetos (POO), lo que facilita la creación de aplicaciones más robustas y mantenibles en Java.

Manejo de excepciones: El manejo de excepciones permite a los programadores manejar errores y eventos inesperados sin terminar abruptamente.

1. **Excepciones:** Son eventos anómalos que ocurren durante la ejecución del programa, como intentos de acceder a un índice fuera de los límites del array.

2. **Manejo de excepciones:**

    - **Try.catch:** Estructura que permite capturar y manejar

excepciones. El bloque try contiene el código que puede generar una excepción, y el bloque catch manejar la excepción si ocurre.

- **Finally:** Bloque opcional que ejecuta el código independientemente si se lanzó una excepción o no.

```
try {
    int[] array = new int[5];
    System.out.println(array[10]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index is out of bounds!");
} finally {
    System.out.println("This block always executes");
}
```

CAPÍTULO 5

Conexión a bases de datos

5

Capítulo

5

Conexión a BD

5. Conexión a BD

5.1. Introducción

¿Qué es una base de datos?

Una base de datos es una colección organizada de datos que permite el almacenamiento, la manipulación y la recuperación de información de manera eficiente. Las bases de datos son esenciales para la gestión de grandes volúmenes de información y se utilizan en diversas aplicaciones, desde sitios web hasta sistemas empresariales.

Tipos de bases de datos

Relacionales: utilizan tablas para representar los datos y sus relaciones. Ejemplos: MySQL, PostgreSQL, Oracle.

NoSQL: diseñadas para datos no estructurados y distribuidos.

Ejemplos: MongoDB, Cassandra, Redis.

Importancia de la conexión a bases de datos

La conexión a bases de datos permite que las aplicaciones interactúen con los datos almacenados, permitiendo realizar operaciones como insertar, actualizar, eliminar y consultar datos (CRUD). Esta interacción es crucial para la funcionalidad de muchas aplicaciones.

5.2. Driver JDBC

¿Qué es JDBC?

Java Database Connectivity (JDBC) es una API de Java que define como un cliente puede acceder a una base de datos. Proporciona métodos para conectar, ejecutar consultas y manejar resultados de una base de datos.

Arquitectura de JDBC

Driver: un componente que permite a la aplicación Java comunicarse con la base de datos.

- **Conexión**: establece un canal de comunicación entre la aplicación y la base de datos.
- **Sentencias (Statements)**: permiten enviar consultas SQL a la base de datos.
- **Resultados (ResultSet)**: almacena los datos recuperados de la base de datos.

Tipos de Drivers JDBC

- **Type 1**: JDBC-ODBC Bridge
- **Type 2**: Native-API Driver
- **Type 3**: Network Protocol Driver
- **Type 4**: Thin Driver (100% Java)

Instalación del driver JDBC

Para conectar una aplicación Java a una base de datos, es necesario instalar el driver JDBC correspondiente. Por ejemplo, para MySQL se utilizará el IDE de programación Netbeans:

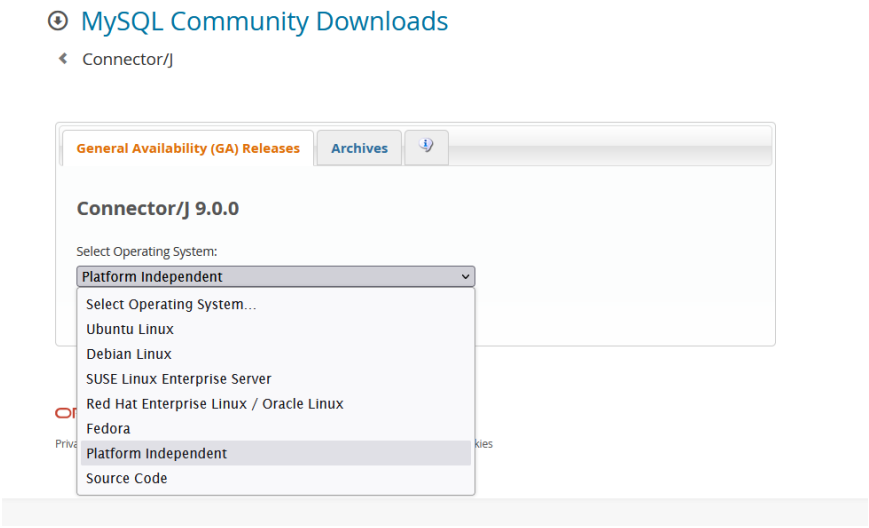
Descarga el driver JDBC desde el https://dev.mysql.com/downloads/connector/j/.
Añade el archivo JAR del driver al classpath del proyecto.

Ejemplo práctico utilizaremos el IDE Netbeans

1.- Descargas el archivo. En la Figura 14 muestra la página de descargas de MySQL Community para el conector Connector/J en su versión 9.0.0.

Figura 14

Página de descarga del conector MySQL

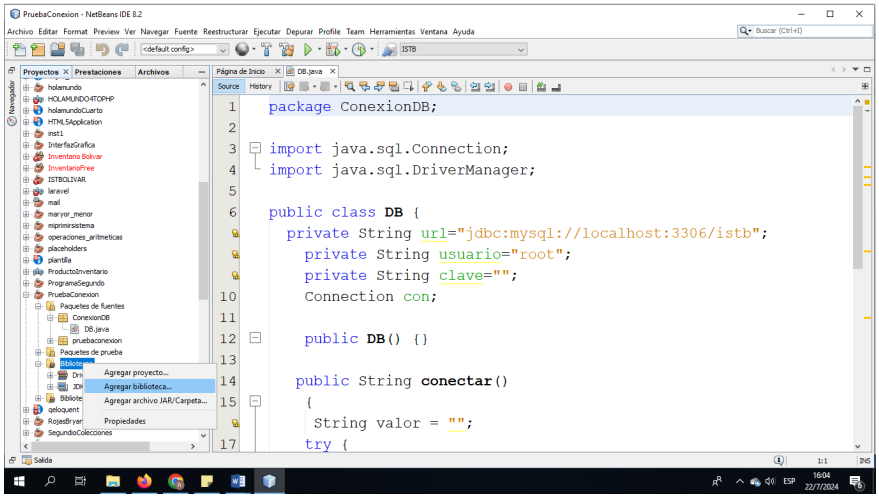


Nota. Imagen adaptada de MySQL, n.d. Recuperado de https://dev.mysql.com/downloads/connector/j/

La Figura 15 ilustra el proceso de configuración y el código necesario para conectar un proyecto Java a una base de datos MySQL utilizando el conector JDBC en NetBeans.

Figura 15

Instalación del conector JDBC en NetBeans

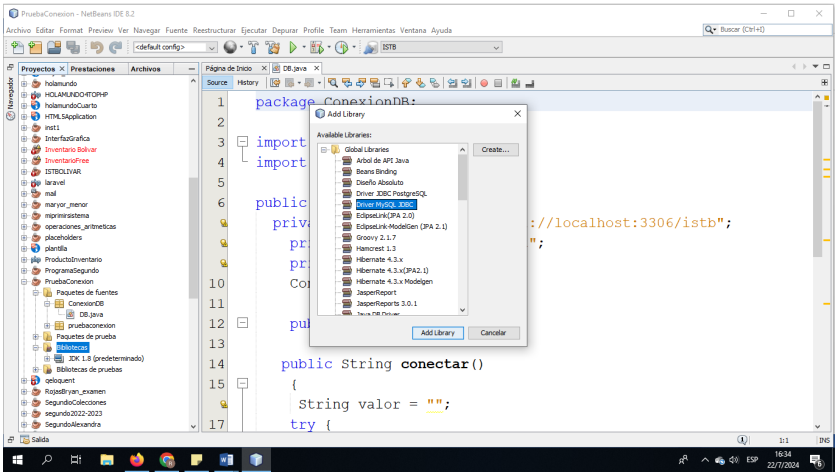


Nota Instalación del conector JDBC en NetBeans.

La Figura 16 muestra una ventana emergente en NetBeans titulada "Add Library" (Agregar Biblioteca). Esta ventana es parte del proceso para agregar una biblioteca a un proyecto. En este caso, el usuario está agregando la biblioteca del conector JDBC para MySQL.

Figura 16

Búsqueda del DRIVER de conexión a MYSQL en NetBeans

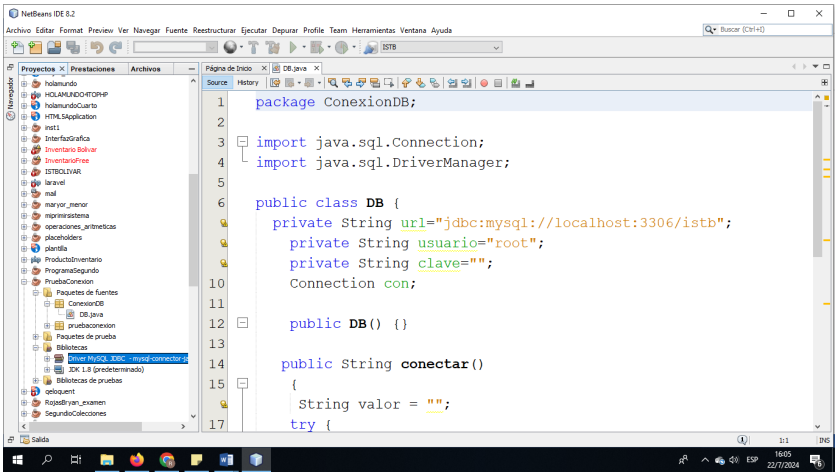


Nota. Elaboración propia

La Figura 17 muestra cómo se ha agregado correctamente la biblioteca del conector JDBC de MySQL al proyecto en NetBeans. Esta biblioteca ahora está lista para ser utilizada en el proyecto para facilitar la conexión y comunicación con una base de datos MySQL.

Figura 17

Verificar si se agregó el driver a las librerías de NetBeans



Nota. Elaboración propia

Configuración del driver JDBC

Para utilizar el driver JDBC, se debe cargar el driver en la aplicación Java. Esto se realiza utilizando la clase Class.forName.

Ejemplo de código:

```
try {
      Class.forName("com.mysql.cj.jdbc.Driver");
    }
catch (ClassNotFoundException e)
   {  e.printStackTrace();  }
```

5.3. Conexión a la BD

Establecer el código de conexión

Para conectar a una base de datos, se utiliza el método getConnection de la clase DriverManager. Este método requiere la URL de la base de datos, el nombre de usuario y la contraseña.

Ejemplo de código:

```
String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
String usuario = "mi_usuario";
String clave = "mi_clave";

Connection conexion = null;

try {
    conexion = DriverManager.getConnection(url, usuario,
clave);
    System.out.println("Conexión exitosa a la base de
datos.");
  }
catch (SQLException e)
{     e.printStackTrace();}
```

Parámetros de conexión

- **URL de conexión**: Especifica el protocolo (jdbc), el subprotocolo (mysql), la ubicación del servidor (localhost) y el nombre de la base de datos (mi_base_de_datos).

- **Nombre de usuario y contraseña**: Credenciales necesarias para autenticarse en la base de datos.

Manejo de excepciones

El proceso de conexión puede generar excepciones como SQLException, que deben ser manejadas adecuadamente para evitar fallos en la aplicación.

Sí, el API JDBC (Java Database Connectivity) soporta dos modelos diferentes de acceso a bases de datos: el modelo de dos capas y el modelo de tres capas. A continuación, se describen ambos modelos:

Modelo de dos capas (Two-Tier Architecture)

En el modelo de dos capas, también conocido como cliente-servidor, una aplicación cliente se comunica directamente con la base de datos. Este modelo es más simple y se utiliza generalmente en aplicaciones pequeñas o de escritorio.

Características:

1. **Cliente-Servidor Directo**: El cliente se conecta directamente al servidor de la base de datos.
2. **Menos Complejidad**: Menos componentes involucrados, lo que reduce la complejidad de la arquitectura.
3. **Rápida Implementación**: Ideal para aplicaciones pequeñas donde la implementación rápida es crucial.
4. **Escalabilidad Limitada**: No es adecuado para aplicaciones que requieren alta escalabilidad.

