Compás
capacitación e investigación

2025

Programación orientada a objetos en java: fundamentos, modelado y aplicaciones prácticas

Joffre J Cartuche Calva
Bertha E Mazón Olivo
Wilmer Braulio Rivas Asanza

Programación orientada a objetos en java: fundamentos, modelado y aplicaciones prácticas

Joffre J Cartuche Calva
Bertha E Mazón Olivo
Wilmer Braulio Rivas Asanza

ISBN: 978-9942-53-026-4 DOI: http://doi.org/10.48190/9789942530264

Compás
capacitación e investigación

Bertha E Mazón Olivo
Wilmer Braulio Rivas Asanza

Primera edición, 2025-11-08

ISBN: 978-9942-53-026-4
DOI: http://doi.org/10.48190/9789942530264
Distribución online
Acceso abierto

Cita
Cartuche, J., Mazón, B., Rivas, W. (2025) Programación orientada a objetos en java: fundamentos, modelado y aplicaciones prácticas. Editorial Grupo Compás

Este libro es parte de la colección de la Univesidad Técnica de Machala y ha sido debidamente examinado y valorado en la modalidad doble par ciego con fin de garantizar la calidad de la publicación.

DEDICATORIA

A nuestras queridas familias, especialmente a nuestras parejas e hijos, por su inquebrantable apoyo, paciencia y comprensión en cada paso de este camino. Su respaldo ha sido fundamental en la realización de este trabajo.

A nuestros estudiantes de la Carrera de Tecnologías de la Información, Ciencias de Datos e Inteligencia Artificial quienes dan sus primeros pasos en el apasionante mundo de la programación orientada a objetos. Que este libro sea una guía en su proceso de aprendizaje, inspirándolos a desarrollar soluciones innovadoras y a cultivar una pasión por la tecnología.

A todos aquellos que creen en el poder del conocimiento y la educación como herramientas de transformación.

INTRODUCCIÓN

En el dinámico mundo del desarrollo de software, la evolución constante de la tecnología demanda enfoques innovadores y estructurados para crear aplicaciones robustas y escalables. La *Programación orientada a objetos (POO)* ha transformado radicalmente la manera en que concebimos y desarrollamos soluciones, al permitir representar problemas del mundo real mediante modelos intuitivos y reutilizables.

A diferencia de la programación estructurada tradicional, centrada en procedimientos y funciones, la *POO* descompone un problema en entidades denominadas objetos, cada uno con sus propias propiedades y comportamientos. Este enfoque facilita la comprensión, el mantenimiento y la evolución del código, al mismo tiempo que fomenta la reutilización, el encapsulamiento, la abstracción y la modularidad, pilares esenciales para el desarrollo de software moderno y adaptable a cambios constantes.

Sin embargo, la implementación exitosa de un sistema no depende únicamente del código, sino también de una planificación y diseño precisos. Aquí es donde *UML (Unified Modeling Language)* se convierte en una herramienta vital, permitiendo representar gráficamente la estructura y el comportamiento de un sistema. *UML* no solo facilita la comunicación entre desarrolladores, arquitectos y analistas, sino que también asegura que el diseño conceptual se traduzca de manera coherente en una implementación real y efectiva.

Entre los múltiples lenguajes que han adoptado el paradigma orientado a objetos, Java se destaca por su portabilidad, seguridad y potencia. Basado en la *máquina virtual de Java (JVM)*, este lenguaje permite desarrollar aplicaciones para una amplia gama de entornos, desde sistemas empresariales hasta soluciones móviles y en la nube.

Esta convergencia de técnicas y herramientas – la solidez de la *POO*, la claridad del modelado con *UML* y la versatilidad de Java sienta las bases para construir software que no solo resista el paso del tiempo, sino que también evolucione con las necesidades del mercado. Este libro, Programación Orientada a Objetos en Java: Fundamentos, Modelado y Aplicaciones Prácticas, invita a docentes, estudiantes universitarios y profesionales a explorar y dominar este enfoque integral, que va desde el diseño conceptual hasta la implementación de soluciones de software eficientes y escalables.

¡Prepárate para embarcarte en un viaje donde la teoría y la práctica convergen para construir software robusto y escalable!

OBJETIVOS GENERALES

El objetivo de este libro es proporcionar una guía integral sobre la *programación orientada a objetos (POO)*, combinando fundamentos teóricos y su aplicación práctica en el desarrollo de software. Se abordan de manera estructurada los conceptos esenciales, incluyendo clases, objetos, encapsulamiento, herencia, clases abstractas, interfaces y colecciones de objetos, junto con el modelado de datos mediante *UML* (*Unified Modeling Language*). A través de una metodología que equilibra teoría, modelado y ejercicios prácticos, el libro facilita una transición efectiva desde el diseño hasta la implementación del código en Java.

ESTRUCTURA GENERAL DEL LIBRO

A continuación, se presenta un resumen de lo que se tratará en cada capítulo del libro:

Capítulo I – Fundamentos de la programación orientado a objetos

Este capítulo introduce los fundamentos esenciales de la *POO*. Se explorarán los principios, ventajas y desventajas del paradigma, destacando sus pilares fundamentales como el encapsulamiento, la herencia y el polimorfismo. Además, se presentarán las herramientas necesarias para el desarrollo de software, haciendo énfasis en el uso de *entornos integrados de desarrollo (IDE)* para Java, así como en la importancia del modelado de datos mediante *UML*.

Capítulo II - Clases y objetos

En este capítulo se profundizará en la creación y manejo de clases y objetos en Java. Se explicará cómo definir clases, establecer atributos y métodos, y se abordarán los distintos tipos de constructores y su sobrecarga. Se analizará el ciclo de vida de los objetos, así como aspectos de memoria y referencias. También se cubrirán los especificadores de acceso y las buenas prácticas de encapsulamiento.

Capítulo III - Herencia y polimorfismo

Este capítulo se centra en los mecanismos que permiten la reutilización de código a través de la herencia y la implementación del polimorfismo. Se explicará cómo se representan las relaciones entre clases superclase y subclase y la propagación de atributos y métodos. Asimismo, se abordará en

detalle el concepto de polimorfismo, analizando la sobrecarga y la sobreescritura de métodos.

Capítulo IV - Clases abstractas e interfaces

En este capítulo se estudiarán dos elementos clave para diseñar arquitecturas de software robustas: las clases abstractas y las interfaces. Se explicarán las características y limitaciones de las clases abstractas, se presentará el concepto de interfaces, destacando su propósito, las diferencias con las clases abstractas y la posibilidad de implementar múltiples interfaces en una c

Metodología Didáctica

La metodología de este libro se sustenta en un enfoque constructivista y de aprendizaje activo, que considera al estudiante como protagonista del proceso educativo y al docente como facilitador del conocimiento (Mora & García, 2022). El propósito es lograr una integración efectiva entre la teoría, el modelado y la práctica de la *POO* con Java y *UML*.

La propuesta metodológica se articula en cuatro fases secuenciales, aplicadas en cada capítulo:

1. Exploración Conceptual

   Se inicia con la presentación de los fundamentos teóricos de la programación orientada a objetos (clases, objetos, encapsulamiento, herencia, polimorfismo, interfaces, entre otros). Estos contenidos se acompañan de ejemplos de la vida cotidiana, analogías y preguntas de enfoque que estimulan la reflexión inicial y preparan al estudiante para conectar los conceptos con la práctica (Vera & Vera, 2023).

2. Modelado y Representación Gráfica

   El segundo paso consiste en el uso del *UML* como herramienta de abstracción y comunicación. Se construyen diagramas de clases, objetos, casos de uso y secuencia que permiten visualizar la estructura del sistema antes de programarlo. Esta fase promueve el desarrollo de competencias en análisis y diseño de software, al servir como puente entre la teoría y la codificación (F. Fernández & López, 2024).

3. Implementación en Java

   Una vez comprendido el modelo, se procede a su traducción al código Java, aplicando buenas prácticas de programación orientada a objetos. Los ejemplos parten de casos simples y evolucionan hacia sistemas de

complejidad intermedia, fortaleciendo la progresión pedagógica. El uso de entornos de desarrollo integrados como *NetBeans* o *IntelliJ IDEA* complementa la práctica, ofreciendo al estudiante un acercamiento real a las herramientas utilizadas en la industria (García-Molina & Pérez, 2022).

4. Consolidación y Evaluación del Aprendizaje

Finalmente, cada capítulo culmina con una fase de consolidación, que incluye:

- Ejercicios propuestos para reforzar la práctica autónoma.
- Autoevaluaciones con preguntas de selección múltiple que permiten retroalimentar el aprendizaje.
- Proyectos integradores, diseñados bajo la metodología de *aprendizaje basado en problemas (ABP)* y *proyectos (PBL)*, con el fin de promover la resolución de situaciones reales y el desarrollo de competencias en la construcción de software (Ramírez, 2023a).

Estrategia Pedagógica

La estrategia metodológica responde a una secuencia didáctica clara:

1. Comprender los conceptos teóricos.
2. Representar el sistema mediante diagramas UML.
3. Implementar el modelo en Java.
4. Resolver ejercicios y proyectos aplicados.
5. Evaluar el aprendizaje mediante autoevaluaciones y retroalimentación.

El **docente** asume el rol de mediador que guía la práctica, fomenta la reflexión crítica y estimula el trabajo colaborativo; mientras que el **estudiante** adopta una postura activa, investigadora y creativa, centrada en el diseño e implementación de soluciones de software.

SIMBOLOGÍA

RECUERDA:

		Instrucciones para el aprendizaje
	**Sugerencia**	Le indicará apoyos de contenido por parte del profesor
	**Tarea**	Actividades complementarias y de refuerzo
	**Observación**	Recomendaciones del profesor
	**Ejercicios**	Podrá practicar lo que se ha enseñado en el tema estudiado
	**Ejemplos**	Podrá reforzar el conocimiento adquirido
	**Texto Importante**	Resumen de la idea principal del tema

TABLA DE CONTENIDOS

Contenido

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO I: FUNDAMENTOS DE LA PROGRAMACION ORIENTADO A OBJETOS

Objetivos de la unidad

- Definir los principios fundamentales de la programación orientada a objetos mediante sus características, ventajas, desventajas, así como los pilares fundamentales que lo sustentan, para comparaciones con otros paradigmas de programación.
- Identificar los principales entornos integrados de desarrollo utilizados en la *POO* a través de sus características, funcionalidades, técnicas de depuración y pruebas de código para mejorar la productividad en el desarrollo de software orientado a objetos en Java.
- Describir la representación gráfica de sistemas orientados a objetos por medio del *UML* para facilitar la planificación, comunicación y documentación en el desarrollo de software.

En este primer capítulo se revisa los fundamentos esenciales de la *POO*, *IDE* para java y la importancia del modelado de datos mediante *UML*.

Preguntas de enfoque

1. ¿Qué es la programación orientada a objetos y por qué es importante en el desarrollo de software?
2. ¿Cuáles son los cuatro principios fundamentales de la *POO* y cómo se aplican en Java?
3. ¿Cuáles son las ventajas y desventajas de este paradigma en comparación con otros enfoques?
4. ¿Qué herramientas facilitan el desarrollo en *POO* y cómo se configuran adecuadamente?
5. ¿Cómo ayuda el modelado *UML* en la representación de estructuras de software orientado a objetos

"La programación orientada a objetos no es solo una técnica, sino una forma de pensar en soluciones reutilizables y escalables."
Grady Booch, cocreador de UML

1.1 Introducción a la programación orientada a objetos

La *POO* surgió en la década de 1960 con el desarrollo del lenguaje Simula, creado por Ole-Johan Dahl y Kristen Nygaard en el Centro Noruego de Computación en 1967. Simula fue el primer lenguaje en introducir el concepto de clases y objetos, sentando las bases del paradigma de la orientación a objetos (Nygaard & Dahl, 1978).