Ejemplo de conexión en modelo de dos capas:

```
String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
String usuario = "mi_usuario";
String clave = "mi_clave";

try (Connection conexion = DriverManager.getConnection(url,
usuario, clave)) {
       System.out.println("Conexión exitosa a la base de
datos.");
    // Realizar operaciones CRUD
} catch (SQLException e)
     {  e.printStackTrace();}
```

Ejemplo Práctico

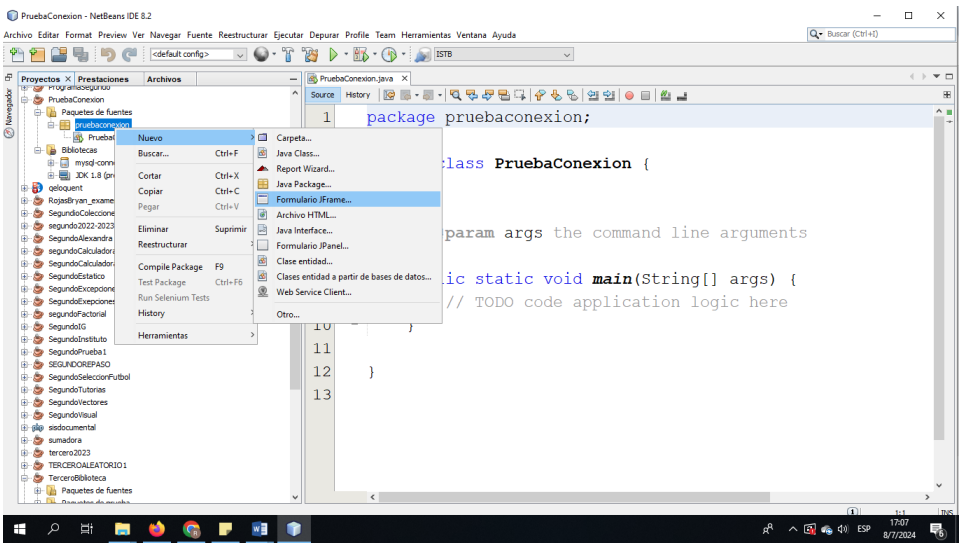
Crear la clase BD.java
Dentro del paquete ConexionBD
Código

```
package ConexionDB;

 import java.sql.Connection;
 import java.sql.DriverManager;

public class DB {
   private String url="jdbc:mysql://localhost:3306/istb";
   private String usuario="root";
   private String clave="";
   Connection con;

   public DB() {    }

  public String conectar()
   {
     String valor = "";
      try {
             Class.forName("com.mysql.jdbc.Driver");
             con = DriverManager.getConnection(url, usuario, clave);
            valor = "Conexión exitosa";
           }
      catch (ClassNotFoundException e)
          {valor = "Error: Driver JDBC no encontrado - " + e.getMessage();}
      catch (Exception e)
         {valor = "Error general: " + e.getMessage();}
     return valor;
   }
}
```

En la Figura 18 muestra en el menú contextual (clic derecho) sobre el paquete pruebaconexion, se ha seleccionado la opción "Nuevo" y luego "JFrame Form...", lo que sugiere que se está a punto de crear una nueva interfaz gráfica de usuario (GUI) en el proyecto, específicamente un formulario de tipo JFrame.

Figura 18

Crear formulario en NetBeans

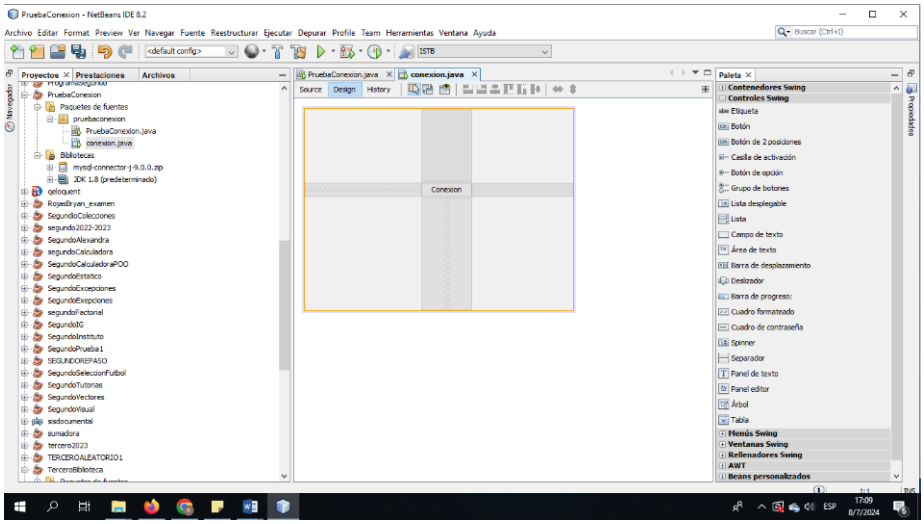


Nota. Elaboración propia

En la Figura 19 muestra en la vista de diseño (Design), se está visualizando un formulario gráfico que incluye un botón etiquetado como "Conexión".

Figura 19

Colocar un botón para probar la conexión.

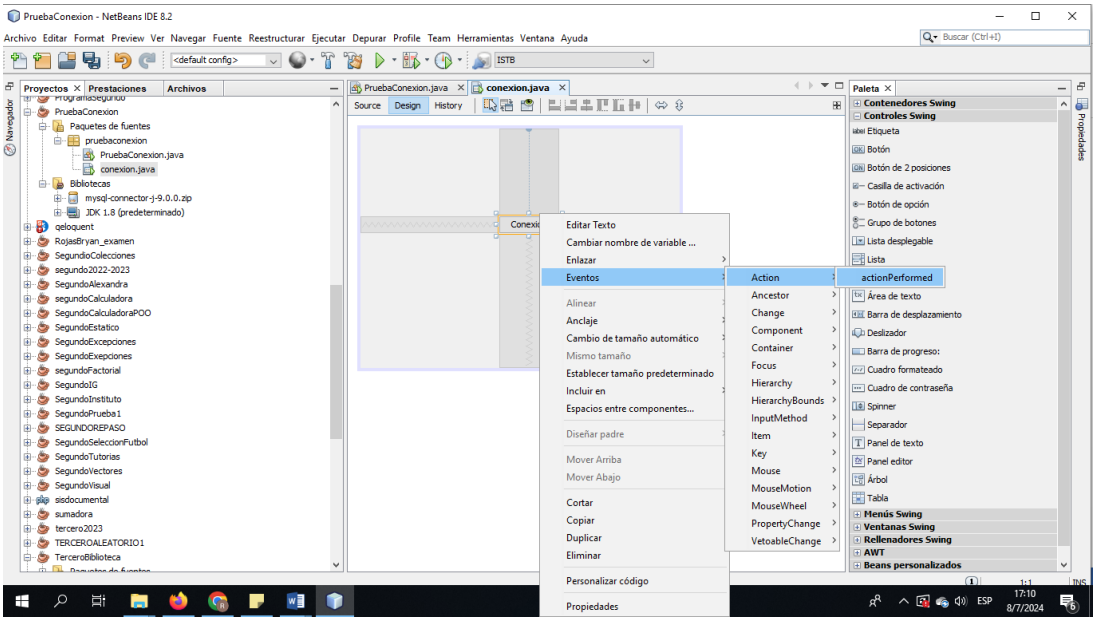


Nota. Elaboración propia

En la Figura 20 enfocándose en la ventana de diseño (Design) de un formulario de tipo JFrame en el archivo conexion.java. Se ha hecho clic derecho sobre un botón etiquetado como "Conexión", abriendo un menú contextual. Dentro de este menú, se está navegando hacia la sección de "Eventos" y luego a "Action", lo que sugiere que se está configurando un evento de acción para el botón.

Figura 20

Crear un evento del botón (Actionperformed)



Nota. Elaboración propia

En el formulario dentro de la acción colocamos el siguiente código

```
private void btnConexionActionPerformed(java.awt.event.ActionEvent evt)
    {
      String a = bd.conectar();
      JOptionPane.showMessageDialog(this, a, "Mensaje",
      JOptionPane.ERROR_MESSAGE);
    }
```

Para evitar errores de compilación

Al inicio del código del formulario declaramos las variables a utilizar (En la clase)

```
ConexionDB.DB bd= new DB();
Statement st;
ResultSet rs;
Connection conn;
```

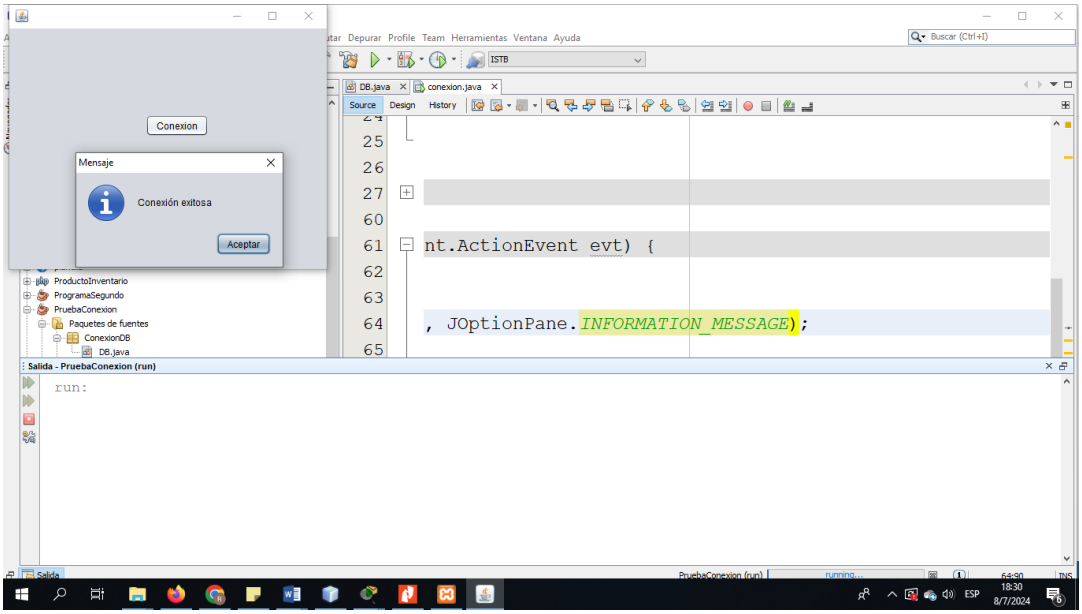
Para que no de error importamos las siguientes librerías

```
import ConexionDB.DB;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.swing.JOptionPane;
```

En la Figura 21 muestra una ventana emergente en donde se muestra el resultado de la ejecución del evento actionperformend del botón conexión, en el cual indica una afirmación de la conexión

Figura 21

Resultado de la prueba de conexión en NetBeans



Nota. Elaboración propia

Modelo de tres capas (Three-Tier Architecture)

En el modelo de tres capas, hay una capa intermedia entre el cliente y la base de datos. Esta capa intermedia se llama lógica de negocio o capa de aplicación, y maneja la comunicación entre el cliente y el servidor de la base de datos.

Características

1. **Capa Intermedia**: introduce una capa de aplicación que maneja la lógica de negocio y la comunicación con la base de datos.
2. **Separación de Preocupaciones**: mejora la modularidad y separación de preocupaciones.
3. **Escalabilidad Mejorada**: adecuado para aplicaciones que necesitan escalar y manejar múltiples clientes.
4. **Mayor Complejidad**: más componentes y puntos de fallo potenciales.

Ejemplo de arquitectura en modelo de tres capas

- **Capa Cliente**: interfaz de usuario que interactúa con la capa de aplicación.
- **Capa de Aplicación**: contiene la lógica de negocio y se comunica con la base de datos.
- **Capa de Base de Datos**: servidor de base de datos que almacena y gestiona los datos.

Para organizar el código en archivos de manera coherente con una estructura de capas en Java, puedes crear un archivo por cada clase. La estructura de carpetas y archivos se vería algo así:

Ejemplo de código simplificado:

proyecto/
│
├── src/
│   ├── Cliente.java
│   ├── Servicio.java
│   └── BaseDatos.java
├── lib/ (Aquí irían las dependencias, por ejemplo, el conector JDBC de MySQL)
├── build/ (Carpeta para archivos compilados)
└── README.md (Descripción del proyecto)

1. **Capa Cliente**:

```
public class Cliente {
    public static void main(String[] args) {
       Servicio servicio = new Servicio();
       servicio.realizarOperacion();
    }
}
```

2. **Capa de Aplicación (Servicio)**:

```
public class Servicio {
    private BaseDatos bd = new BaseDatos();

    public void realizarOperacion() {
        bd.conectar();
       // Lógica de negocio aquí se realiza el código de
       cualquier proceso
        bd.desconectar();
    }
}
```

3. **Capa de Base de Datos (BaseDatos)**:

```
public class BaseDatos {
    private static final String URL =
"jdbc:mysql://localhost:3306/mi_base_de_datos";
    private static final String usuario = "mi_usuario";
    private static final String clave = "mi_clave";
    private Connection conexion;

    public void conectar() {
     try {
```

```
            conexion = DriverManager.getConnection(URL,
usuario, clave);
            System.out.println("Conexión exitosa a la base de
datos.");
        }
     catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void desconectar() {
        if (conexion != null) {
            try {
                conexion.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    // Métodos CRUD
}
```

La Tabla 2 compara dos modelos de arquitectura de software, el Modelo de Dos Capas y el Modelo de Tres Capas, en función de varias características clave:

Tabla 2.

Comparación entre modelos

Característica	Modelo de Dos Capas	Modelo de Tres Capas
Arquitectura	Cliente-Servidor Directo	Cliente - Aplicación - Base de Datos
Complejidad	Baja	Alta
Escalabilidad	Limitada	Alta
Mantenimiento	Simple	Más complejo
Uso	Aplicaciones pequeñas y de escritorio	Aplicaciones empresariales y web

Ambos modelos tienen sus ventajas y desventajas, y la elección entre ellos depende de los requisitos específicos de la aplicación, como la complejidad, la escalabilidad y el mantenimiento.