A lo largo de los años, surgieron varios lenguajes de programación que fueron pioneros en la adopción y desarrollo del paradigma de la *POO*, estableciendo los fundamentos sobre los cuales se han construido los lenguajes modernos.

- Smalltalk (1972-1980): Desarrollado por Alan Kay y su equipo en Xerox PARC, Smalltalk introdujo principios esenciales de la POO, como la herencia, encapsulación y polimorfismo, convirtiéndose en una referencia clave para el desarrollo de software basado en objetos.
- Objective-C (1984): Creado por Brad Cox y Tom Love, este lenguaje combinó la eficiencia del lenguaje C con las características orientadas a objetos de Smalltalk, facilitando el desarrollo de software modular y reutilizable.
- C++ (1983): Diseñado por Bjarne Stroustrup, C++ amplió las capacidades de C al integrar soporte para objetos, consolidándose como un lenguaje fundamental para el desarrollo de software a gran escala, incluyendo sistemas operativos, videojuegos y aplicaciones de alto rendimiento.
- Eiffel (1986): Creado por Bertrand Meyer, Eiffel promovió principios de diseño avanzado, como el contrato de programación (Design by Contract), que permite garantizar la calidad del software a través de reglas estrictas de interacción entre componentes.

Estos lenguajes pioneros sentaron las bases para la evolución de otros como Java (1995) y Python (1991), que popularizaron la programación orientada a objetos en la industria del software, haciéndola accesible y versátil para una amplia variedad de aplicaciones.

Hoy en día, la mayoría de los lenguajes de programación, como C#, Javascript, TypeScript, PHP, Angular, Ionic, etc. Incorporan el paradigma orientado a objetos, consolidándose como el enfoque predominante en el desarrollo de software moderno debido a su capacidad para mejorar la modularidad, reutilización de código y escalabilidad.

Principios de la programación orientada a objetos

En el paradigma de la *POO*, los principios fundamentales giran en torno a la abstracción, encapsulación, la herencia, el polimorfismo. Estos principios

proporcionan un marco estructural que facilita la modularidad y la reutilización de código, permitiendo a los desarrolladores construir sistemas más flexibles y mantenibles(Vera & Vera, 2023).

Ventajas de la programación orientada a objetos

Entre las principales ventajas de la *POO* se encuentran:

- Reutilización de código**:** Los objetos y clases pueden ser reutilizados en diferentes partes del programa o en otros proyectos, lo que reduce el tiempo de desarrollo.
- Mantenibilidad**:** La estructura modular de la *POO* facilita la localización y corrección de errores, así como la implementación de mejoras.
- Escalabilidad**:** La posibilidad de extender clases y crear nuevas funcionalidades permite que el software crezca y se adapte a nuevas necesidades sin requerir reescrituras significativas.
- Modularidad**:** Facilita la división del software en componentes independientes, lo que mejora la organización y mantenimiento del código.

Desventajas de la programación orientada a objetos

A pesar de sus beneficios, la *POO* también presenta algunas desventajas:

- Complejidad**:** El uso excesivo de jerarquías de clases y relaciones entre objetos puede conducir a sistemas difíciles de entender y mantener.
- Rendimiento**:** La abstracción y las capas adicionales pueden introducir sobrecarga, afectando el rendimiento en aplicaciones que requieren alta eficiencia.

1.2 Pilares de la programación orientada a objetos

Figura 1: Pilares de la POO.
Fuente: Elaboración propia.

Según (A. Amahan & C. Sanqui, 2021). La programación orientada a objetos se fundamenta en cuatro pilares esenciales: encapsulamiento, herencia, polimorfismo y abstracción. A continuación, se describe cada uno de estos conceptos fundamentales:

- **Encapsulamiento**: Consiste en restringir el acceso directo a los datos de un objeto y controlar su modificación a través de métodos específicos. Con esta característica, los atributos de una clase permanecen ocultos y solo pueden ser manipulados mediante métodos públicos, privados o protegidos. El principal objetivo de la encapsulación es garantizar la integridad de los datos, asegurando que los usuarios de una clase no necesiten conocer su implementación interna.
- **Herencia**: Es el mecanismo que permite que una clase derive atributos y métodos de otra, promoviendo la reutilización de código y la eficiencia en el desarrollo de software. Gracias a la herencia, se pueden construir jerarquías de clases, facilitando la organización y mantenimiento del código. Existen distintos tipos de herencia en los lenguajes de programación modernos, como herencia simple, múltiple, jerárquica e híbrida, cada una con aplicaciones específicas que optimizan la estructuración del software.
- **Polimorfismo**: Permite que un mismo método o función adopte diferentes comportamientos según el contexto en el que se utilice. En la programación, el polimorfismo se manifiesta a través de la sobrecarga y la sobreescritura de métodos, brindando mayor flexibilidad y escalabilidad a los programas. Esta característica ofrece múltiples ventajas, como la capacidad de que los objetos actúen de distintas maneras sin modificar su estructura base, la simplificación de la lógica del código y la ampliación de la funcionalidad de las aplicaciones.
- **Abstracción**: Este principio permite reducir la complejidad del mundo real mediante el modelado de los aspectos más relevantes de un objeto,

omitiendo detalles innecesarios. Gracias a la abstracción, los desarrolladores pueden centrarse en las características esenciales de cada entidad, facilitando un diseño más claro y eficiente. Se considera el pilar más importante de la *POO*, ya que un adecuado proceso de abstracción es clave para un buen análisis y diseño orientado a objetos.

Recuerda estos cuatro pilares constituyen la base del paradigma de la programación orientada a objetos, proporcionando herramientas fundamentales para el desarrollo de software modular, reutilizable y escalable.

1.3 Entornos integrados de desarrollo (IDE)

En el ámbito del desarrollo de software, los *IDE* son herramientas esenciales que proporcionan a los programadores un conjunto de funcionalidades para escribir, depurar y compilar código de manera eficiente. A continuación, se presentan algunos de los principales IDEs utilizados en 2025, detallando sus características, ventajas y desventajas:

IntelliJ IDEA

Desarrollado por *JetBrains*, *IntelliJ IDEA* es reconocido por su inteligencia de código y su capacidad de autocompletado avanzado, lo que reduce significativamente el tiempo de codificación. Ofrece una integración impecable con sistemas de control de versiones como *Git* y soporta una amplia gama de lenguajes de programación.

Ventajas IntelliJ IDEA

- ✓ Asistencia inteligente en la escritura de código.
- ✓ Integración nativa con herramientas de control de versiones.
- ✓ Soporte para múltiples lenguajes y *frameworks*.

Desventajas IntelliJ IDEA

- ✗ La versión *Ultimate* es de pago, lo que puede ser una barrera para algunos desarrolladores.
- ✗ Requiere una cantidad considerable de recursos del sistema, lo que podría afectar el rendimiento en máquinas menos potentes.

Referencia: JetBrains. (2025). *IntelliJ IDEA*. https://www.jetbrains.com/idea/

Visual studio code

Desarrollado por Microsoft, es un editor de código fuente ligero pero potente, compatible con una amplia variedad de lenguajes de programación. Su ecosistema de extensiones permite a los desarrolladores personalizar y ampliar sus funcionalidades según las necesidades del proyecto.

Ventajas visual studio code

- ✓ Interfaz intuitiva y fácil de usar.
- ✓ Gran biblioteca de extensiones para diversos lenguajes y herramientas.
- ✓ Integración con sistemas de control de versiones como *Git*.

Desventajas visual studio code

- ✗ Al ser ligero, algunas funcionalidades avanzadas requieren extensiones adicionales.
- ✗ Puede consumir una cantidad considerable de memoria con múltiples extensiones activas.

📚 Referencia: Microsoft. (2025). *Visual Studio Code*. https://code.visualstudio.com/

Eclipse

Eclipse es un *IDE* de código abierto ampliamente utilizado, especialmente en el desarrollo con Java. Ofrece una arquitectura de *plugins* que permite a los desarrolladores ampliar sus funcionalidades según las necesidades específicas del proyecto.

Ventajas eclipse

- ✓ Extensible mediante una amplia variedad de *plugins*.
- ✓ Comunidad activa que contribuye al desarrollo y mejora continua del *IDE*.
- ✓ Soporte para múltiples lenguajes de programación.

Desventajas eclipse

- ✗ La instalación y configuración de *plugins* pueden ser complejas.
- ✗ Interfaz menos moderna en comparación con otros *IDEs*.

📚 Referencia: Eclipse Foundation. (2025). *Eclipse IDE*. https://www.eclipse.org/ide/

Apache NetBeans

NetBeans es un *IDE* de código abierto que facilita el desarrollo de aplicaciones en Java, PHP, C++ y otros lenguajes. Ofrece herramientas para la gestión de proyectos, edición de código, diseño de interfaces gráficas y depuración.

Ventajas apache netBeans

- ✓ Interfaz intuitiva y fácil de usar.
- ✓ Soporte nativo para múltiples lenguajes sin necesidad de configuraciones adicionales.
- ✓ Integración con servidores de aplicaciones y bases de datos.

Desventajas apache netBeans

- ✗ Menor cantidad de *plugins* y extensiones en comparación con otros *IDEs*.
- ✗ Interfaz menos moderna y personalizable.

📚 Referencias: Apache Software Foundation. (2025). *Apache NetBeans*. https://netbeans.apache.org/

1.4 Herramientas de lenguaje de modelado unificado

El *UML* es una herramienta clave en el desarrollo de software, utilizada para visualizar, diseñar y documentar sistemas complejos. Existen diversas herramientas de modelado que permiten crear diagramas *UML* con diferentes niveles de funcionalidad, integración y soporte para metodologías de desarrollo.

A continuación, se presentan las principales herramientas de modelado de datos con *UML* en 2025, junto con sus definiciones, ventajas y desventajas.

Enterprise Architect

Enterprise Architect es una herramienta de modelado *UML* desarrollada por *Sparx Systems*. Es ampliamente utilizada en el diseño de software, bases de datos, arquitectura empresarial y

modelado de sistemas embebidos. Ofrece compatibilidad con UML, BPMN, SysML y otras notaciones.

Ventajas enterprise architect

Soporte completo para UML y otros estándares de modelado.

- ✓ Integración con lenguajes como Java, *C++, Python* y bases de datos.
- ✓ Permite la gestión de requisitos y modelado de arquitectura empresarial.
- ✓ Compatible con metodologías ágiles y modelado en tiempo real.

Desventajas enterprise architect

- ✗ Interfaz algo compleja para nuevos usuarios.
- ✗ Es una herramienta de pago con licencias costosas para grandes equipos.
- ✗ Algunas funcionalidades avanzadas pueden requerir complementos adicionales.

📚 **Referencias:** Sparx Systems (2024). *Enterprise Architect - UML & Business Process Modeling*. https://sparxsystems.com

Altova UModel 2025

Es una herramienta de modelado UML que admite los 14 tipos de diagramas UML 2 y ofrece funcionalidades de ingeniería de código, incluyendo generación y reversión de código en Java, C# y Visual Basic. Wikipedia+1XML Soluciones Móviles+1

Ventajas altova umodel

- ✓ Amplia compatibilidad con diversos tipos de diagramas *UML*.
- ✓ Integración con entornos de desarrollo como *Eclipse* y *Visual Studio*.
- ✓ Soporte para ingeniería directa e inversa de código.

Desventajas altova umodel

- ✗ Disponible únicamente para sistemas operativos *Windows*.
- ✗ Es una herramienta de pago, lo que puede ser una barrera para algunos usuarios.

Referencia: Altova (2024). *UModel - UML Software Modeling Tool*. https://www.altova.com/umodel

Lucidchart

Es una aplicación en línea para crear diagramas *UML* y otros tipos de diagramas. Ofrece una interfaz intuitiva basada en la nube que facilita la colaboración en tiempo real.

Ventajas lucidchart

- ✓ Accesible desde cualquier dispositivo con conexión a Internet.
- ✓ Facilita la colaboración en equipo gracias a sus funciones en línea.
- ✓ Ofrece una amplia biblioteca de formas y plantillas.

Desventajas lucidchart

- ✗ Requiere una suscripción para acceder a todas las funcionalidades avanzadas.
- ✗ Dependencia de una conexión a Internet estable para su uso.

Referencia: Lucid Software Inc. (2024). *Lucidchart - Visual Collaboration Suite*. https://www.lucidchart.com

PlantUML

PlantUML es una herramienta de código abierto que permite crear diagramas *UML* a partir de texto descriptivo. Es especialmente útil para desarrolladores que prefieren escribir especificaciones en lugar de utilizar interfaces gráficas.

Ventajas plantUML

- ✓ Ligero y fácil de integrar en otros entornos y herramientas.
- ✓ Soporte para múltiples tipos de diagramas *UML*
- ✓ Gratuito y de código abierto.

Desventajas PlantUML

- ✗ Requiere familiaridad con su lenguaje de marcado para crear diagramas.

- ✗ Carece de una interfaz gráfica intuitiva, lo que puede ser una barrera para algunos usuarios.

Referencias: PlantUML (2024). *PlantUML - Open Source UML tool*. https://plantuml.com

1.5 Lenguaje de modelado unificado UML

Es un lenguaje gráfico de modelado utilizado en ingeniería de software para visualizar, especificar, construir y documentar los componentes de un sistema. *UML* permite representar tanto la estructura estática como el comportamiento dinámico de los sistemas, especialmente aquellos basados en el paradigma de la programación orientada a objetos**.**

¿Por qué es lenguaje de modelado unificado?

- *Unified* (UNIFICADO):

  El aporte de muchos métodos y notaciones independiente de implementaciones, plataformas y lenguajes.

- *Modeling* (MODELADO):

  Los modelos son utilizados en todas las ingenierías

- *Lenguaje* (LENGUAJE):

Cuando existen interacciones humanas, surge la necesidad de comunicación. Para que dicha comunicación sea efectiva, es fundamental que las partes se comprendan mutuamente. Esta comprensión solo es posible si comparten un lenguaje común que facilite la interpretación y transmisión de ideas.

¿Qué es UML?

UML es un lenguaje de modelado, no de programación, permite crear modelos que representan distintos aspectos de un sistema: actores, procesos, clases, interacciones, infraestructura, etc.

UML fue creado a mediados de los años 90 por tres expertos en ingeniería de software orientada a objetos.

Tabla 1: Biografía de los creadores de UML.

Autores	**Biografía**
 Grady Booch	Es un ingeniero de software, científico computacional y autor reconocido internacionalmente por sus contribuciones al desarrollo de la programación orientada a objetos y por ser uno de los creadores del *UML*.
 James Rumbaugh	Es un científico computacional y pionero en la ingeniería de software, reconocido principalmente por ser uno de los creadores del *UML*. Su trabajo ha tenido un impacto profundo en el desarrollo de metodologías para el análisis y diseño orientado a objetos.
 Ivar Jacobson	Es un ingeniero de software, académico y pionero en el desarrollo de metodologías orientadas a objetos, ampliamente reconocido por su papel fundamental en la creación del *UML* y por haber introducido el concepto de casos de uso (*use cases*) en el análisis y diseño de sistemas.

Ten en cuenta que estos tres autores unificaron sus métodos para crear *UML* bajo el auspicio del *OMG* en 1997 y en la actualidad es un estándar internacional mantenido por el *Object Management Group* (*OMG*).

1.6 Principales diagramas de UML

Los diagramas de *UML* se clasifican en dos grandes grupos según su propósito: diagramas estáticos(estructurales) y diagramas dinámicos (de comportamiento). Además, estos diagramas están organizados de acuerdo a diferentes vistas del sistema, que permiten representar el sistema desde múltiples perspectivas. La Figura 2 presenta los principales diagramas del lenguaje de modelado unificado, tanto estáticos como dinámicos.

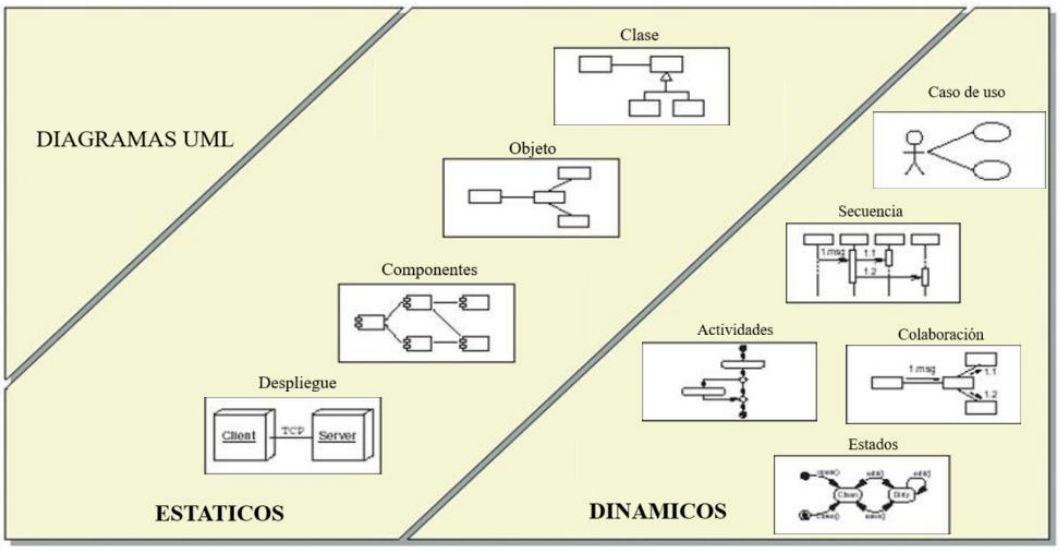


Figura 2: Diagramas del lenguaje de modelado unificado.
Fuente: Elaboración propia.

Diagramas estáticos (estructurales)

Los diagramas estáticos en *UML* muestran la estructura fija del sistema. Es decir, representan los elementos que lo componen (como clases, objetos, paquetes o componentes) y las relaciones entre ellos. Estos diagramas no muestran cómo cambian o se comportan esos elementos a lo largo del tiempo.

Por ejemplo, un diagrama de clases permite ver las clases de un sistema, sus atributos, métodos y cómo se relacionan entre sí.

Tabla 2: Diagramas estáticos de UML.

Diagrama	Descripción
Diagrama de clases	Muestra las clases del sistema, sus atributos, métodos y relaciones (herencia, asociación, etc.).
Diagrama de objetos	Representa instancias concretas de las clases en un momento determinado.
Diagrama de componentes	Representa la organización física de los componentes de software.
Diagrama de despliegue	Muestra la disposición física del hardware y cómo se asignan los componentes del software.
Diagrama de paquetes	Organiza elementos del modelo en grupos o módulos lógicos.

Según (Coronel & Morris, 2019), los diagramas estructurales ayudan a visualizar la arquitectura lógica del sistema, permitiendo entender su organización antes de implementarlo.

Diagramas dinámicos (comportamiento)

Los diagramas dinámi**cos** se enfocan en mostrar cómo se comporta el sistema con el tiempo, cómo interactúan los objetos, cómo fluye la información o cómo cambian los estados de los elementos.

Por ejemplo, un diagrama de casos de uso muestra qué funcionalidades ofrece el sistema y cómo los usuarios (actores) interactúan con él. También hay diagramas como el de secuencia, que muestra el orden en que ocurren las acciones.

Tabla 3: Diagramas dinámicos de UML

Diagramas	Descripción
Diagrama de casos de uso	Muestra las funcionalidades del sistema desde la perspectiva del usuario (actores).
Diagrama de secuencia	Representa la interacción entre objetos a través del tiempo.
Diagrama de comunicación	Similar al de secuencia, pero enfocado en la relación entre objetos.
Diagrama de actividades	Modela flujos de trabajo, decisiones, bifurcaciones y concurrencia.
Diagrama de estados	Representa los estados de un objeto y las transiciones entre ellos.

De acuerdo con el (Object Management Group, 2023), los diagramas de comportamiento permiten modelar los procesos, actividades y eventos del sistema en tiempo real.

Modelado de datos

Es el proceso de analizar, organizar, estructurar y representar gráficamente los datos relevantes de un sistema o dominio de información, con el objetivo de diseñar estructuras lógicas y físicas que permitan su almacenamiento, acceso, integridad y recuperación eficiente.
Este modelado se realiza a través de modelos abstractos que describen:

- Las entidades (objetos del mundo real)
- Los atributos de esas entidades
- Las relaciones entre entidades
- Las reglas de integridad (como claves primarias, claves foráneas, restricciones, etc.)

A continuación, se describen los distintos niveles del modelado de datos, junto con las principales herramientas utilizadas en cada uno de ellos.

Tabla 4: Niveles de modelado de datos.

Nivel	Descripción
Conceptual	Representa qué información es importante sin entrar en detalles técnicos. Usa modelos entidad-relación (ER).
Lógico	Define cómo se estructuran los datos en términos de tablas, atributos, tipos de datos y relaciones lógicas.
Físico	Detalla cómo los datos se almacenan en el sistema de gestión de bases de datos (SGBD). Incluye índices, particiones, tipos físicos.

1.7 Diferencia entre Lenguaje de modelado unificado y modelado de datos

Aunque el *UML* y el Modelado de Datos comparten la idea de representar sistemas mediante diagramas, no son lo mismo. Aquí te explico claramente sus diferencias clave:

Tabla 5: Modelado UML vs modelado de datos

Concepto	Definición
UML	Es un lenguaje gráfico estandarizado utilizado para modelar sistemas de software en general (estructura, comportamiento, interacciones, despliegue, etc.). No se limita a datos.
Modelado de Datos	Es el proceso de definir cómo se organizan, almacenan y relacionan los datos en un sistema, generalmente enfocado en bases de datos.

A continuación, se presenta una tabla comparativa entre el *UML* y el modelado de datos, en la que se detallan aspectos claves como su propósito, los tipos de diagramas utilizados y las herramientas asociadas a cada enfoque.

Tabla 6: Comparativa de lenguaje de modelado unificado y modelado de datos.

Enfoque	Propósito	Diagramas	Herramientas
UML	Modela todo el sistema: clases, actores, procesos, componentes, etc. Enfocado en análisis, diseño y documentación de software. Considera comportamiento, como flujos, estados o interacciones.	Usa 14 tipos de diagramas (casos de uso, clases, secuencia, estados, etc.).	Visual Paradigm, Enterprise Architect, StarUML, IBM Rhapsody.
Modelado de Datos	Modela solo los datos: estructuras, relaciones, integridad, dependencias. Enfocado en diseño de bases de datos (conceptual, lógico y físico). Se concentra en la estructura estática de los datos.	Usa principalmente tres niveles de modelos: - Modelo conceptual (entidad-relación) - Modelo lógico (atributos, claves) - Modelo físico (tablas, índices)	MySQL Workbench, ERwin Data Modeler, Oracle Data Modeler, DB Designer.

1.8 Diagramas de clase en  UML

¿**Qué es una clase**?

Una clase es una estructura que define un conjunto de atributos (propiedades) y métodos (comportamientos) comunes a todos los objetos que pertenecen a esa categoría. Es, en esencia, un modelo o plantilla a partir de la cual se crean instancias individuales conocidas como objetos. Una clase permite encapsular datos y funciones relacionadas, promoviendo la modularidad y la reutilización del código (García-Molina & Pérez, 2022)

Definición: Una clase es una abstracción que agrupa atributos y métodos comunes de un conjunto de objetos, sirviendo como plano para su creación y comportamiento en el sistema (Ramírez, 2023b).