5.4. CRUD (Operaciones Básicas)

En JDBC, hay tres tipos principales de objetos Statement que se utilizan para enviar comandos SQL a la base de datos: Statement, PreparedStatement, y CallableStatement. Cada uno tiene su propio propósito y características específicas.

1. Statement

Statement se utiliza para ejecutar sentencias SQL estáticas, es decir, aquellas que no cambian en tiempo de ejecución. (Revisar formato de fuente)

Características:

- **Simplicidad**: es fácil de usar para consultas simples y estáticas.
- **Inyección de SQL**: vulnerable a ataques de inyección de SQL si los datos de entrada del usuario no se manejan adecuadamente.
- **Rendimiento**: menos eficiente para consultas repetitivas, ya que cada vez que se ejecuta una consulta, se vuelve a compilar.

Ejemplo de uso:

```
Statement stmt = conexion.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM usuarios");

while (rs.next()) {
    System.out.println("Nombre: " + rs.getString("nombre"));
    }
```

2. PreparedStatement

PreparedStatement se utiliza para ejecutar sentencias SQL parametrizadas. Es adecuado para consultas que se ejecutan con frecuencia con diferentes parámetros.

Características

- **Seguridad**: protege contra ataques de inyección de SQL, ya que los parámetros se envían por separado y no se concatenan en la consulta SQL.
- **Rendimiento**: más eficiente para consultas repetitivas, ya que la consulta SQL se compila una vez y se puede ejecutar múltiples veces con diferentes parámetros.
- **Parámetros**: soporta parámetros que se pueden cambiar en tiempo de ejecución.

Ejemplo de uso:

```
String sql = "SELECT * FROM usuarios WHERE nombre = ?";
PreparedStatement ps = conexion.prepareStatement(sql);
ps.setString(1, "Juan");
ResultSet rs = ps.executeQuery();

while (rs.next()) {
    System.out.println("Correo: " + rs.getString("correo"));
}
```

3. `CallableStatement`

`CallableStatement` se utiliza para ejecutar procedimientos almacenados en la base de datos.

Características:

- **Procedimientos Almacenados**: permite llamar a procedimientos almacenados que encapsulan la lógica del negocio en la base de datos.

- **Parámetros de Entrada y Salida**: soporta tanto parámetros de entrada como de salida.
- **Seguridad y Rendimiento**: similar a `PreparedStatement` en términos de seguridad y rendimiento.

Ejemplo de uso:

```
String sql = "{call obtenerUsuarioPorNombre(?)}";
CallableStatement cs = conexion.prepareCall(sql);
cs.setString(1, "Juan");
ResultSet rs = cs.executeQuery();

while (rs.next()) {
    System.out.println("Correo: " + rs.getString("correo"));
}
```

Comparación rápida

La Tabla 3 compara tres tipos de declaraciones en JDBC (Java Database Connectivity): Statement, PreparedStatement, y CallableStatement, en función de varias características clave:

Tabla 3

Declaraciones en JDBC

Característica	Statement	PreparedStatement	CallableStatement
Tipo de Consulta	Estáticas	Parametrizadas	Procedimientos almacenados
Vulnerabilidad a Inyección	Alta	Baja	Baja
Rendimiento	Menor eficiencia para consultas repetitivas	Mayor eficiencia para consultas repetitivas	Mayor eficiencia para procedimientos almacenados
Uso de Parámetros	No	Sí	Sí
Ejemplo de Uso	Consultas simples	Consultas frecuentes con parámetros	Llamada a procedimientos almacenados

Cuando usar cada uno

- **Statement**: úsalo para consultas SQL simples y únicas donde la seguridad y el rendimiento no son una preocupación crítica.
- **PreparedStatement**: úsalo para consultas que se ejecutan con frecuencia con diferentes parámetros y cuando necesitas protección contra inyección de SQL.
- **CallableStatement**: úsalo cuando necesites ejecutar procedimientos almacenados que encapsulan lógica compleja del negocio en la base de datos.

Cada tipo de Statement tiene su propio conjunto de beneficios y es importante elegir el adecuado según el caso de uso específico para asegurar la eficiencia, la seguridad y la facilidad de mantenimiento de la aplicación.

CRUD

El CRUD (Create, Read, Update, Delete) es un conjunto de operaciones básicas que se utilizan en la gestión de bases de datos relacionales y sistemas de información. Estas operaciones permiten crear, leer, actualizar y eliminar datos de una base de datos. Aquí te detallo cada una de estas operaciones:

Crear (Create)

La operación de creación inserta nuevos registros en la base de datos. Se utiliza la sentencia SQL INSERT INTO.

Ejemplo de código:

```
String query = "INSERT INTO usuarios (nombre, correo) VALUES
(?,?)";
PreparedStatement ps = conexion.prepareStatement(query);
ps.setString(1, "Juan");
ps.setString(2, "juan@example.com");
ps.executeUpdate();
```

Leer (Read)

La operación de lectura recupera datos de la base de datos. Se utiliza la sentencia SQL SELECT.

Ejemplo de código:

```
String query = "SELECT * FROM usuarios";
Statement st = conexion.createStatement();
ResultSet rs = st.executeQuery(query);

while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id"));
    System.out.println("Nombre: " + rs.getString("nombre"));
    System.out.println("Correo: " + rs.getString("correo"));
}
```

Actualizar (Update)

La operación de actualización modifica registros existentes en la base de datos. Se utiliza la sentencia SQL UPDATE.

Ejemplo de código:

```
String query = "UPDATE usuarios SET correo = ? WHERE nombre
= ?";
PreparedStatement ps = conexion.prepareStatement(query);
ps.setString(1, "nuevo_correo@example.com");
ps.setString(2, "Juan");
ps.executeUpdate();
```

Eliminar (Delete)

La operación de eliminación borra registros de la base de datos. Se utiliza la sentencia SQL DELETE.

Ejemplo de código:

```
String query = "DELETE FROM usuarios WHERE nombre = ?";
PreparedStatement ps = conexion.prepareStatement(query);
ps.setString(1, "Juan");
ps.executeUpdate();
```

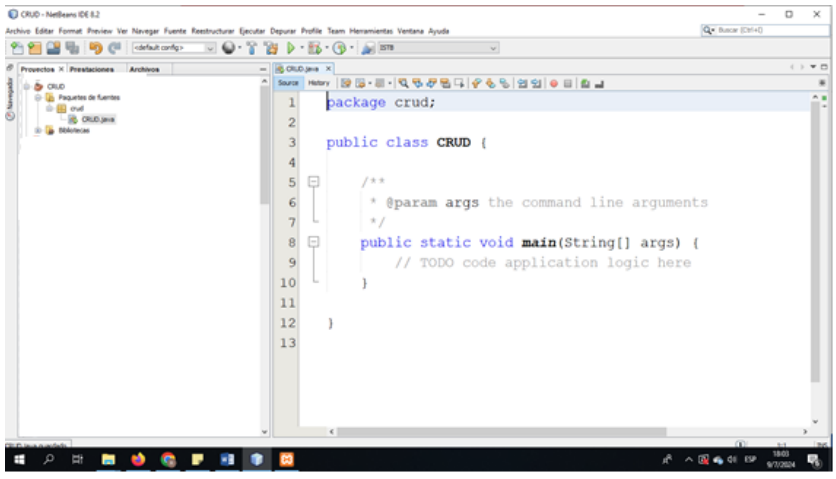
Ejemplo completo de CRUD

Aquí tienes un ejemplo completo que muestra cómo realizar todas las operaciones CRUD en una base de datos.

En la Figura 22 se muestra la creación de un proyecto en java el cual consta de la carpeta principal, el paquete y la clase.

Figura 22

Crear paquete Conexión en NetBeans

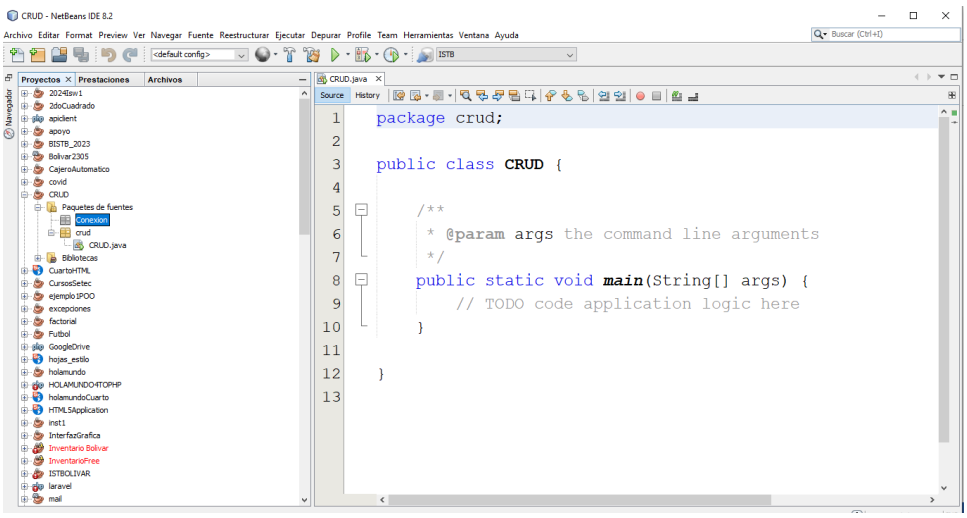


Nota. Elaboración propia

En la Figura 23 se muestra la creación de un paquete adicional para la clase conexión

Figura 23

Crear una Clase llamada BD.java en NetBeans

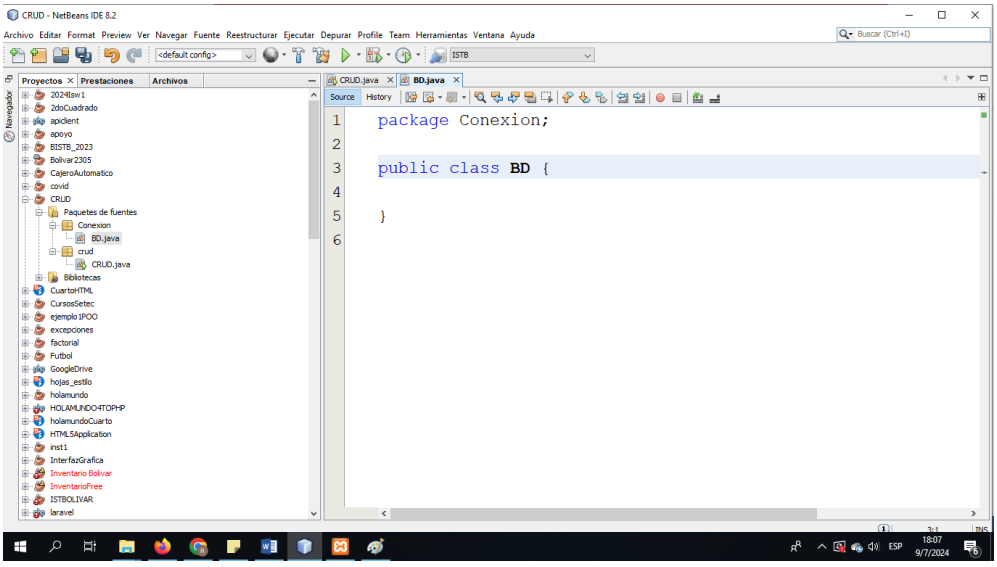


Nota. Elaboración propia

En la Figura 24 se muestra la creación de una clase dentro del paquete recién creado llamada BD.java

Figura 24

Escritorio de desarrollo dentro de la clase en NetBeans



Nota. Elaboración propia

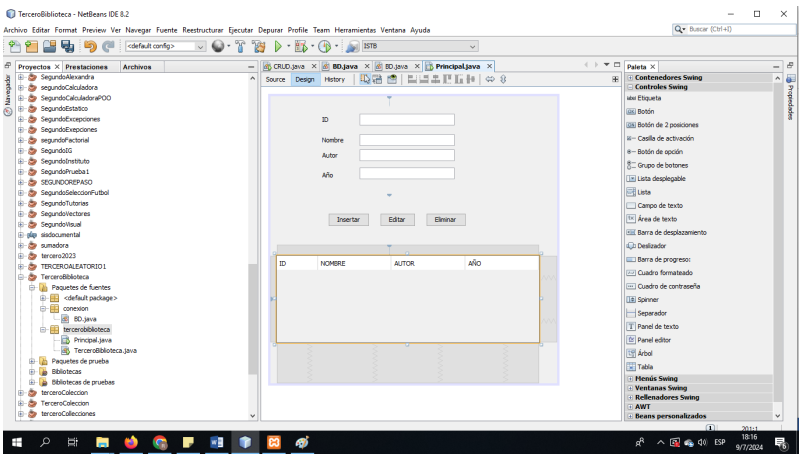
En la clase BD.java colocamos el código

```
package conexion;
import java.sql.Connection;
import java.sql.DriverManager;

public class BD {
    private String url="jdbc:mysql://localhost:3306/biblioteca";
    private String usuario="root";
    private String clave="";
    Connection con;
        public Connection conectar()
    {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            con=DriverManager.getConnection(url, usuario, clave);
        } catch (Exception e) {
        }
    return con;
    }
}
```

En la Figura 25 se muestra un formulario que contiene campos de texto para ingresar "ID", "Nombre", "Autor", y "Año". Debajo de estos campos, hay botones etiquetados como "Insertar", "Editar", y "Eliminar". Debajo de los botones, se encuentra una tabla con las columnas "NOMBRE", "AUTOR", y "AÑO", lo que sugiere que está diseñada para mostrar una lista de libros o registros relacionados con la biblioteca.