En *UML*, una clase se representa mediante un rectángulo dividido en tres secciones. La parte superior contiene el nombre de la clase, la sección central presenta los atributos o datos, y la inferior muestra los métodos o comportamientos. Además, tanto los atributos como los métodos pueden incluir especificadores de acceso que determinan su nivel de visibilidad u ocultamiento dentro del sistema.

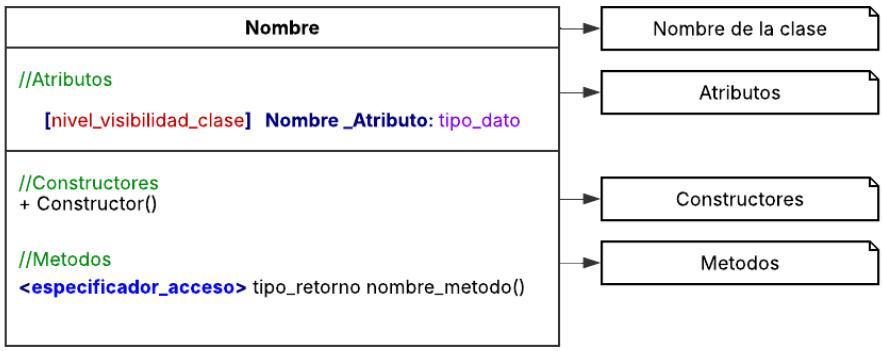


Figura 3: Estructura de un diagrama de clase.
Fuente: Elaboración propia.

El nombre de la clase: representa la identidad del tipo de objeto que se desea modelar en un programa. Este nombre debe ser claro, descriptivo y estar escrito en singular, ya que define una plantilla general a partir de la cual se crean objetos concretos (instancias). Por ejemplo, en una clase llamada Perro, el nombre indica que todos los objetos creados a partir de esa clase (como perro1, perro2, etc.) representarán perros individuales con características y comportamientos comunes.

Los atributos o datos: representan las características de un objeto y se definen como variables con un tipo específico, como int (entero), double (decimal) o char (carácter). Estos atributos permiten describir las propiedades esenciales de objetos reales que tienen algo en común.

Un constructor: es un método especial dentro de una clase que se llama automáticamente cuando se crea (instancia) un objeto. Su principal propósito es inicializar los atributos del objeto, es decir, asignarles valores cuando el objeto comienza a existir.

En programación orientada a objetos, los niveles de visibilidad de la clase y especificadores de acceso determinan el nivel de visibilidad u ocultamiento de los atributos y métodos de una clase. Estos se representan con símbolos específicos:

- **private (-)**: El atributo o método solo puede ser accedido desde dentro de la misma clase. Es el nivel más restrictivo.
- **protected (#)**: Permite el acceso desde la propia clase, clases internas y también desde clases hijas (heredadas).
- **package** (sin símbolo): Permite el acceso desde cualquier clase dentro del mismo paquete, pero no desde fuera del paquete.
- **public (+)**: El atributo o método es accesible desde cualquier clase del programa, sin restricciones.

Los métodos: son las acciones o comportamientos que puede realizar un objeto. A través de los métodos, es posible modificar los valores de los atributos de la clase o comunicarse con otros objetos enviándoles mensajes (llamadas a métodos).

1.9 Implementación de una clase

Caso de estudio: Calcular el pago por consumo de electricidad.

Una empresa de electricidad cobra el servicio a sus clientes de acuerdo a la siguiente escala:

- $ 0,10 por kilovatio por los primeros 100 kilovatios de consumo.
- $ 0,12 por kilovatio por el consumo de 101 a 200 kilovatios.
- $ 0,15 por kilovatio por el consumo de 201 kilovatios en adelante.

Crear una clase para que dado el consumo en kilovatios de un determinado cliente calcule e informe el total a pagar por el mismo.

- Ejemplo 1: Si se ingresa un consumo de 55 kilovatios, entonces el programa calculará: $ 0,10 x 55= $ 5,50
- Ejemplo 2: Si se ingresa un consumo de 125 kilovatios, entonces el programa calculará: $ 0,10 x 100 + $ 0,12 x 25=$ 13
- Ejemplo 3: Si se ingresa un consumo de 250 kilovatios, entonces el programa calculará: $ 0,10 x 100 + $ 0,12 x 100 + $ 0,15 x 50 = $ 29,50.

Ejercicio #1: Calcular el pago por consumo energético.

Crear un diagrama de clase UML del enunciado del problema, definir el nombre de la clase, atributos, constructor y métodos.

A continuación, se realizar la abstracción de la entidad Electricidad.

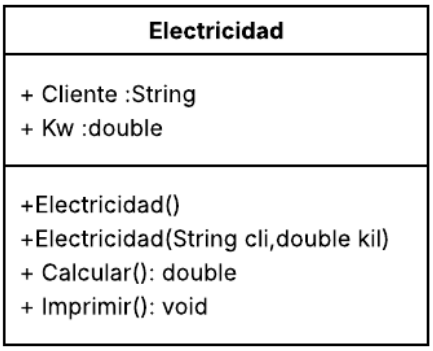


Figura 4: Diagrama de clase Electricidad.
Fuente: Elaboración propia.

¿**Qué es un diagrama de clase**?

Un diagrama de clases es una representación gráfica que muestra la estructura de un sistema orientado a objetos, destacando las clases, sus atributos, métodos y las relaciones entre ellas. Es uno de los diagramas más importantes del *UML* y se utiliza en las etapas de análisis y diseño de software para visualizar cómo interactúan las diferentes partes de un sistema (Martínez & Herrera, 2021).

Definición: Un diagrama de clases es una representación visual de las clases de un sistema, sus atributos, métodos y las relaciones entre ellas, utilizado para modelar la estructura estática del software (P. Fernández & López, 2024).

1.10 Relación entre clase y diagrama de clases

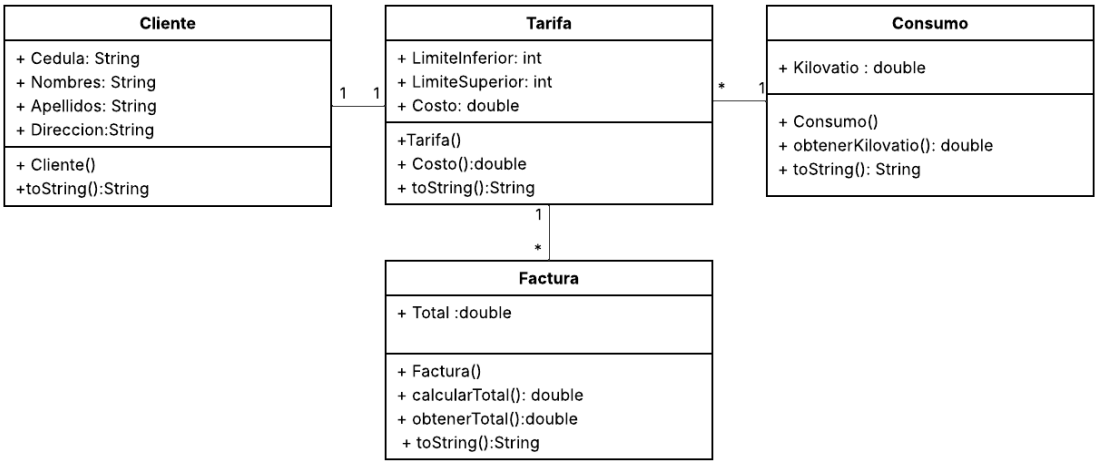


Figura 5: Relación entre diagramas de clase.
Fuente: Elaboración propia.

La relación entre una clase y un diagrama de clases es de representación y análisis: mientras la clase es un componente del sistema de software que define estructura y comportamiento, el diagrama de clases representa gráficamente esa información para comprender, documentar y diseñar el sistema. En otras palabras, el diagrama de clases muestra cómo las clases interactúan entre sí y cómo se organiza el sistema desde una perspectiva estática.

Según (Gómez & Estrada, 2023). El diagrama de clases ofrece una visión estructural del sistema mediante la representación de las clases, permitiendo identificar su interacción, composición y dependencias, aspectos esenciales para un diseño de software efectivo. En la Figura 5 se ilustran las asociaciones entre las distintas clases, destacando cómo se comunican y comparten información.

RESUMEN DEL CAPITULO

En este capítulo, se abordaron los fundamentos de la Programación Orientada a Objetos, incluyendo sus principios, ventajas, desventajas, pilares fundamentales, principales herramientas para el desarrollo de software y modelado de datos. Se exploró cómo la *POO* permite el desarrollo de software modular, reutilizable y escalable, lo que facilita el mantenimiento y la evolución de los sistemas. Además, se discutieron conceptos clave como el lenguaje de modelado unificado y modelado de datos, estableciendo una base sólida para los capítulos posteriores.

AUTOEVALUACIÓN

En las preguntas de selecciona múltiple un solo literal es la respuesta correcta

1. ¿Cuál fue el primer lenguaje en introducir el concepto de clases y objetos?
    a) Java.
    b) Smalltalk
    c) Simula
    d) C++

2. ¿Cuál de las siguientes afirmaciones sobre la POO es falsa?
    a) Permite la reutilización del código.
    b) Facilita la organización del software.
    c) No es compatible con el paradigma de programación estructurada.
    d) Se basa en la interacción de objetos.

3. ¿Cuál de los siguientes es un pilar de la programación orientada a objetos?
    a) Modularidad.
    b) Encapsulamiento.
    c) Lenguaje.
    d) Sintaxis.

4. El encapsulamiento permite:
    a) Ejecutar código más rápido.
    b) Compartir directamente todos los datos.
    c) Restringir el acceso directo a los datos.
    d) Evitar la creación de objetos.

5. El mecanismo que permite que una subclase adquiera atributos y métodos de una superclase se conoce como:
    a) Polimorfismo.
    b) Herencia.
    c) Encapsulamiento.
    d) Abstracción.

6. El polimorfismo permite:

a) Reutilizar el código sin cambios.
b) Dividir el código en paquetes.
c) Ocultar atributos.
d) Usar un mismo método con diferentes comportamientos.

7. La abstracción en POO sirve para:
   a) Compilar código más rápido.
   b) Modelar solo el hardware.
   c) Centrarse en lo esencial de un objeto.
   d) Escribir código procedural

8. ¿Cuál es una ventaja principal de la programación orientada a objetos?
   a) Complejidad elevada.
   b) Escalabilidad.
   c) Baja mantenibilidad.
   d) Poca reutilización de código.

9. ¿Qué representa un diagrama de clases en UML?
   a) La secuencia de acciones.
   b) La interfaz gráfica.
   c) Los flujos de trabajo.
   d) Las clases, atributos, métodos y relaciones.

10. ¿Qué expertos crearon el lenguaje UML?
    A) Alan Turing, Donald Knuth, Ken Thompson
    B) Dahl, Nygaard y Meyer
    C) Booch, Rumbaugh y Jacobson
    D) Gosling, Stroustrup y Kay

REFERENCIA BIBLIOGRÁFICA

Amahan, P., & C. Sanqui, R. (2021). Syntax to Syntax: Assessment of Orthogonality in the Design of Object-oriented Programming Languages using Code Listing Method. Proceedings of the 2021 4th International Conference on Software Engineering and Information Management, 52-55. https://doi.org/10.1145/3451471.3451480.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). The Unified Modeling Language User Guide, 2nd Edition. Addison-Wesley Professional.

Coronel, C., & Morris, S. (2019). Database systems: Design, implementation, & management (13.a ed.). Cengage Learning.

Dennis, A., Wixom, B., & Tegarden, D. (2015). Systems Analysis and Design: An Object-Oriented Approach with UML (5.a ed.). Wiley.

Fernández, F., & López, M. (2024). Diseño y modelado de sistemas con UML y Java. Alfaomega.