Figura 25

Crear componentes visuales en el formulario pestaña Desing

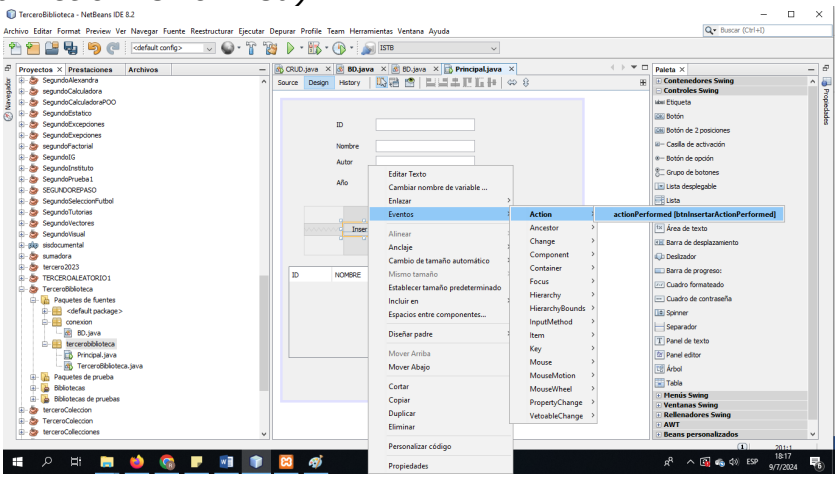


Nota. Elaboración propia

En la Figura 26 se indica que se ha hecho clic derecho sobre el botón "Insertar", lo cual ha desplegado un menú contextual. Este menú muestra varias opciones para editar propiedades y acciones del botón. Se ha seleccionado la opción "Eventos", y dentro de esta, se está navegando hacia la opción "Action" y su subopción "actionPerformed", que es un evento comúnmente utilizado en botones para capturar la acción cuando el usuario hace clic en el botón.

Figura 26

Crear botón Insertar e implantar el evento (btnInsertarActionPerformed)



Nota. Elaboración propia

Dentro del ActionPerformend colocamos los métodos agregar() y mostrar()

Nota en el encabezado debemos importar las siguientes librerías

```
Adicional import java.sql.Connection;
import Conexion.BD;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
```

declarar las variables e instanciar los objetos a utilizar al inicio de la clase

```
        BD con=new BD();
        Connection cn;
        Statement st;
        PreparedStatement ps;
        ResultSet rs;
        DefaultTableModel modelo;

private void btnInsertarActionPerformed(java.awt.event.ActionEvent evt) {
      agregar();
      mostrar();
   }
```

Desarrollo del método agregar

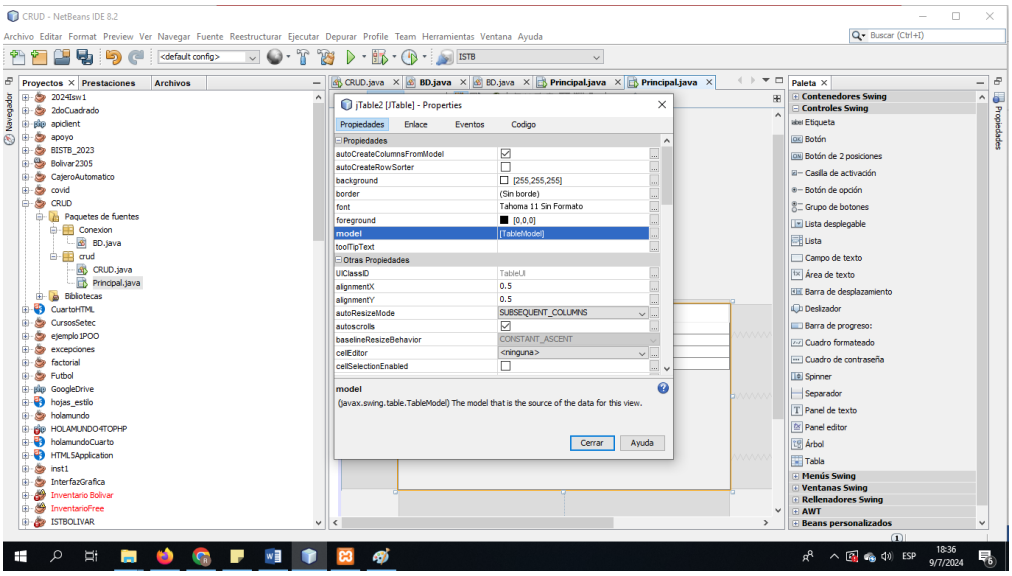
```
public void agregar()
   {
    String nombre,autor,anio;
    nombre=txtNombre.getText();
autor=txtAutor.getText();anio=txtAnio.getText();
    String sql = "INSERT INTO libro (nombre, autor, anio_edicion) VALUES (?,
?, ?)";
       try {
          cn=con.conectar();
          ps = cn.prepareStatement(sql);
ps.setString(1, nombre);
ps.setString(2, autor);
ps.setString(3, anio);
          ps.executeUpdate(sql);
          limpiar();
          JOptionPane.showMessageDialog(this, "Libro Ingresado",
"Información",
          JOptionPane.INFORMATION_MESSAGE);
       } catch (Exception e) {
```

```
            JOptionPane.showMessageDialog(this, "error"+ e.getMessage(),
"Información",
            JOptionPane.INFORMATION_MESSAGE);
        }
    }
```

Esta la Figura 27 se muestra cómo se están ajustando las configuraciones de una tabla en la interfaz gráfica para gestionar cómo los datos se muestran y se comportan dentro de una aplicación Java. La configuración del modelo de la tabla es crucial para enlazar los datos que se desean mostrar en la tabla con la interfaz visual.

Figura 27

Modificar campo model

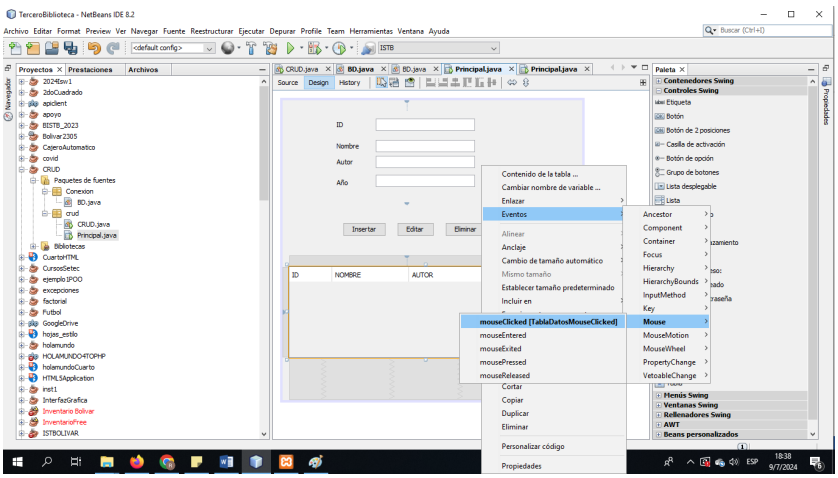


Nota. Elaboración propia

La Figura 28 ilustra el proceso de ajuste de las configuraciones de una tabla en la interfaz gráfica para habilitar el evento TablaDatosMouseClicked. Este evento permite seleccionar datos de la tabla, colocarlos en las cajas de texto correspondientes y, de esta manera, editar o eliminar los registros seleccionados.

Figura 28

Habilitar el evento (TablaDatosMouseClicked)



Nota. Elaboración propia

Se creará un método ahí colocamos el siguiente código

```
private void TablaDatosMouseClicked(java.awt.event.MouseEvent evt) {
        int fila=TablaDatos.getSelectedRow();
        if(fila==-1)
        {
            JOptionPane.showMessageDialog(this, "no tiene seleccion", "Titulo",
JOptionPane.ERROR_MESSAGE);
        }
        else
        {
          String id, a,b,c;
          id=TablaDatos.getValueAt(fila, 0).toString();
          a=TablaDatos.getValueAt(fila, 1).toString();
          b=TablaDatos.getValueAt(fila, 2).toString();
          c=TablaDatos.getValueAt(fila, 3).toString();
          txtid.setText(id);
          txtNombre.setText(a);
          txtAutor.setText(b);
          txtAnio.setText(c);
        }
    }
```

Método mostrar

```
public void mostrar()
    {
      String sql="SELECT * FROM libro";
        try {
```

```
            cn=con.conectar();
            st=cn.createStatement();
            rs=st.executeQuery(sql);
            Object[] libro1=new Object[4];
            modelo=(DefaultTableModel) TablaDatos.getModel();
            while(rs.next())
            {
             libro1[0]=rs.getInt("id");
             libro1[1]=rs.getString("nombre");
             libro1[2]=rs.getString("autor");
             libro1[3]=rs.getString("anio_edicion");
            modelo.addRow(libro1);
            }
            TablaDatos.setModel(modelo);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, e, "titulo",
JOptionPane.ERROR_MESSAGE);
        }
    }
```

Implementamos el método limpiar para que no se repitan los datos

```
public  void limpiar()
    {
      for(int i=0;i<=TablaDatos.getRowCount();i++)
      {
      modelo.removeRow(i);
      i=i-1;
      }
    }
```

Actualizar

En primer lugar debemos seleccionar la fila o el registro que se muestra en la tabla al dar clic los valores se mostraran en las cajas de texto que se encuentran en la parte superior del formulario modificamos el texto y pulsamos en El botón Editar.

Se ejecutará el evento

```
private void btnEditarActionPerformed(java.awt.event.ActionEvent evt) {
        int id;
        String nombre, autor,anio;
        id= Integer.parseInt(txtid.getText());
        nombre=txtNombre.getText();
        autor=txtAutor.getText();
        anio=txtAnio.getText();
        String sql = "UPDATE libro SET nombre = ?, autor = ?, anio_edicion = ?
WHERE
         id = ?";
        try {
            cn=con.conectar();
            ps=cn. cn.prepareStatement(sql);
ps.setString(1, nombre);
ps.setString(2, autor);
ps.setString(3, anio);
ps.setInt(4, id);
            int r=ps.executeUpdate(sql);
            limpiar();
            if(r>0)
            {
                JOptionPane.showMessageDialog(this, "Libro Actualizado",
"Información",
                JOptionPane.INFORMATION_MESSAGE);
            }
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "error"+e, "Información",
            JOptionPane.INFORMATION_MESSAGE);
        }
        mostrar();
    }
```

Eliminar

```
private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
    int id;
    id = Integer.parseInt(txtid.getText());
    String sql = "DELETE FROM libro WHERE id = ?";
    try {
        cn = con.conectar();
        PreparedStatement ps = cn.prepareStatement(sql);
        ps.setInt(1, id);
        int r = ps.executeUpdate();
        limpiar();
        if (r > 0) {
```

```
            JOptionPane.showMessageDialog(this, "Libro eliminado",
"Información",
            JOptionPane.INFORMATION_MESSAGE);
        }
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this, "Error: " + e.getMessage(),
"Información",
        JOptionPane.INFORMATION_MESSAGE);
    }
    mostrar();
}
```

```
public class CRUD_Ejemplo {
 private static final String URL =
"jdbc:mysql://localhost:3306/mi_base_de_datos";
 private static final String USER = "mi_usuario";
 private static final String PASSWORD = "mi_contraseña";
Connection conexion;

    public static void main(String[] args) {
     try (conexion = DriverManager.getConnection(URL, USER, PASSWORD))
       {
     // Crear
     String insertQuery = "INSERT INTO usuarios (nombre, correo)
                  VALUES (?,?)";
     try (PreparedStatement ps =
          conexion.prepareStatement(insertQuery)) {
          ps.setString(1, "Juan");
          ps.setString(2, "juan@example.com");
          ps.executeUpdate();
         }

     // Leer
     String selectQuery = "SELECT * FROM usuarios";
     try (Statement st = conexion.createStatement();
     ResultSet rs = st.executeQuery(selectQuery)) {
     while (rs.next()) {
            System.out.println("ID: " + rs.getInt("id"));
            System.out.println("Nombre: " + rs.getString("nombre"));
            System.out.println("Correo: " + rs.getString("correo"));
           }
        }

    // Actualizar
    String sql= "UPDATE usuarios SET correo =? WHERE nombre=?";
   try (PreparedStatement ps = conexion.prepareStatement(sql))
       {
        ps.setString(1, "nuevo_correo@example.com");
        ps.setString(2, "Juan");
        ps.executeUpdate();
```

```
        }

        // Eliminar
        String deleteQuery = "DELETE FROM usuarios WHERE nombre = ?";
        try (PreparedStatement ps =
             conexion.prepareStatement(deleteQuery)) {
                 ps.setString(1, "Juan");
                 ps.executeUpdate();
             }
         } catch (SQLException e) {
             e.printStackTrace();
         }
    }
}
```

Cómo realizar la conexión a diferentes tipos de bases de datos (SQL Server, PostgreSQL, Oracle, etc) desde una aplicación Java, es importante seguir una estructura clara y detallada. Aquí te dejo una guía paso a paso para cada uno de estos sistemas de gestión de bases de datos.