Fernández, P., & López, M. (2024). Diseño y modelado de sistemas con UML y Java. Editorial Alfaomega.

García-Molina, L., & Pérez, S. (2022). Fundamentos de programación orientada a objetos con Java. McGraw-Hill Education.

Gómez, R., & Estrada, C. (2023). Ingeniería de Software: Modelado y análisis de sistemas. Ra-Ma.

Larman, C. (2005). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3.a ed.). Prentice Hall.

Martínez, V., & Herrera, D. (2021). UML y Patrones: Una introducción al análisis y diseño orientado a objetos. Pearson Educación.

Mora, J., & García, D. (2022). Estrategias de aprendizaje activo en la enseñanza de la programación.: Vol. 17(2). Revista Iberoamericana de Tecnología Educativa.

Nygaard, K., & Dahl, O.-J. (1978). The development of the SIMULA languages. En R. L. Wexelblat (Ed.), History of programming languages (pp. 439-480). ACM.

https://doi.org/10.1145/800025.1198392

Object Management Group. (2017). OMG Unified Modeling Language (OMG UML) (2.a ed.). https://www.omg.org/spec/UML/2.5.1/

Object Management Group. (2023). OMG Unified Modeling Language (UML). https://www.omg.org/spec/UML

Pressman, R. (2020). Software Engineering: A Practitioner's Approach (9.a ed.). McGraw-Hill Education.

Ramírez, J. (2023a). Programación orientada a objetos: Conceptos y aplicaciones prácticas (Cengage Learning.).

Ramírez, J. (2023b). Programación orientada a objetos: Conceptos y aplicaciones prácticas. Cengage Learning.

Sierra, K., & Bates, B. (2022). OCA/OCP Java SE 8 Programmer Certification (2.a ed.).

Vera, J., & Vera, J. (2023). El papel de la programación orientada a objetos en el desarrollo de software sostenible y escalable. Universidad Ciencia y Tecnología, 27(121), 85-94. https://doi.org/10.47460/uct.v27i121.757

CAPÍTULO II: CLASES Y OBJETOS

OBJETIVOS DE LA UNIDAD

- Comprender los principios fundamentales de la programación orientada a objetos aplicados al modelado de clases y objetos.
- Aplicar el *UML* para representar diagramas de clases, sus atributos, métodos y relaciones.
- Crear clases con atributos privados mediante el encapsulamiento utilizando modificadores de acceso, métodos de acceso (*getters* y *setters*), y buenas prácticas de codificación, para qué garantice la seguridad de los datos.

En este segundo capítulo se abordará de manera detallada el proceso de creación y gestión de clases y objetos en el lenguaje Java. Se estudiará cómo declarar clases, definir sus atributos y métodos, así como el uso de diferentes tipos de constructores y su sobrecarga. Además, se explorará el ciclo de vida de los objetos, incluyendo la gestión de la memoria y el manejo de referencias. Finalmente, se explicarán los modificadores de acceso y se destacarán las mejores prácticas para aplicar el encapsulamiento de forma efectiva en el desarrollo de software orientado a objetos.

Preguntas de enfoque

- ¿Qué ventajas ofrece *UML* en la programación orientada a objetos?
- ¿Cómo se representa una clase y sus relaciones en un diagrama *UML*?
- ¿Cuál es el proceso para convertir un modelo *UML* en código Java?
- ¿Por qué es importante identificar correctamente los atributos y métodos de una clase?
- ¿Qué es el encapsulamiento y cuando utilizarlo?

La verdadera comprensión de la programación orientada a objetos no comienza al escribir código, sino al aprender a modelar correctamente la realidad con clases y objetos. Modelar es pensar antes de programar.

Joffre Cartuche Calva

2.1 Introducción a las clases y objetos

En la *POO*, los programas se construyen a partir de objetos, que representan cosas reales o ideas abstractas. Cada objeto pertenece a una clase, que funciona como un modelo o plantilla que define sus características (llamadas atributos) y sus comportamientos (llamados métodos).

Imaginemos el siguiente ejemplo: mi perro **Peluche**. Es un objeto que pertenece a la clase **Perro**. Todos los perros comparten ciertos atributos, como un nombre, una raza, una fecha de nacimiento, un color y un peso. En el caso de mi perro, su nombre es Peluche, su raza es *Shih-tzu*, a partir de la fecha de nacimiento se puede calcular la edad de tres años y su color es blanco con rayas crema, su peso es de 5.4 kilos, vive en mi casa y es parte de la familia.

Procesos de Abstracción e Instanciación

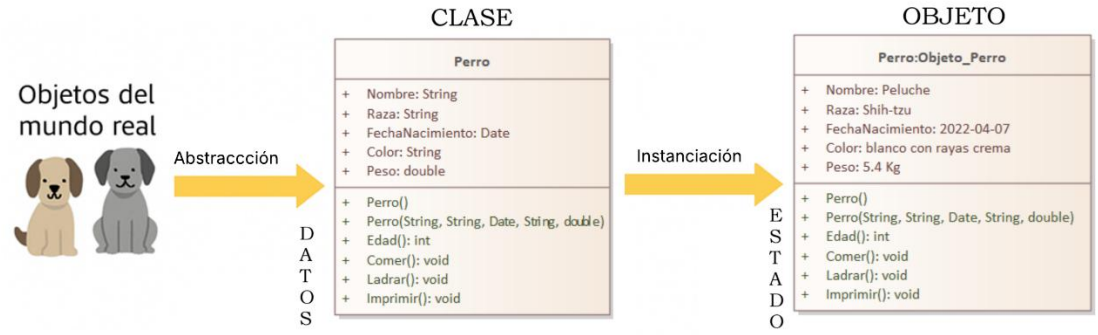


Figura 6: Diagrama de clase y objetos de la entidad Perro.
Fuente: Elaboración propia.

Abstracción

Es el proceso mediante el cual identificamos las características y comportamientos comunes de los objetos del mundo real para representarlos en un modelo. En programación orientada a objetos, ese modelo es la clase.

Por ejemplo, al observar distintos perros, notamos que todos tienen características similares como nombre, raza, fecha de nacimiento, color y peso. También comparten comportamientos, como comer, ladrar. Con base en estas similitudes, creamos la clase **Perro**, que incluye estos **atributos** y **métodos**

comunes. La clase funciona como una plantilla que nos permite construir múltiples perros con sus propios datos.

Instanciación

La instanciación es el proceso de crear un objeto concreto a partir de una clase. Al instanciar, se asigna espacio en memoria para guardar los valores particulares de ese objeto y se habilitan los métodos que puede ejecutar.

En la imagen, a partir de la clase **Perro**, se crea un objeto llamado **Peluche**, con atributos específicos como:

- Nombre: Peluche
- Raza: Shih-tzu
- Fecha de nacimiento: 2022-04-07
- Color: blanco con rayas crema
- Peso: 5.4 kg

Este objeto puede realizar las mismas acciones definidas en la clase, como Comer()**,** Ladrar() o Imprimir(), pero con sus propios datos. A continuación, representaremos gráficamente estos datos mediante *UML*:

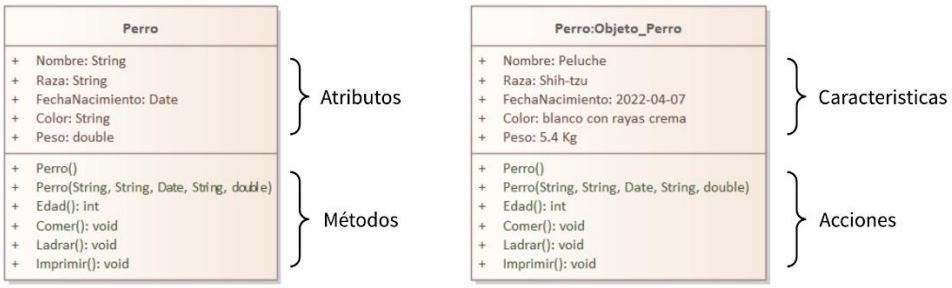


Figura 7: Diagrama de la clase y objeto Perro en UML.
Fuente: Elaboración propia.

La clase Perro es una entidad propia que tiene sus atributos y métodos, pero bien vamos a explicar más detalladamente:

Atributos

En programación orientada a objetos, los atributos representan las características que describen a un objeto. Son variables que almacenan la información o estado del objeto.

La clase Perro tiene los siguientes atributos:

- Nombre: Indica cómo se llama el perro (por ejemplo, "Peluche").
- Raza: Especifica la raza del perro (por ejemplo, "Shih-tzu").
- FechaNacimiento: Señala la fecha cuando nació, por ejemplo, años-mes-día, 2022-04-09).
- Color: Describe el color del perro (por ejemplo, "blanco con rayas crema").
- Peso: Indica cuanto es el peso actual (por ejemplo, 5.2 kilos)

Recuerda: Los atributos responden a la pregunta **"¿Qué es o qué tiene el objeto?"**.

Constructores

Los constructores son métodos especiales que llevan el mismo nombre de la clase, no retornan ningún valor y se invocan automáticamente al momento de crear o instanciar un objeto.

- **Perro()**: Es un constructor sin parámetros. Sirve para crear un perro vacío, es decir, sin establecer aún sus datos como nombre, raza, color, etc.
- **Perro(**String nom, String raz, Date fec, String col, double pes**)**: Es otro constructor, pero este permite crear un perro estableciendo todos sus datos desde el principio: nombre, raza, fecha de nacimiento, color y peso.

Métodos

Los métodos representan las acciones o comportamientos que un objeto puede realizar. Son funciones dentro de la clase que permiten que los objetos realicen tareas o modifiquen su información.
La clase Perro incluye varios métodos que definen lo que el perro puede hacer o cómo se comporta. A continuación, se describen de forma sencilla:

- **Edad()**: Este método calcula la edad actual del perro a partir de su fecha de nacimiento y el año actual. Devuelve un número entero que representa la edad en años.
- **Comer()**: Este método simula que el perro come. Puede funcionar considerando el peso del perro para decidir la cantidad de alimento.
- **Ladrar()**: Este método hace que el perro "ladre", simulando este comportamiento.

- **Imprimir()**: Este método muestra en pantalla toda la información del perro, como su nombre, raza, color, fecha de nacimiento y peso.

Recuerda: Los métodos responden a la pregunta **"¿Qué puede hacer el objeto?"**.

Luego de tener el molde o plantilla de la clase Perro, creamos algo específico y obtenemos un objeto(Objeto_Perro) y corresponde a un perro real con características específicas:

- **Nombre:** Peluche
- **Raza:** Shih-tzu
- **Fecha de Nacimiento:** 2022-04-09
- **Color:** Blanco con rayas crema
- **Peso:** 5.2 Kilos

Estos datos rellenan los atributos que definimos en la clase Perro.

2.2 Clases

Para (Moreno Pérez, 2015) las clases son los moldes de los cuales se generan los objetos. Los objetos se instancian y se generan, con lo cual "instancia" y "objeto" son sinónimos.
De igual forma (Mazón Olivo, 2015). Una clase es un tipo de dato definido por el programador que incluye la estructura de datos y las operaciones asociadas con esos datos.

En otras palabras, podemos decir que una clase es simplemente un modelo o plantilla que se utiliza para describir uno o más objetos del mundo real con similares atributos y comportamiento

Para la creación de una clase se utiliza la palabra reservada class.

```
[nivel_visibilidad_clase] class NombreClase
{
   // -Atributos de la clase públicos, protegidos o privados-
   [especificadores_acceso_clase] tipo_dato Nombre_Atributo;

    // -Métodos de la clase-

    //Definición de constructores

    //Definición de métodos
```

```
}
```

El **nivel de visibilidad de una clase** define quién puede ver o utilizar esa clase en un proyecto de programación.

Los niveles de visibilidad principales son:

- **public**: La clase puede ser utilizada desde cualquier lugar del programa.
  Ejemplo:
  ```
  public class Perro{
    // Código de la clase
  }
  ```

- **default** (sin escribir nada): Si no se escribe ningún nivel (no se pone ni public ni private ni protected), la clase será visible solo dentro del mismo paquete.
  Ejemplo:
  ```
  class Perro{
    // Código de la clase
  }
  ```

Es importante saber que, en lenguajes como Java, no puedes declarar una clase como private o protected directamente. Solo public o dejarla sin especificador (default).

Los **especificadores de acceso** controlan quién puede ver o modificar los atributos (variables) o métodos (funciones) de la clase. Los principales son:

- **public**: El atributo o método puede ser accedido desde cualquier parte del programa.
- **private**: El atributo o método solo puede ser accedido dentro de la misma clase. (Protege la información).
- **protected**: El atributo o método puede ser accedido dentro de la misma clase, sus subclases y el mismo paquete.
- **(default)** (sin escribir nada): Si no se coloca nada, el atributo o método será accesible solo dentro del mismo paquete.

Ejemplo de la clase perro con el nivel de visibilidad de una clase y sus especificadores de acceso públicos.

```
public class Perro {
```

```
    /// -Atributos de la clase públicos
    public String Nombre;
    public String Raza;
    public Date FechaNacimiento;
    public String Color;
    public double Peso;
    public double color;

    // -Métodos de la clase-

    //Definición de constructores

    //Definición de métodos

}
```

2.3 Constructores

Su función principal es reservar memoria para el nuevo objeto e inicializar sus atributos. Aunque los constructores no se heredan en la programación orientada a objetos, una clase puede sobrecargar constructores, es decir, definir varios constructores diferenciados por la cantidad o el tipo de parámetros que reciben, permitiendo crear objetos de diversas formas según las necesidades.

Constructor vacío o por defecto: Es un constructor que no recibe ningún parámetro. Se utiliza para crear objetos asignándoles valores iniciales dentro del mismo constructor o dejando sus atributos con los valores por defecto según su tipo de dato. Por ejemplo, en Java, los números se inicializan en 0 y los objetos en null si no se asigna otro valor.

```
    //Definición de constructores
    public Perro()
    {

    }
```

Constructor parametrizado: Es un constructor que recibe parámetros, ya sean valores primitivos (como int, double, char) o referencias a otros objetos. Se usa para dar valores personalizados a los atributos de un objeto en el momento de su creación.

```
//Definición de constructores
public Perro(String nom, String raz, Date fec, String col,double pes) {
      this.Nombre = nom;
```

```
        this.Raza = raz;
        this.FechaNacimiento = fec;
        this.Color = col;
        this.Peso=pes;

    }
```

Constructor copia: Es un constructor que recibe como parámetro un objeto de la misma clase**.** Su función es crear un nuevo objeto copiando exactamente los datos de otro objeto ya existente. Es útil cuando quieres duplicar un objeto con los mismos valores.

```
//Definición de constructores
    public Perro(Perro obp) {
        this.Nombre = obp.Nombre;
        this.Raza = obp.Raza;
        this.FechaNacimiento = obp.FechaNacimiento;
        this.Color = obp.Color;
        this.Peso=obp.Peso;
    }
```

2.4 Métodos

Los métodos son **acciones** o **comportamientos** que los objetos pueden realizar sobre sus propios datos. Es decir, son funciones que se definen dentro de una clase y que permiten que el objeto haga algo, como mostrar información, calcular valores o modificar sus atributos, estas funciones están definidas dentro de la clase mediante la siguiente sintaxis:

```
<especificador_acceso> tipo_retorno nombre_metodo(tipo_parametro
param1,..)
{
  // Instrucciones de código
  return valor_retorno;
}
```

```
//Definición de métodos
public int Edad()
{
    Calendar cal= new GregorianCalendar();
    cal.setTime(FechaNacimiento);
    Calendar hoy = new GregorianCalendar();
    hoy.setTime(new Date());
    int anios = hoy.get(Calendar.YEAR) - cal.get(Calendar.YEAR);
    if(hoy.get(Calendar.MONTH) < cal.get(Calendar.MONTH)||
```

```
    hoy.get(Calendar.MONTH) == cal.get(Calendar.MONTH)&&
        hoy.get(Calendar.MONTH)< cal.get(Calendar.MONTH)))
  {
      anios--;
  }
    return anios;
}
public void Ladrar()
{
  JOptionPane.showMessageDialog(null, "Guao..Guao");
}
```

 "Es recomendable comentar tu código para mejorar la legibilidad".

En programación orientada a objetos, los métodos pueden clasificarse en dos tipos, al igual que en la programación estructurada:

- Procedimientos: son métodos que **no devuelven ningún valor**, es decir, su tipo de retorno es void. Se utilizan para ejecutar acciones, como mostrar información en pantalla o modificar atributos del objeto.
- Funciones: son métodos que **sí devuelven un valor** después de ejecutar una operación. Estos métodos especifican un tipo de dato como valor de retorno, como int, double o String, y son útiles para realizar cálculos o devolver información del objeto.

Ambos tipos permiten definir el comportamiento de los objetos en un programa orientado a objetos.

2.5 Objetos o entidades reales en la POO

En la *POO*, los objetos representan entidades del mundo real. Estos pueden clasificarse en dos categorías: objetos físicos y objetos lógicos. Comprender esta clasificación es fundamental para diseñar sistemas de software que reflejen con fidelidad los elementos y procesos del entorno (Booch et al., 2005).

Figura 8: Objetos físicos y lógicos.
Fuente: Elaboración propia.

Objetos Físicos

Son entidades tangibles, es decir, que se pueden ver o tocar. Se utilizan en el modelado para representar elementos reales que forman parte del sistema. Por ejemplo, en un sistema de gestión hospitalaria, algunos objetos físicos son: pacientes, camillas, enfermeras, Habitaciones, Termómetros, Ambulancias.

Estos elementos permiten establecer relaciones claras entre las entidades reales y su representación digital (Alan & Haley Wixom, 2020).

Objetos Lógicos

Son entidades intangibles, que no existen físicamente, pero son necesarias para representar información o comportamiento dentro del sistema. En el mismo sistema de gestión hospitalaria, algunos objetos lógicos incluyen: • Historia clínica electrónica, formulario de ingreso del paciente, agenda de turnos, reportes médicos, Interfaz gráfica del sistema, Algoritmo de asignación de camas

Estos objetos permiten organizar, procesar y presentar datos esenciales para la toma de decisiones médicas (Larman, 2020).

2.6 Objetos

Un objeto en la *POO* representa una instancia concreta de una clase que encapsula datos (atributos) y comportamientos (métodos). Los objetos permiten modelar elementos del mundo real dentro del software de forma estructurada y reutilizable (Sierra & Bates, 2022).

Un objeto es algo tangible o visible, algo que puede ser entendido intelectualmente, algo sobre lo que puede ser dirigido un pensamiento o una acción.

Creación de objetos en java

Instanciar un objeto en Java significa crear una copia basada en el modelo definido por una clase, utilizando el operador new y un constructor. Esta acción reserva memoria y habilita el uso de los métodos y atributos del objeto (Deitel & Deitel, 2022).

```
Perro Objeto_Perro= new Perro("Peluche","Shih-tzu","2022-04-09","blanco con rayas crema",5.2 );
```

Ciclo de Vida de un Objeto

El ciclo de vida de un objeto en Java consta de cuatro fases:

1. Creación, mediante instanciación.
2. Uso, mediante llamadas a métodos o acceso a atributos.
3. Inaccesibilidad, cuando ya no hay referencias al objeto.
4. Recolección de basura, en la que la *JVM* libera automáticamente la memoria del objeto (Bloch, 2018).

Primero se crea el objeto cuando usamos la palabra clave ***new*** junto con un **constructor**. Luego, el objeto vive mientras es utilizado, es decir, mientras existe una o más referencias que lo apuntan y permiten acceder a sus datos y métodos. Cuando ya no hay ninguna variable que lo referencie, el objeto se vuelve inaccesible. En esta etapa, aunque todavía existe en la memoria, ya no puede ser usado. Finalmente, el *Recolector de basura (Garbage Collector)* de Java se encarga de eliminarlo automáticamente para liberar espacio en la memoria. Este proceso permite que la gestión de memoria sea más segura y eficiente, sin necesidad de que el programador elimine objetos manualmente.

Para el modelado de datos en *UML*, este libro empleará la herramienta *Enterprise Architect* en su versión 17. La implementación del código se desarrollará en el lenguaje de programación Java, utilizando como entorno de desarrollo integrado (*IDE*) *Apache NetBeans* 25.

La captura de datos se realizará a través de formularios gráficos construidos con *JFrame Form* en un entorno *Windows*, evitando el uso de la consola. Para la presentación de información al usuario, se utilizará la clase *JOptionPane*,

específicamente el método *showMessageDialog()*, lo que permitirá generar ventanas emergentes de manera sencilla y efectiva.

2.7 Implementación de objetos

Caso estudio: Clase perro con sus principales atributos públicos.

Ejercicio #2: Implementación de una clase Perro

Crear un proyecto con el nombre 02_IntroduccionPOO, para la creación de la clase Perro, con todos los atributos y métodos que se presentan en el diagrama de la clase Perro. Figura 9.

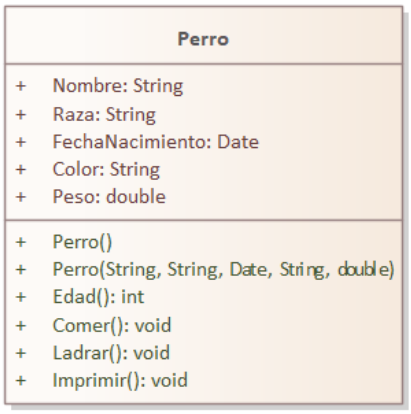


Figura 9: Diagrama de clase Perro con atributos públicos.
Fuente: Elaboración propia.

Implementación del programa en java

1. Crear una nueva aplicación en Java *With Maven – Java Application*

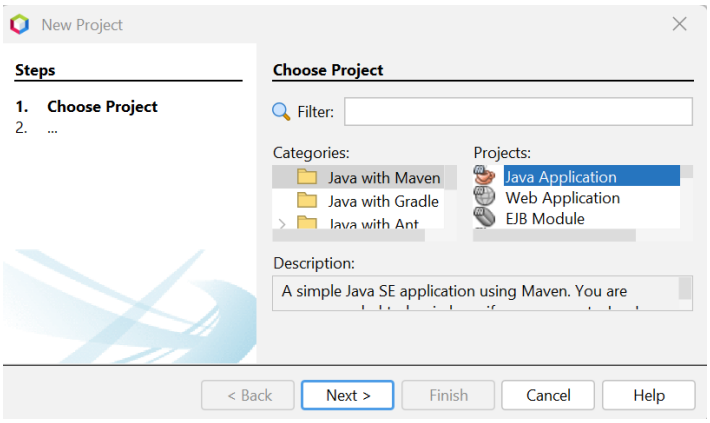


Figura 10: Creación de un proyecto en java con Maven.
Fuente: Elaboración propia.

2. Crear un proyecto con el nombre 01_IntroduccionPOO, en la opción *Package* escribe Entidades.

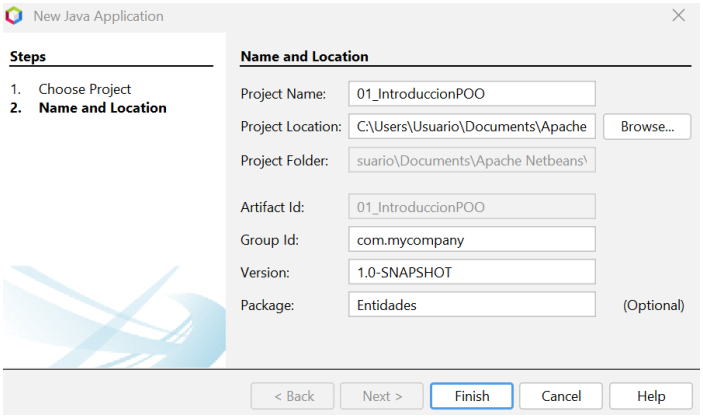


Figura 11: Creación de un proyecto en java.
Fuente: Elaboración propia.