1. Requisitos previos

Antes de comenzar, asegúrate de que tus alumnos tengan:

- JDK instalado.
- Un IDE como IntelliJ IDEA, Eclipse, o NetBeans.
- Las bibliotecas JDBC específicas para cada base de datos.

2. Estructura general del proceso

1. **Incluir la Biblioteca JDBC en el proyecto:**
    - Descargar el driver JDBC específico para la base de datos.
    - Añadir el driver JDBC al classpath del proyecto.

2. **Crear la cadena de conexión:**

   - Una URL específica para la base de datos.
   - Credenciales de usuario (nombre de usuario y contraseña).

3. **Escribir el código para conectar a la base de datos:**
   - Cargar el driver JDBC.
   - Establecer la conexión utilizando DriverManager.

4. **Ejecutar consultas SQL:**
   - Crear y ejecutar declaraciones SQL utilizando objetos Statement o PreparedStatement.
   - Procesar los resultados usando ResultSet.

3. Ejemplos específicos

a. Conexión a SQL Server

Paso 1: Incluir la biblioteca JDBC

Descarga el driver JDBC para SQL Server desde el sitio web de Microsoft y añade el archivo JAR al classpath del proyecto.

Paso 2: Crear la cadena de conexión

```
String url = "jdbc:sqlserver://localhost:1433;databaseName=mi_base";
String user = "mi_usuario";
String password = "mi_clave";
```

b. Conexión a PostgreSQL

Paso 1: Incluir la Biblioteca JDBC

Descarga el driver JDBC para PostgreSQL desde el sitio web de PostgreSQL y añade el archivo JAR al classpath del proyecto.

Paso 2: Crear la Cadena de Conexión

```
String url = "jdbc:postgresql://localhost:5432/mi_base";
String user = "mi_usuario";
String password = "mi_clave";
```

c. Conexión a Oracle

Paso 1: Incluir la Biblioteca JDBC

Descarga el driver JDBC para Oracle desde el sitio web de Oracle y añade el archivo JAR al classpath del proyecto.

Paso 2: Crear la Cadena de Conexión

```
String url = "jdbc:oracle:thin:@localhost:1521:mi_base";
String user = "mi_usuario";
String password = "mi_clave";
```

La conexión y el crud se pueden reutilizar el código que se utilizó como ejemplo entre JAVA y MySQL

CAPÍTULO 6

Entornos de desarrollo en Lenguaje JAVA

6

Capítulo

6

Entornos de desarrollo en Lenguaje JAVA

6. POO y Lenguaje Java

La programación orientada a objetos (POO) es un paradigma de desarrollo de software que organiza el código en torno a entidades llamadas objetos, los cuales encapsulan datos y comportamientos relacionados. Este enfoque permite crear sistemas modulares, flexibles y reutilizables, facilitando la representación de conceptos del mundo real en estructuras de código coherentes y mantenibles.

En la POO, los objetos son instancias de clases, que actúan como plantillas o moldes para crear dichos objetos. Estas clases definen las propiedades (atributos) y métodos (funciones) que los objetos poseerán, estableciendo así una estructura jerárquica y lógica para el diseño del software. Como señalan Kośnik y Pochmara, (2021), "la POO proporciona una forma natural de modelar el dominio del problema, permitiendo una transición más fluida entre el diseño conceptual y la implementación del software".

Los principios fundamentales de la POO incluyen:

1. Encapsulación
2. Herencia
3. Abstracción

Estos principios trabajan en conjunto para crear un ecosistema de desarrollo que fomenta la creación de código limpio, mantenible y escalable. Como observan Aldabjan, Hammad y Abualigah, (2022), "la POO no es solo una técnica de programación, sino una filosofía de diseño que influye en cómo conceptualizamos y estructuramos soluciones de software”.

La POO ha evolucionado significativamente desde sus inicios, adaptándose a las necesidades cambiantes de la industria del software. En los últimos años, ha incorporado conceptos de programación funcional y reactiva, dando lugar a enfoques híbridos que combinan lo mejor de múltiples paradigmas. Según Alhazmi, Altalhi y Alshehri, (2023) "la POO moderna se caracteriza por su capacidad de integración con otros paradigmas, ofreciendo una flexibilidad sin precedentes en el diseño de arquitecturas de software".

En el contexto del desarrollo ágil y las arquitecturas de microservicios, la POO ha demostrado su valor al proporcionar una base sólida para la creación de componentes independientes y reutilizables. Su énfasis en el modularidad y la encapsulación se alinea perfectamente con los principios de diseño de sistemas distribuidos y escalables.

La programación orientada a objetos no solo es un conjunto de técnicas de codificación, sino un enfoque holístico para el diseño y desarrollo de software que promueve la creación de sistemas robustos,

flexibles y fáciles de mantener. Al modelar el software en términos de objetos que interactúan entre sí, la POO proporciona una abstracción poderosa que permite a los desarrolladores crear soluciones complejas de manera más intuitiva y estructurada.

En última instancia, la POO es una herramienta fundamental en el arsenal del desarrollador moderno, que continúa evolucionando y adaptándose a medida que surgen nuevos desafíos en el campo del desarrollo de software. Su capacidad para representar conceptos complejos de manera clara y organizada la convierte en un paradigma esencial para enfrentar los retos de la programación en el siglo XXI.

6.1 Entorno lenguaje Java

Java es un lenguaje de programación versátil y potente, ampliamente utilizado en el desarrollo de aplicaciones empresariales, móviles y web. Su popularidad en el ámbito universitario se debe a su robustez, portabilidad y enfoque en la programación orientada a objetos (POO) (Gosling et al., 2021).

Para el uso de Java es necesario, instalar programas que facilitan el desarrollo, uno de ellos es Apache NetBeans 20, un entorno de desarrollo integrado (IDE), de código abierto, gratuito, fácil manejo en la edición y depuración de código (Foundation, 2024). Además, Development Kit 21.0.4 (JDK), la última versión para desarrollo en el lenguaje java, actualmente debe registrarse en Oracle para poder acceder a sus descargas (Oracle, 2024).

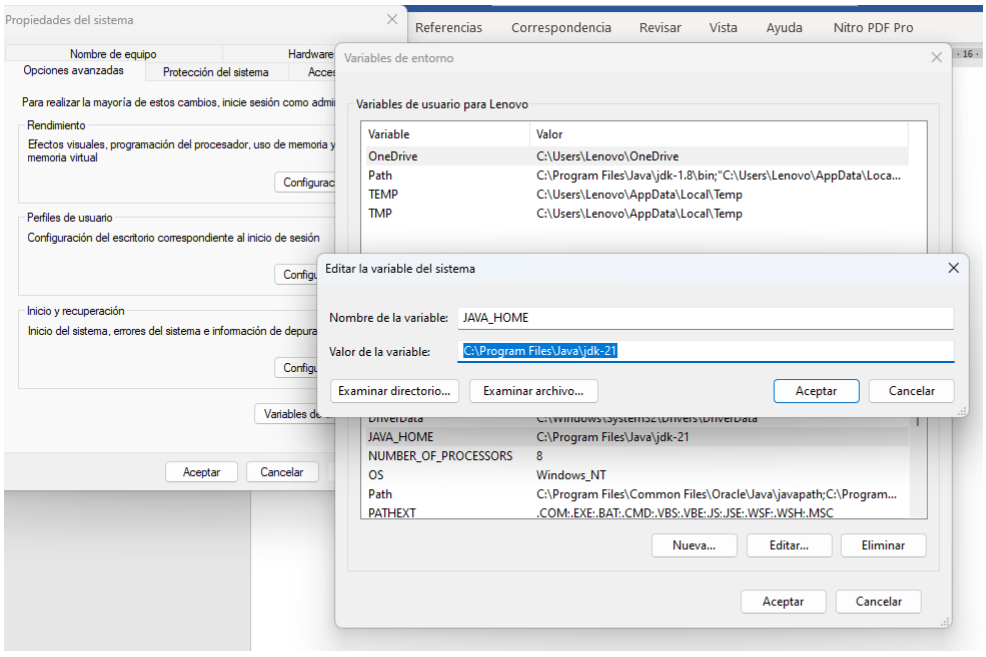
En la sección de referencias bibliográficas se encuentra los links de descarga, pero es necesario aclarar que a medida que el tiempo transcurre las versiones cambian. El proceso de instalación debe iniciar

con JDK 21, pues es requisito para poder instalar Apache NetBeans 20. En el momento que ya cuente con los instaladore de JDK 21, ejecútelo siga los pasos que sugiere el instalador, una vez finalizada la instalación hay que crear la variable de entorno de JDK 21, para esto diríjase a:

- Este Equipo, clic derecho y seleccione propiedades,
- Seleccione Configuración Avanzada del Sistema, clic en Variables de Entorno.
- Ubíquese en Variables del sistema clic en nueva
- Digite en nombre de la variable: JAVA_HOME
- Digite en valor de la Variable: C:\Program Files\Java\jdk-21

Figura 29

Configuración de variable de entorno

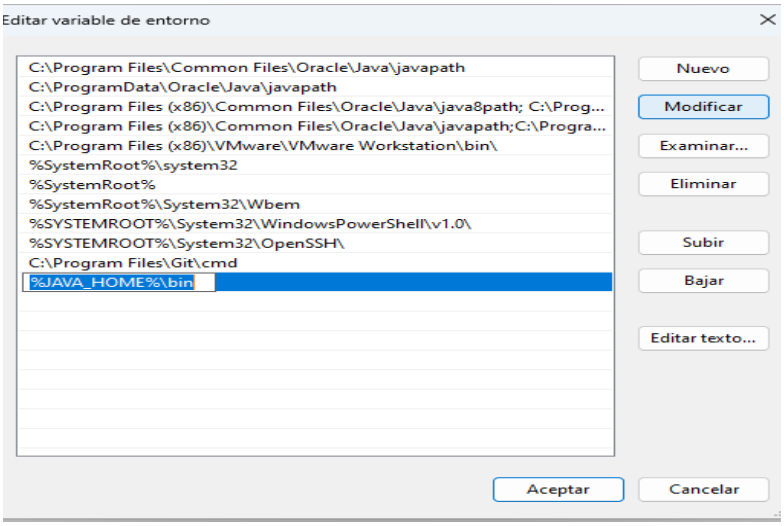


Nota. La figura muestra la configuración de la variable de entorno. Fuente: Elaboración Propia

- Ahora busque la variable Path, de click en nuevo y escriba %JAVA_HOME%\bin, para que la variable JAVA_HOME se registre en el sistema.

Figura 30

Registro de variable en el sistema

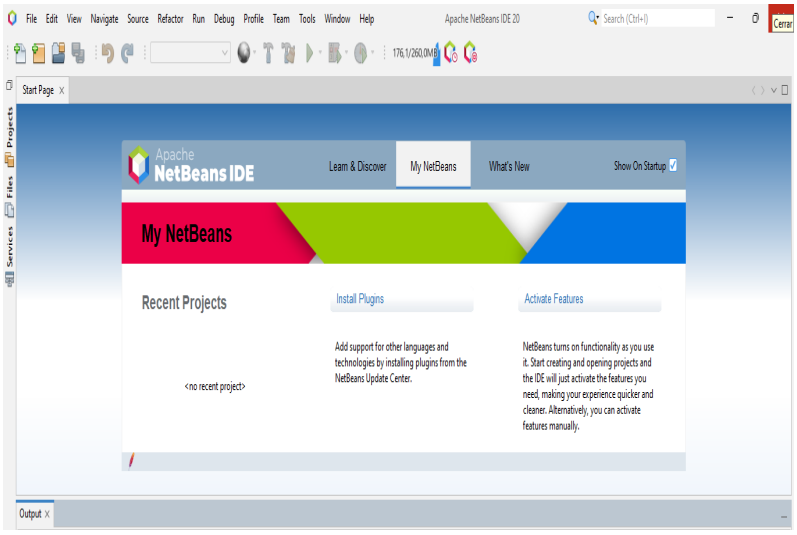


Nota. La figura muestra como registrar la variable en el sistema. Fuente: Elaboración Propia

Por otra parte, el proceso de instalación Apache NetBeans 20, cuando ya disponga del instalador, solo siga los pasos que él mismo sugiere, una vez que haya finalizado la instalación se le presentara la siguiente pantalla.