3. Crear un Java *Package* con el nombre de Entidades dentro de este paquete crea una clase Perro. Luego crear otro Java *Package* con el nombre de Formularios dentro del *package* un *JFrame Form* con el nombre de frmPerro.

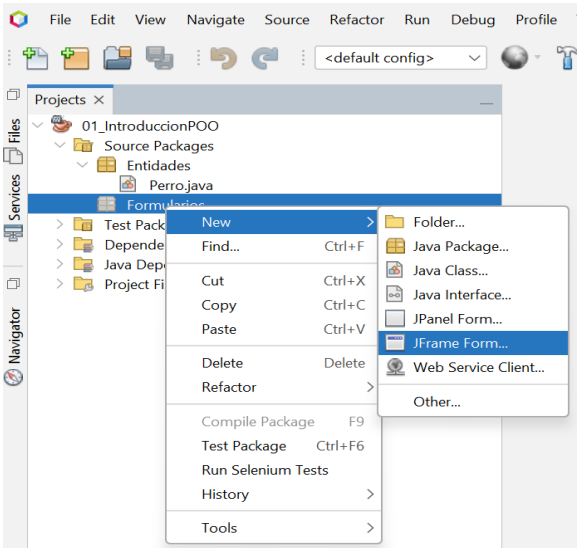


Figura 12: Creación de package y clases.
Fuente: Elaboración propia.

4. Creación de la clase Perro con atributos públicos.

```
public class Perro
{
   /// -Atributos de la clase públicos
   public String Nombre;
   public String Raza;
```

```
  public Date FechaNacimiento;
  public String Color;
  public double Peso;
  //Definición de constructores
  public Perro() {
  }

  //Definición de metodos
  public int Edad()
  {
     Calendar cal= new GregorianCalendar();
     cal.setTime(FechaNacimiento);
     Calendar hoy = new GregorianCalendar();
     hoy.setTime(new Date());
     int anios = hoy.get(Calendar.YEAR) - cal.get(Calendar.YEAR);
     if(hoy.get(Calendar.MONTH) < cal.get(Calendar.MONTH)||
     (hoy.get(Calendar.MONTH) == cal.get(Calendar.MONTH)&&
           hoy.get(Calendar.MONTH)< cal.get(Calendar.MONTH)))
     {
        anios--;
     }
     return anios;
  }
  public void Comer()
  {
     String men="";
     if(Peso <10)
     {
       men="Necesitar entre 30 y 130 gramos de alimento seco por día";
     }
     if(Peso >= 10 && Peso <20)
     {
       men="Necesitar entre 180 y 320 gramos de alimento seco por día";
     }
     if(Peso >= 20 && Peso <30)
     {
       men="Necesitar entre 320 y 440 gramos de alimento seco por día";
     }
     JOptionPane.showMessageDialog(null, men);

  }
  public void Ladrar()
  {
      JOptionPane.showMessageDialog(null, "Guao..Guao");
  }

public void Imprimir()
```

```
  {

        JOptionPane.showMessageDialog(null,"Datos del Perro :" +
                this.Nombre +"\n"+ "Raza :" + this.Raza +"\n"+
        "Edad :" + Edad() +"\n"+ "Color :" + this.Color);
  }

}
```

5. Diseño del formulario frmPerro

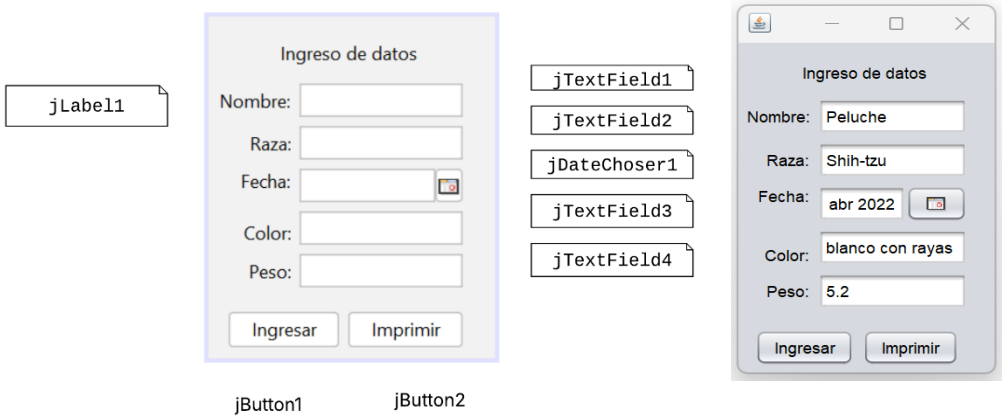


Figura 13: Diseño del formulario JFrame frmPerro.
Fuente: Elaboración propia.

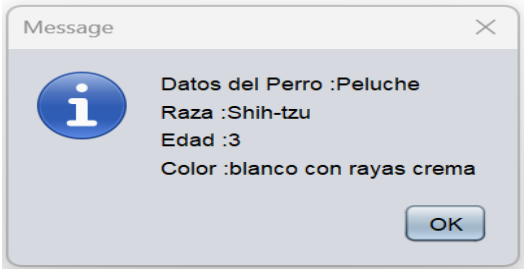
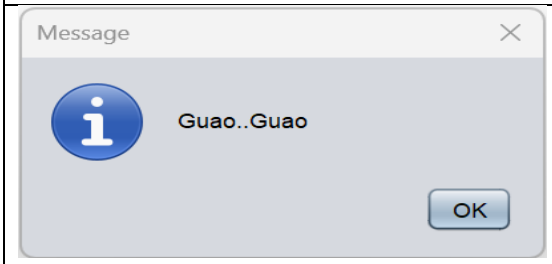
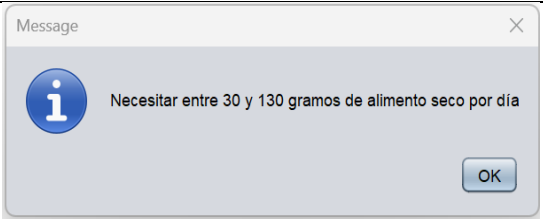
6. Implementación del *JFrame frmPerro*.

```
package Formularios;
import Entidades.Perro;
import java.util.Date;

public class frmPerro extends javax.swing.JFrame {

  /**
   * Creates new form frmPerro
   */
  public frmPerro() {
     initComponents();
  }

  // Variables Globales
  Perro Objeto_Perro;
```

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{
      // TODO add your handling code here:
      String nom=jTextField1.getText();
      String raz=jTextField2.getText();
      Date fn=jDateChooser1.getDate();
      String col= jTextField3.getText();
      double pes= Double.parseDouble(jTextField4.getText());
      Objeto_Perro=new Perro(nom,raz,fn,col,pes);

 }

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)
{
      // TODO add your handling code here:
      Objeto_Perro.Imprimir();
      Objeto_Perro.Ladrar();
      Objeto_Perro.Comer();
 }
}
```

7. Salidas del sistema.

Tabla 7: Presentación de resultados de la clase Perro.

Interfaz de salida	Llamada a metodos de la clase Perro
	Objeto_Perro.Imprimir();
	Objeto_Perro.Ladrar();
	Objeto_Perro.Comer();

2.8 Especificadores de acceso

Los modificadores de acceso o especificadores de visibilidad en Java determinan qué clases, métodos o atributos pueden ser accedidos desde otras clases. Son una parte fundamental de la encapsulación, uno de los pilares de la *POO*, ya que permiten proteger la información interna de una clase y controlar su nivel de exposición al resto del sistema.

Encapsulamiento y Visibilidad

Significa ocultar los detalles internos de una clase y exponer solo aquello que es necesario. Este principio promueve el aislamiento de los datos y el control del acceso a los mismos.

Al encapsular los atributos se logra:

- Reducir el acoplamiento entre clases.
- Proteger la integridad de los datos.
- Mejorar la mantenibilidad del código.
- Favorecer la reutilización de componentes (Bloch, 2018).

Tabla 8: Modificadores de acceso public, private, protected y default.

Modificador	**Clase**	**Subclase (mismo paquete)**	**Subclase (otro paquete)**	**Otra clase**
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

- *public*: el miembro puede ser accedido desde cualquier clase.
- *private*: el miembro es accesible solo dentro de la clase en la que está declarado.
- *protected*: accesible dentro del mismo paquete o en subclases, incluso si están en otro paquete.
- *default*: si no se especifica ningún modificador, el miembro solo es accesible dentro del mismo paquete.

Métodos de Acceso (*Getters y Setters*)

Para aplicar encapsulamiento correctamente, se declaran los atributos como *private* y se proporciona acceso a ellos mediante métodos públicos denominados *getters y setters*.

```
public class Perro {
   // -Atributos privados
   private String Nombre;

   //Metodos get y set
   public String getNombre(){
      return Nombre;
   }
   public void setNombre(String nom){
      this.Nombre = nom;
   }
}
```

Buenas prácticas en el  encapsulamiento

1. Declarar los atributos como *private*: evita accesos indebidos desde otras clases.
2. Usar *getters y setters* con lógica de validación: para verificar que los datos asignados sean válidos.
3. Evitar el uso innecesario de *public*: expone demasiado la estructura interna de las clases.
4. Utilizar final cuando sea apropiado: para atributos que no deberían cambiar una vez asignados.
5. Proteger la mutabilidad de los objetos: en el caso de atributos que son referencias a colecciones u otros objetos complejos, considerar devolver copias defensivas.

2.9 Implementación de clases perro con atributos privados

Caso estudio: Crear un clase perro con sus principales atributos privados.

Ejercicio #3: Clase perro con atributos privados

Abrir el proyecto con el nombre 03_IntroduccionPOO, para la modificación de la clase Perro, con todos los atributos privados y métodos que se presentan en la Figura 14.

Abstracción de la entidad Perro.

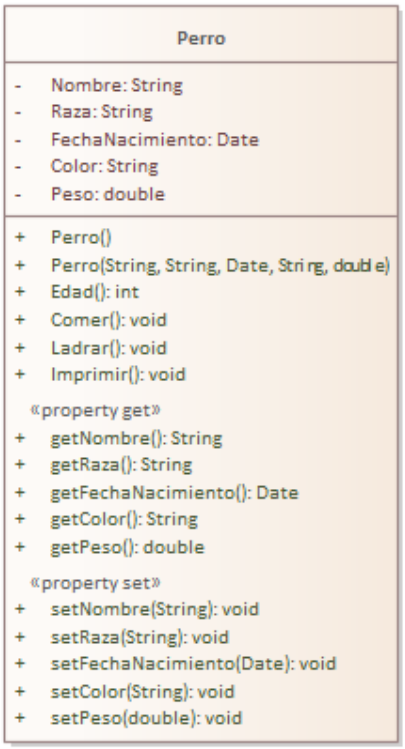


Figura 14: Diagrama de la clase Perro.
Fuente: Elaboración propia.

1. Creación de la clase Perro con atributos privados.