Figura 31

Entorno de NetBeans



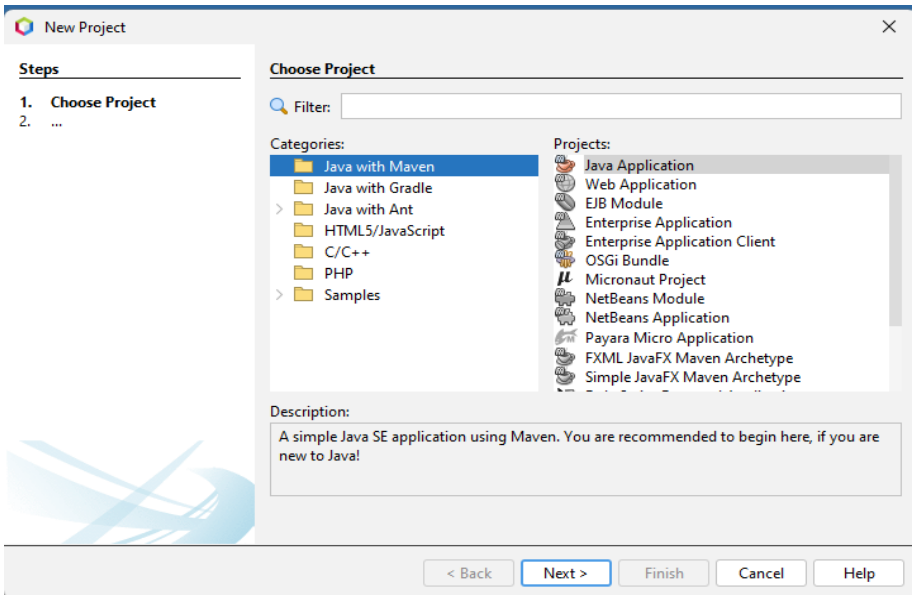
Nota. La figura muestra el entorno de *NetBeans*. Fuente: Elaboración Propia

En este punto ya puede iniciar con su primer proyecto de programación orientada a objetos.

- Diríjase a *file*
- Clic en *New Project*
- Se muestra la siguiente pantalla

Figura 32

Seleccionar el proyecto

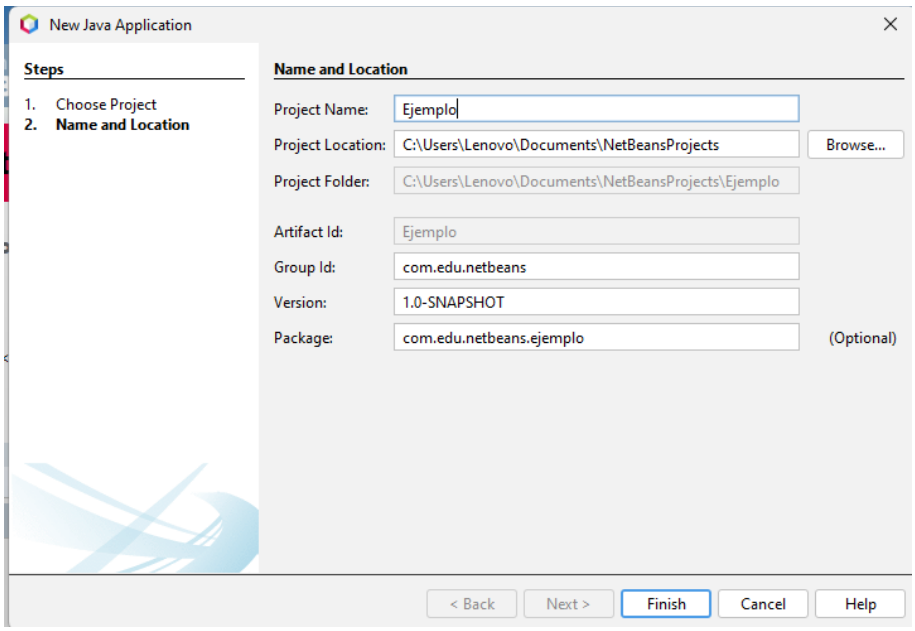


Nota. La figura muestra como seleccionar el proyecto en java. Elaboración Propia.

- Seleccione en *Categories: Java with Maven* y en *Projects*: *Java Application*.
- Clic en *Next*, de un nombre a su proyecto, el programa le pone por defecto la ubicación donde se guardará, como sugerencia personal déjelo así, cambie el *Group Id:* a *com.edu.netbeans.*

Figura 33

Nombre del proyecto y localización



Nota. La figura muestra el nombre y localización del proyecto. Fuente: Elaboración Propia

- Clic en *Finish.*
- El programa le devolverá la siguiente pantalla con un mensaje, lo he cambiado por "Hola Programadores".

Figura 34

Código base

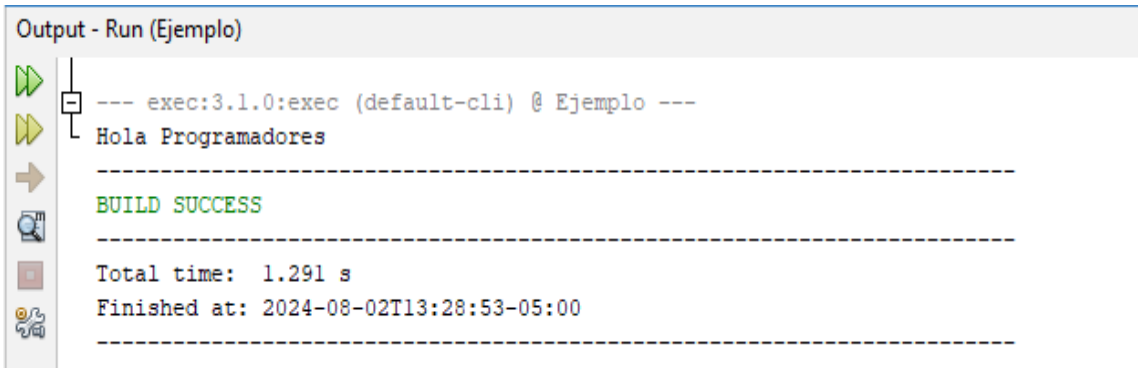
```
1
2   package com.edu.netbeans.ejemplo;
3
4
5   public class Ejemplo {
6
7       public static void main(String[] args) {
8           System.out.println("Hola Programadores ");
9       }
10  }
```

Nota. La figura muestra el código que devuelve java al crear el programa. Fuente: Elaboración Propia

- Para ejecutar el programa basta con dar clic derecho y seleccionar *Run File* o presionar *Shift+f6*, casi inmediatamente se puede observar el mensaje en pantalla "Hola Programadores"

Figura 35

Impresión en pantalla



Nota. La figura muestra el resultado del código. Fuente: Elaboración Propia

- Es importante resaltar que los programas desarrollados en java tienen la a extensión *.java*

6.2. Clases y objetos con Java

Una clase según la definición de Bermón (2021), es un molde, modelo o plantilla del mundo real, que se caracteriza por atributos, constructores y métodos. Para iniciar con la definición de una clase en Java se debe usar la palabra *class,* seguida de un nombre que identifique a la misma, (es importante mencionar que el nombre de la clase debe coincidir con el nombre del archivo), entre llaves {}, se escriben los atributos de la clase, o las variables que conforman las características de los objetos que se crean desde esta clase.

Los objetos son considerados como una instancia de clase, tienen identidad, que es el nombre que distingue al objeto, el estado que son

las características que describen al objeto y el comportamiento que es lo que puede hacer el objeto, el nivel de visibilidad o accesibilidad de los mismos, pueden ser *private* al cual se accede solamente desde la clase a la que le pertenece, *public* se accede a este desde un método desarrollado en cualquier clase y *protected* solo se accede si el método fue implementado en una clase que herede o que haya sido realizada dentro del mismo paquete.

Para ejemplificación se trabaja con el cálculo del área de un prisma triangular, mismo que se desarrolla conforme se avanza con la temática, para el desarrollo, se debe ingresar por teclado la base, altura del triángulo y altura del rectángulo, por lo que se usa la clase *Scanner* que ayuda a ingresar datos por teclado, además se crea dos clases una para la parte lógica en donde se desarrollan los cálculos y otra para el ingreso de datos y publicación del resultado. A continuación, se presenta una versión preliminar de una clase llamada AreaPrismaTriangular, en el que se declaran 4 variables de tipo *double*: vbase, valturatriangulo, valturarectangulo, areatotal, las tres primeras servirán para recibir los valores ingresados por teclado y la última para almacenar el resultado de la operación;

Figura 36

Declaración de una clase

```
package com.edu.netbeans.areaprismatriangular;

public class AreaPrismaTriangular {
    double vbase, valturatriangulo, valturarectangulo, areatotal;

}
```

Nota. La figura muestra como declarar una clase. Fuente: Elaboración Propia

Los métodos son el comportamiento del que puede tener el objeto, en una clase se declaran con un modificador de acceso, tipo de retorno, nombre del método, parámetros y el cuerpo del método que va escrito entre {}. La sintaxis es la siguiente aplicado al ejemplo que se está desarrollando.

Figura 37

Declaración de un método

```
public void areatotal(){
    areatotal= (vbase*valturarectangulo*3)+ (((vbase*valturatriangulo)/2)*2);
}
```

Nota. La figura muestra como declarar un método. Fuente: Elaboración Propia

Los métodos constructores son aquellos que crean objetos a partir de una clase, es obligatorio que al ser creados tengan el mismo nombre que la clase, no tienen valor de retorno, usan el operador new para crear el objeto y como referencia de uso de otras clases, de acuerdo al tipo de visibilidad que este tenga.

Figura 38

Declaración de un método constructor

```
AreaPrismaTriangular valores = new AreaPrismaTriangular
```

Nota. La figura muestra como declarar un método constructor. Fuente: Elaboración Propia.

En lo referente a los modificadores de tipo o visibilidad de los métodos, existen el modificador *static,* también llamados métodos de clase, indica que el método pertenece a la clase y no a una instancia de la misma, es decir que puede ser llamado sin necesidad de crear un objeto de la clase*, public* se puede invocar al método desde cualquier clase*, private* solo pueden ser llamados desde la clase en la que están

definidos y *protected* solo pueden ser invocados por clases que heredan de la clase en la que fueron creados o que estén dentro del mismo paquete.

A continuación, se muestra el programa completo para el cálculo del área de un prisma triangular. Como se mencionó anteriormente, se necesita el ingreso por teclado de la base, altura del triángulo y altura del rectángulo, se crean dos clases AreaPrismaTriangularMain.java y AreaPrismaTriangular.java, la primera para el ingreso de los datos por teclado y la segunda para el cálculo del área. Para el ingreso de los datos, es necesario el uso de la clase *Scanner* y para poder usarla se debe importar la librería (línea de código 3), se declara un objeto "entrada" de tipo *Scanner* (línea de código 7), se declara variables (línea de código 10, 13, 16) que con apoyo del objeto "entrada" recibirán los ingresos por teclado. Para enviar los datos a la clase donde se desarrolla el cálculo se crea un constructor del mismo tipo de la clase que recibe los datos (línea de código 18), para mostrar los resultados se usa el constructor con el método imprimir (línea de código 20).

Figura 39

Ejemplo en java cálculo del área de un prisma triangular parte 1

```
1
2   package com.edu.netbeans.areaprismatriangular;
3   import java.util.Scanner;
4
5   public class AreaPrismaTriangularMain {
6       public static void main(String[] args) {
7           Scanner entrada =new Scanner (System.in);
8
9           System.out.println("Ingrese la base: ");
10          double base = entrada.nextDouble();
11
12          System.out.println("Ingrese la altura del triangulo: ");
13          double alturadeltriangulo = entrada.nextDouble();
14
15          System.out.println("Ingrese la altura del rectángulo; ");
16          double alturadelrectangulo = entrada.nextDouble();
17
18          AreaPrismaTriangular valores = new AreaPrismaTriangular (base,alturadelrectangulo, alturadeltriangulo);
19
20          valores.imprimir();
21      }
22  }
```

Para el cálculo del área, se declaran 3 variables que recibirán los datos ingresados por teclado (línea de código 5), con el constructor se pasan los valores a la clase donde se desarrollan los cálculos (línea de código 7), con la palabra *this,* que hace referencia a la clase AreaPrismaTriangular.java igualamos los valores que llegaron se guarden en las variables creadas para el efecto (línea de código 8, 9, 10), con esto se crea un método que permita calcular el área del prisma triangular, de tipo *public void* para poder retornar el resultado (línea de código 12), se realiza el cálculo (línea de código 13) y se asigna a la variable, para imprimir el resultado en pantalla se crea un método imprimir (línea de código 15), dentro de este se llama al método que realizo los cálculos (línea de código 16), y se imprime el resultado (línea de código 17). Finalmente, desde la clase AreaPrismaTriangularMain.java llamamos al método imprimir (línea de código 20)

Figura 40

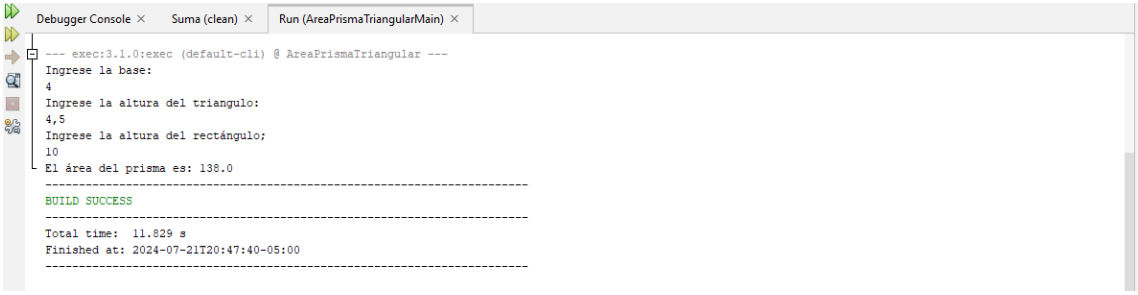
Ejemplo en java cálculo del área de un prisma triangular parte 2

```
1
2   package com.edu.netbeans.areaprismatriangular;
3
4   public class AreaPrismaTriangular {
5       double vbase, valturatriangulo, valturarectangulo, areatotal;
6
7       public AreaPrismaTriangular (double base,double alturadelrectangulo, double alturadeltriangulo){
8           this.vbase = base;
9           this.valturatriangulo = alturadeltriangulo;
10          this.valturarectangulo = alturadelrectangulo;
11      }
12      public void areatotal(){
13          areatotal= (vbase*valturarectangulo*3)+ (((vbase*valturatriangulo)/2)*2);
14      }
15      public void imprimir (){
16          areatotal();
17          System.out.println("El área del prisma es: " + areatotal );
18      }
19  }
```

Al ejecutar el programa se visualiza la siguiente pantalla. El usuario debe ingresar por teclado los valores solicitados y se muestra el resultado.

Figura 41

Resultado del ejemplo en java cálculo del área de un prisma triangular



6.3 Herencia y polimorfismo con Java

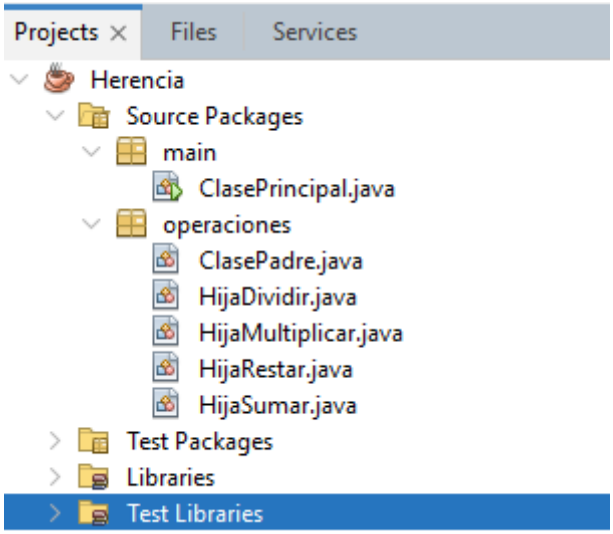
La herencia es cuando una clase hereda atributos y métodos de clases antecesoras, es decir que permite reutilizar código existente sin necesidad de escribirlo nuevamente, solo se debe indicar que se usará algo creado anteriormente, por ende, ahorra código y tiempo. El código que se va a reutilizar se escribe en la clase padre, es decir se declaran variables y se crean los métodos y en la clase derivada se reutilizan sin ningún inconveniente. Existen dos tipos de herencia la múltiple y la simple, que es, la que se utiliza en java y esta consiste en compartir el código creado en una clase padre y usarlo cuantas veces sea necesario. Por otra parte, el polimorfismo indica que un mismo método puede comportarse de diferente manera en consideración del objeto que lo ha llamado, es decir el polimorfismo permite que un método se comunique con clases diferentes y que su comportamiento cambie dependiendo con quien se comunicó (Bermón, 2021).

Para mejor explicación sobre la herencia, se crea un programa que incluye una clase padre en la que se solicita el ingreso de dos números por teclado, cuatro clases hijas, la primera que permita realizar la suma de los dos números, la segunda para que reste, la tercera que multiplique y la cuarta que divida. Para el efecto las clases

hijas heredan las variables y funciones creadas en la clase padre, lo que permite ahorrar tiempo y escritura de código repetitivo las cuatro clases mencionadas pertenecen al paquete operaciones. Así también se crea una clase principal *main*, que permita crear instancias para la comunicación con las clases creadas e imprimir en pantalla los resultados, esta clase pertenece al paquete main como se muestra en la siguiente pantalla.

Figura 42

Proyecto de Herencia con sus clases



A continuación, se describe el código de la clase padre, en el que se declaran tres variables, numero1, numero2 y resultado, las dos primeras para almacenar los valores ingresados por teclado y la tercera para almacenar el resultado, anteriormente se mencionó a la clase *Scanner,* en esta ocasión se usa nuevamente para recibir el ingreso de los números. Se crea el método *SolicitarDatos,* que permite almacenar en las variables los valores ingresados. También, el método *ImprimeResultado,* que permite la impresión en pantalla del resultado.

Figura 43

Ejemplo de Herencia clase padre

```
package operaciones;
import java.util.Scanner;
//clase padre
public class ClasePadre {
    protected int numero1, numero2, resultado;
    Scanner ingreso = new Scanner(System.in);

    //Método que solicita números al usuario
    public void SolicitarDatos(){
        System.out.print("Ingrese el primer numero: ");
        numero1 = ingreso.nextInt();

        System.out.print("Ingrese el segundo numero: ");
        numero2 = ingreso.nextInt();
    }

    //Método imprime el resultado en pantalla
    public void ImprimeResultado(){
        System.out.println(resultado);
    }
}
```

Seguidamente, se muestra las clases hijas en donde solamente se desarrollan los cálculos con los valores heredados de la clase padre. Para que la clase hija herede es necesario usar la palabra *extends* junto con el nombre de la clase padre (línea de código 3). Para retornar el valor de la operación se crea un método (línea de código 4) en el cual se desarrolla la operación, con las variables declaradas y heredadas en la clase padre.

Figura 44

Ejemplo de Herencia clase HijaSumar

```
1  package operaciones;
2
3  public class HijaSumar extends ClasePadre{
4      public void Sumar(){
5          resultado = numero1 + numero2;
6      }
7  }
```

Figura 45

Ejemplo de Herencia clase HijaRestar

```
1   package operaciones;
2
3   public class HijaRestar extends ClasePadre{
4       public void Restar(){
5           resultado = numero1 - numero2;
6       }
7   }
```

Figura 46

Ejemplo de Herencia clase HijaMultiplicar

```
1   package operaciones;
2
3   public class HijaMultiplicar extends ClasePadre{
4       public void Multiplicar(){
5           resultado = numero1 * numero2;
6       }
7   }
```

Figura 47

Ejemplo de Herencia clase HijaDividir

```
1   package operaciones;
2
3   public class HijaDividir extends ClasePadre{
4       public void Dividir(){
5           resultado = numero1 / numero2;
6       }
7   }
```

Finalmente, en la clase principal se deben importar las clases hijas.

Figura 48

Ejemplo de Herencia clase principal llamada a clases hijas

```
1   package main;
2   import operaciones.HijaMultiplicar;
3   import operaciones.HijaSumar;
4   import operaciones.HijaRestar;
5   import operaciones.HijaDividir;
```

Es necesario crear objetos, del tipo de clase de cada clase hija, con la finalidad de llamar a los métodos para que se desarrollen las operaciones y poder mostrar resultados.

Figura 49

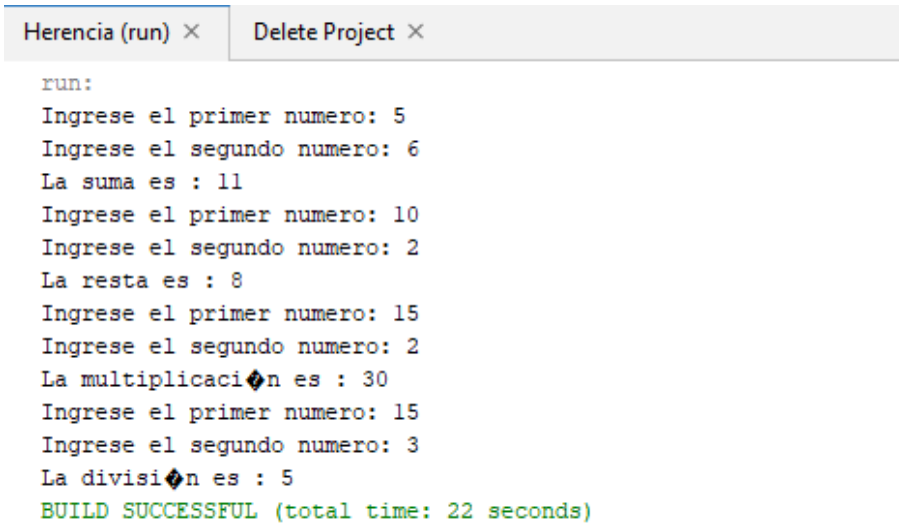
Ejemplo de Herencia clase principal llamada a métodos e impresión de resultados

```
7   public class ClasePrincipal {
8       public static void main(String[] args) {
9           HijaSumar RespuestaSuma = new HijaSumar();
10          RespuestaSuma.SolicitarDatos();
11          RespuestaSuma.Sumar();
12          System.out.print("La suma es : ");
13          RespuestaSuma.ImprimeResultado();
14
15          HijaRestar RespuestaResta = new HijaRestar();
16          RespuestaResta.SolicitarDatos();
17          RespuestaResta.Restar();
18          System.out.print("La resta es : ");
19          RespuestaResta.ImprimeResultado();
20
21          HijaMultiplicar RespuestaMultiplicacion = new HijaMultiplicar();
22          RespuestaMultiplicacion.SolicitarDatos();
23          RespuestaMultiplicacion.Multiplicar();
24          System.out.print("La multiplicación es : ");
25          RespuestaMultiplicacion.ImprimeResultado();
26
27          HijaDividir RespuestaDivision = new HijaDividir();
28          RespuestaDivision.SolicitarDatos();
29          RespuestaDivision.Dividir();
30          System.out.print("La división es : ");
31          RespuestaDivision.ImprimeResultado();
32      }
```

Al ejecutar el programa se obtiene lo siguiente:

Figura 50

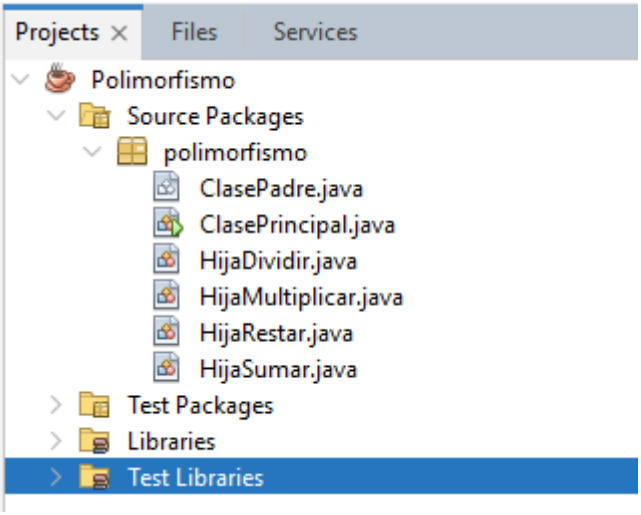
Ejemplo de Herencia resultado del programa



Para ejemplificar el polimorfismo se plantea un programa que permita realizar las cuatro operaciones básicas, para esto se crean seis clases, la clase principal que permite interactuar entre clases, la clase padre que hereda a las clases hijas los métodos y variables para evitar la reescritura de código.

Figura 51

Proyecto de Polimorfismo con sus clases



En la clase padre, se declara tres variables numero1, numero2, resultado, las dos primeras almacenan los valores ingresados por teclado y la tercera el resultado de las operaciones, un objeto de la clase Scanner para solicitar datos desde teclado. Se crea el método *SolicitarDatos* y para aplicar el polimorfismo el método *Operaciones* (línea de código 17), este método tendrá un comportamiento diferente dependiendo con la clase que se comunique (es importante aclarar que tanto el método que se usa para polimorfismo como la clase deben estar declaradas como *public abstract*) y el método *ImprimirResultado* que imprime el resultado en pantalla.

Figura 52

Proyecto de Polimorfismo Clase Padre

```
1   package polimorfismo;
2   import java.util.Scanner;
3
4   public abstract class ClasePadre {
5
6       protected int numero1, numero2, resultado;
7       Scanner ingreso = new Scanner(System.in);
8
9       public void SolicitarDatos() {
10          System.out.print("Ingrese el primer numero: ");
11          numero1 = ingreso.nextInt();
12
13          System.out.print("Ingrese el segundo valor: ");
14          numero2 = ingreso.nextInt();
15      }
16
17      public abstract void Operaciones();
18
19      public void ImprimirResultado(){
20          System.out.println(resultado);
21      }
22  }
```

Para las clases hijas, es necesario aplicar herencia (línea de código 3) y cuando se habla de polimorfismo aparece la palabra @Override, que indica que el método se va a sobrescribir, y se escribe el método que se sobrescribe (línea de código 6) y la operación.

Figura 53

Proyecto de Polimorfismo Clase HijaSumar

```
1   package polimorfismo;
2
3   public class HijaSumar extends ClasePadre{
4
5       @Override
6       public void Operaciones(){
7           resultado = numero1 + numero2;
8       }
9   }
```

Figura 54

Proyecto de Polimorfismo Clase HijaRestar

```
1   package polimorfismo;
2
3   public class HijaRestar extends ClasePadre{
4
5       @Override
6       public void Operaciones(){
7           resultado = numero1 - numero2;
8       }
9   }
```

Figura 55

Proyecto de Polimorfismo Clase HijaMultiplicar

```
1   package polimorfismo;
2
3   public class HijaMultiplicar extends ClasePadre{
4
5       @Override
6       public void Operaciones(){
7           resultado = numero1 * numero2;
8       }
9   }
```

Figura 56

Proyecto de Polimorfismo Clase HijaDividir

```
1   package polimorfismo;
2
3   public class HijaDividir extends ClasePadre{
4
5       @Override
6       public void Operaciones(){
7           resultado = numero1 / numero2;
8       }
9   }
```

Para la clase principal, se crea un objeto para poder instanciar, del tipo de la clase padre llamado RespuestaSuma que será igual al nombre de la clase con la que se quiere crear el polimorfismo, es decir se crea un constructor que hace referencia a la clase con la que cambiara de comportamiento (línea de código 6, 11,16,21). Se llama a los métodos SolicitarDatos, Operaciones que hace el polimorfismo dependiendo de la instancia, tiene su comportamiento y el método ImprimirResultado.

Figura 57

Proyecto de polimorfismo clase principal

```
package polimorfismo;

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

        ClasePadre RespuestaSuma = new HijaSumar();
        RespuestaSuma.SolicitarDatos();
        RespuestaSuma.Operaciones();
        RespuestaSuma.ImprimirResultado();

        ClasePadre RespuestaResta = new HijaRestar();
        RespuestaResta.SolicitarDatos();
        RespuestaResta.Operaciones();
        RespuestaResta.ImprimirResultado();

        ClasePadre RespuestaMultiplicacion = new HijaMultiplicar();
        RespuestaMultiplicacion.SolicitarDatos();
        RespuestaMultiplicacion.Operaciones();
        RespuestaMultiplicacion.ImprimirResultado();

        ClasePadre RespuestaDivision = new HijaDividir();
        RespuestaDivision.SolicitarDatos();
        RespuestaDivision.Operaciones();
        RespuestaDivision.ImprimirResultado();
    }
}
```

El resultado del programa es:

Figura 58

Proyecto de Polimorfismo Resultado

```
Output - Polimorfismo (run)
run:
Ingrese el primer numero: 1
Ingrese el segundo valor: 2
3
Ingrese el primer numero: 5
Ingrese el segundo valor: 6
-1
Ingrese el primer numero: 5
Ingrese el segundo valor: 3
15
Ingrese el primer numero: 27
Ingrese el segundo valor: 3
9
BUILD SUCCESSFUL (total time: 13 seconds)
```

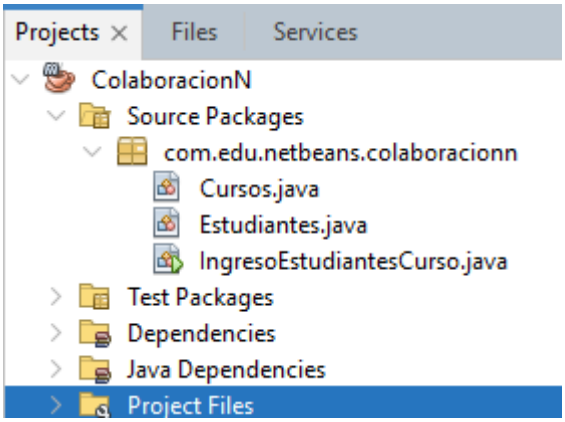
6.4 Colaboración entre clases

La colaboración entre clases permite que varias clases interactúen entre sí para lograr un objetivo común en una aplicación. Se alcanza mediante las relaciones de dependencia entre las clases (Alhazmi et al., 2023).

Para ejemplificar la colaboración entre clases, se desarrolla un programa que permita el ingreso del nombre de un curso, el número de estudiantes a ingresar y datos del estudiante, muestra la colaboración de las clases en la gestión de ingreso de estudiantes a un curso. Se crean tres clases: Cursos, Estudiantes e IngresoEstudinatesCurso.

Figura 59

Proyecto de Colaboración entre clases con sus clases



En la clase Estudiantes se declaran dos variables nombre e id, se crea un constructor con el mismo nombre de la clase, se asigna el valor del parámetro nombre a la variable de instancia nombre, lo mismo sucede con id (línea de código 7, 8), el método ObtenerNombre, devuelve el nombre del estudiante y el método ObtnerId regresa el id del estudiante, el método ImprimirDatos, imprime en pantalla los datos.

Figura 60

Proyecto de Colaboración entre clases, Clase Estudiante

```java
package com.edu.netbeans.colaboracionn;
public class Estudiantes {
    private final String nombre;
    private final int id;

    public Estudiantes(String nombre, int id) {
        this.nombre = nombre;
        this.id = id;
    }

    public String ObtenerNombre() {
        return nombre;
    }

    public int ObtenerId() {
        return id;
    }

    public void ImprimirDatos() {
        System.out.println("ID: " + id + ", Nombre: " + nombre);
    }
}
```

La clase Cursos, se declara la variable nombreCurso que almacenara el nombre del curso, también una lista que almacena objetos del tipo Estudiantes con implementación ArrayList. El constructor con el mismo nombre de la clase que inicializa valores. El método agregarEstudinates que agrega estudiantes a la lista estudiantes y el método mostrar estudiantes que imprime los datos ingresados en pantalla.

Figura 61

Proyecto de Colaboración entre clases, clase cursos

```java
package com.edu.netbeans.colaboracionn;

import java.util.ArrayList;
import java.util.List;

public class Cursos {
    private final String nombreCurso;
    private final List<Estudiantes> estudiantes;

    public Cursos(String nombreCurso) {
        this.nombreCurso = nombreCurso;
        this.estudiantes = new ArrayList<>();
    }

    public void agregarEstudiante(Estudiantes estudiante) {
        estudiantes.add(estudiante);
    }

    public void mostrarEstudiantes() {
        System.out.println("Curso: " + nombreCurso);
        for (Estudiantes estudiante : estudiantes) {
            estudiante.ImprimirDatos();
        }
    }
}
```

Finalmente, la clase IngresoEstudinatesCurso, usa un objeto de la clase scanner para leer los datos ingresados por teclado, se asigna a la variable correspondiente, por ejemplo, a nombreCurso se asigna el nombre de curso que el usuario proporcionó, algo similar ocurre con el número, nombre y el id del estudiante, se crea un objeto estudiante con el nombre e id proporcionados y agrega el estudiante al curso y muestra los estudiantes.

Figura 62

Proyecto de Colaboración entre clases, Clase IngresoEstudinatesCurso

```java
package com.edu.netbeans.colaboracionn;
import java.util.Scanner;
public class IngresoEstudiantesCurso {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Crear curso
        System.out.println("Ingrese el nombre del curso:");
        String nombreCurso = scanner.nextLine();
        Cursos curso = new Cursos(nombreCurso);
        // Ingreso de estudiantes
        System.out.println("¿Cuántos estudiantes desea ingresar?");
        int numeroEstudiantes = scanner.nextInt();
        scanner.nextLine(); // Limpiar el buffer
        for (int i = 0; i < numeroEstudiantes; i++) {
            System.out.println("Ingrese el nombre del estudiante " + (i + 1) + ":");
            String nombreEstudiante = scanner.nextLine();

            System.out.println("Ingrese el ID del estudiante " + (i + 1) + ":");
            int idEstudiante = scanner.nextInt();
            scanner.nextLine(); // Limpiar el buffer

            Estudiantes estudiante = new Estudiantes(nombreEstudiante, idEstudiante);
            curso.agregarEstudiante(estudiante);
        }
        // Imprime información de los estudiantes en el curso
        curso.mostrarEstudiantes();
    }
}
```

Figura 63

Proyecto de Colaboración entre clases, resultado del programa

```
Output - Run (IngresoEstudiantesCurso)
Compiling 3 source files with javac [debug target 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ ColaboracionN ---
Ingrese el nombre del curso:
GTI A
¿Cuántos estudiantes desea ingresar?
2
Ingrese el nombre del estudiante 1:
Luis Pilco
Ingrese el ID del estudiante 1:
1803592453
Ingrese el nombre del estudiante 2:
Klever Chicaiza
Ingrese el ID del estudiante 2:
1803546972
Curso: GTI A
ID: 1803592453, Nombre: Luis Pilco
ID: 1803546972, Nombre: Klever Chicaiza
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Referencias

Aldabjan, A., Hammad, M., & Abualigah, L. (2022). *Software design patterns: Types, benefits, and applications*. PeerJ Computer Science, 8. https://doi.org/10.7717/peerj-cs.1002

Alhazmi, S., Altalhi, A., & Alshehri, M. (2023). *A systematic literature review of object-oriented metrics*. IEEE Access, 591-606. https://doi.org/10.1109/ACCESS.2023.3255577

Bailón, J., & Baltazar, J. (1 de enero de 2021). *Características de la POO*. En *Algoritmos y codificación*. Portal Académico del CCH, UNAM. https://portalacademico.cch.unam.mx/cibernetica1/algoritmos-y-codificacion/caracteristicas-POO

Bermón, L. (2021). *Ejercicios de programación orientada a objetos con Java y UML*. Editorial Universidad Nacional de Colombia.

Foundation, T. A. (20 de julio de 2024). *Apache NetBeans*. https://netbeans.apache.org/front/main/download/nb20/

geeksforgeeks. (23 de noviembre de 2022). *Upcasting vs Downcasting in Java*. https://www.geeksforgeeks.org/upcasting-vs-downcasting-in-java/

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., & Bierman, G. (2021). *The Java® Language Specification Java SE 16 Edition*. Oracle America, Inc.

https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf

Kośnik, L., & Pochmara, M. (2021). *Object-oriented programming paradigm in modern computer science education*. Education Sciences, 11(7), 350. https://doi.org/10.3390/educsci11070350

Oracle. (20 de julio de 2024). *Oracle*. https://www.oracle.com/pe/java/technologies/downloads/#jdk21-windows

Román, C. (s.f.). *Programación orientada a objetos con Java*. http://profesores.fi-b.unam.mx/carlos/java (abril, 2022).

Glosario de términos

A

Abstracción: Propiedad que permite enfocarse en los aspectos esenciales de una clase o objeto, ignorando detalles innecesarios.

Algoritmo: Secuencia de pasos detallados que describe cómo resolver un problema específico.

API: Interfaz de programación de aplicaciones, que proporciona funciones para interactuar con software o servicios.

Archivo JAR: Formato de archivo utilizado en Java para distribuir aplicaciones y bibliotecas.

Array: Estructura de datos que almacena múltiples valores del mismo tipo en secuencia.

ASCII: Conjunto de caracteres numéricos que representan letras, números y símbolos en sistemas informáticos.

B

Base de datos: Sistema organizado que almacena y gestiona información estructurada.

Biblioteca de clases: Colección organizada de clases reutilizables.

Binario: Representación numérica en base 2 (usando solo dígitos 0 y 1).

C

Clase: Plantilla o molde para crear objetos con características y comportamientos similares.

Compilador: Software que traduce código fuente a lenguaje máquina.

Constructor: Método especial que se ejecuta al crear un nuevo objeto.

E

Encapsulamiento: Mecanismo que oculta la implementación interna de una clase.

Excepción: Evento inesperado que indica un error en el programa.

Expresión: Subparte de una sentencia que representa un valor.

H

Herencia: Relación entre clases donde una subclase hereda propiedades de una superclase.

I

IDE: Entorno de desarrollo integrado para programadores.

Instancia: Objeto creado a partir de una clase.

Interfaz: Contrato de comportamiento compartido por varias clases.

L

Lenguaje de programación: Sistema de instrucciones para escribir programas.

M

Método: Función definida dentro de una clase.

O

Objetos: Instancias de clases, que encapsulan datos y comportamientos.

Operador: Símbolo que representa operaciones básicas como suma, resta, multiplicación, etc.

P

Parámetro: Valor pasado a un método cuando se llama.

Polimorfismo: Capacidad de un objeto de tomar muchas formas diferentes.

S

Sentencia: Unidad de código que representa una acción o secuencia de acciones.

Sobrecarga: Definición de varios métodos con el mismonombre pero diferente firma.

Superclase: Clase cuya estructura se hereda por otras clases.

T

Tipo de dato: Categoría de valores que puede almacenar una variable.

V

Variable: Nombre asignado a un espacio de memoria para almacenar un valor.

ISBN: 978-9942-679-46-8