```
package Entidades;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import javax.swing.JOptionPane;
public class Perro
{
   /// -Atributos de la clase privados
```

```
private String Nombre;
private String Raza;
private Date FechaNacimiento;
private String Color;
private double Peso;
public String getNombre()
{
   return Nombre;
}
public void setNombre(String nom)
{
   this.Nombre = nom;
}
public String getRaza()
{
   return Raza;
}
public void setRaza(String raz)
{
   this.Raza = raz;
}
public Date getFechaNacimiento()
{
   return FechaNacimiento;
}
public void setFechaNacimiento(Date fn)
{
   this.FechaNacimiento = fn;
}
public String getColor()
{
   return Color;
}
public void setColor(String col)
{
   this.Color = col;
}
public double getPeso()
{
   return Peso;
}
public void setPeso(double pes)
{
   this.Peso = pes;
}
//Definición de constructores
public Perro(){
```

```
}
 public Perro(String nom, String raz, Date fec, String col,double pes)
{
   this.Nombre = nom;
   this.Raza = raz;
   this.FechaNacimiento = fec;
   this.Color = col;
   this.Peso=pes;
}
//Definición de metodos
public int Edad()
{
   Calendar cal= new GregorianCalendar();
   cal.setTime(FechaNacimiento);
   Calendar hoy = new GregorianCalendar();
   hoy.setTime(new Date());
   int anios = hoy.get(Calendar.YEAR) - cal.get(Calendar.YEAR);
   if(hoy.get(Calendar.MONTH) < cal.get(Calendar.MONTH)||
     (hoy.get(Calendar.MONTH) == cal.get(Calendar.MONTH)&&
            hoy.get(Calendar.MONTH)< cal.get(Calendar.MONTH)))
   {
      anios--;
   }
   return anios;
}
public void Comer()
{
   String men="";
   if(Peso <10)
   {
     men="Necesitar entre 30 y 130 gramos de alimento seco por día";
   }
   if(Peso >= 10 && Peso <20)
   {
     men="Necesitar entre 180 y 320 gramos de alimento seco por día";
   }
   if(Peso >= 20 && Peso <30)
   {
     men="Necesitar entre 320 y 440 gramos de alimento seco por día";
   }
   JOptionPane.showMessageDialog(null, men);

}
public void Ladrar()
{
    JOptionPane.showMessageDialog(null, "Guao..Guao");
}
```

```
  public void Imprimir()
  {

     JOptionPane.showMessageDialog(null,"Datos del Perro:" +
           this.Nombre +"\n"+ "Raza :" + this.Raza +"\n"+
      "Edad :" + Edad() +"\n"+ "Color :" + this.Color);
  }
}
```

2. La implementación del JFrame frmPerro será la misma no cambia la codificación.

2.10 Implementación de clases electrodoméstico con atributos privados

Caso de estudio: Venta de electrodoméstico

Se desea crear una clase electrodoméstico con las siguientes características:

- Sus atributos son privados (Cliente, Nombre, PrecioBase, Color, ConsumoEnergético (letras entre A y D)) y Peso.
- Por defecto, el cliente será "Bruno", el nombre del producto "Nevera", color será Blanco, el consumo energético será D, el precio Base es de $100 y el peso de 5 kg..
- Los colores disponibles son blanco, negro, rojo, azul y gris. No importa si el nombre está en mayúsculas o en minúsculas.
- Los métodos que implementara serán:

  comprobarConsumoEnergetico(): comprueba que la letra es correcta, sino es correcta usara la letra por defecto. Se invocará al crear el objeto y no será visible. La tabla de valores se muestra a continuación.

Tabla 9: Valores de consumo energético.

Letra	Precio
A	$100
B	$80
C	$60
D	$50

comprobarColor(): comprueba que el color es correcto, sino lo es usa el color por defecto. Se invocará al crear el objeto y no será visible.

comprobarPeso(): comprueba que el peso este ingresado correctamente, según su la tabla dependiendo el peso tiene un diferente valor.

Tabla 10: Valores del peso del electrodoméstico.

Peso	Precio
Entre 0 y 19 kg	$10
Entre 20 y 49 kg	$50
Entre 50 y 79 kg	$80
Mayor 80 kg	$50

precioFinal(): calculara el precio a pagar según su precio base más el consumo energético y su peso.

Ejercicio #4: Simulación de venta de electrodoméstico

Crear un proyecto con el nombre 04_Electrodomestico, crear dos Packages Entidades y Formularios. En Package Entidades crear una clase Electrodoméstico con todos los atributos y métodos del enunciado del problema. En el Package Formularios crear un formularios frmElectrodomestico.

1. Al realizar la abstracción del problema se obtiene la clase Electrodoméstico con los atributos privados, constructores y los métodos como se muestra en la Figura 15.

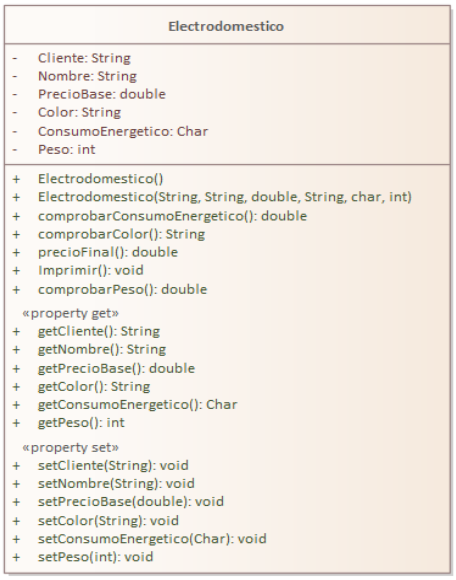


Figura 15: Diagrama de la clase Electrodoméstico en UML
Fuente: Elaboración propia.

2. Implementación de la clase Electrodoméstico

```
package Entidades;

import javax.swing.JOptionPane;

public class Electrodomestico {

        private String Cliente;
        private String Nombre;
        private double PrecioBase;
        private String Color;
        private char ConsumoEnergetico;
        private int Peso;
        public Electrodomestico()
    {
        this.Cliente = "Desconocido";
        this.Nombre = "Desconocido";
        this.Color = comprobarColor();
        ConsumoEnergetico='F';
        PrecioBase =100;
        Peso= 5;
        }
    public Electrodomestico(String cli, String nom, double pb, String col, char
                ce, int peso)
```

```
{
    this.Cliente =cli;
    this.Nombre=nom;
    this.PrecioBase = pb;
    this.Color = col;
    this.ConsumoEnergetico = ce;
    this.Peso = peso;
    }
    public String getCliente()
{
        return Cliente;
    }

    public void setCliente(String newVal)
{
        Cliente = newVal;
    }

    public String getNombre()
{
        return Nombre;
    }
    public void setNombre(String newVal)
{
        Nombre = newVal;
    }

    public double getPrecioBase()
{
        return PrecioBase;
    }

    public void setPrecioBase(double newVal)
{
        PrecioBase = newVal;
    }

    public String getColor()
{
        return Color;
    }

    public void setColor(String newVal)
{
```

```
            Color = newVal;
    }

    public char getConsumoEnergetico()
{
            return ConsumoEnergetico;
    }

    public void setConsumoEnergetico(char newVal)
{
            ConsumoEnergetico = newVal;
    }

    public int getPeso()
{
            return Peso;
    }
    public void setPeso(int newVal)
{
            Peso = newVal;
    }

    public double comprobarConsumoEnergetico()
{

    double val=0;
    if(ConsumoEnergetico=='A')
    {
      val=100;
    }
     if(ConsumoEnergetico=='B')
    {
      val=80;
    }
     if(ConsumoEnergetico=='C')
    {
      val=60;
    }
    if(ConsumoEnergetico=='D')
    {
      val=50;
    }
     if(ConsumoEnergetico=='E')
    {
```

```
        val=30;
      }
       if(ConsumoEnergetico=='F')
      {
        val=10;
      }
      return val;
      }



 public String comprobarColor()
 {

      String col ="Blanco";
      if(Color.equals("Negro"))
      {
        col="Negro";
      }
       if(Color.equals("Rojo"))
      {
        col="Rojo";
      }
        if(Color.equals("Azul"))
      {
        col="Azul";
      }
      if(Color.equals("Gris"))
      {
        col="Gris";
      }
      return col;
      }
      public double precioFinal()
{
              return PrecioBase +
              comprobarConsumoEnergetico()+comprobarPeso();
      }

      public void Imprimir()
{
      JOptionPane.showMessageDialog(null, "Cliente :" + this.Cliente +
              "\n"+"Producto :" + this.Nombre + "\n"+
```

```
        "Precio Cobrar :" + precioFinal());
        }
        public double comprobarPeso()
    {

        double val=0;
        if(Peso >= 0 && Peso <=19)
        {
           val = 10;
        }
        if(Peso >= 20 && Peso <=49)
        {
           val = 50;
        }
        if(Peso >= 50 && Peso <=79)
        {
           val = 80;
        }
          if(Peso >= 80 )
        {
           val = 100;
        }
        return val;
        }

}
```

3. Diseño del formulario para la clase Electrodoméstico

Figura 16: Diseño del JFrame frmElectrodomestico.
Fuente: Elaboración propia.

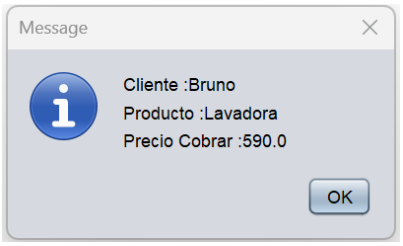
4. Implementación del frmElectrodomestico

```
package Formularios;

import Entidades.Electrodomestico;
import javax.swing.JOptionPane;

public class frmElectrodomestico extends javax.swing.JFrame
{

 public frmElectrodomestico()
 {
     initComponents();
     setSize(500,500);
     setLocationRelativeTo(this);
 }
 public void LimpiarBotones()
 {
      jTextField1.setText("");
      jTextField2.setText("");
      jTextField3.setText("");
      jTextField4.setText("");
      jComboBox1.setSelectedIndex(0);
      jComboBox2.setSelectedIndex(0);
 }
 private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
 {

    if(!jTextField1.getText().equals(" ")&&  !jTextField2.getText().equals("
         ")&& jComboBox1.getSelectedIndex()>0
         &jComboBox2.getSelectedIndex()>0)
    {
     try{
        String cli= jTextField1.getText();
        String pro = jTextField2.getText();
        double pb = Double.parseDouble(jTextField3.getText());
        String col = jComboBox1.getSelectedItem().toString();
        char ce= jComboBox2.getSelectedItem().toString().charAt(0);
        int pes = Integer.parseInt(jTextField4.getText());
        Electrodomestico oe= new  Electrodomestico(cli,pro,pb,col,ce,pes);
         oe.Imprimir();

         }catch(Exception ex)
         {
```

```
            JOptionPane.showMessageDialog(null,ex.getMessage());
        }
    }
    else
        JOptionPane.showMessageDialog(null,"ingrese correctamente los
                datos");
  }

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)
{
    LimpiarBotones();
  }
```

5. Salida del sistema.

Tabla 11: Presentación de resultados de la clase Electrodoméstico.

Interfaz de entrada	Interfaz de salida
	

2.11 Implementación de riesgo cardiovascular con atributos privados

Caso estudio: Determinación del índice de riesgo cardiovascular

Una fórmula permite calcular un índice de riesgo cardiovascular (IRC) basado en la edad, el nivel de colesterol total y la presión arterial sistólica de una persona. La fórmula es la siguiente:

$$ICR = (ColesterolxPresionArterial) \div (Edad * 100)$$

Dependiendo del valor del IRC y de la edad de la persona, se clasifica el riesgo cardiovascular de la siguiente manera:

Edad	**Bajo riesgo**	**Riesgo moderado**	**Alto riesgo**
Hasta 30 años	IRC ≤ 3.0	3.1 ≤ IRC ≤ 6.0	IRC > 6.0
Entre 31 y 50 años	IRC ≤ 4.0	4.1 ≤ IRC ≤ 7.0	IRC > 7.0
Más de 50 años	IRC ≤ 5.0	5.1 ≤ IRC ≤ 8.0	IRC > 8.0

Confeccione un programa en Java que solicite la edad, el nivel de colesterol total (en mg/dL) y la presión arterial sistólica (en mm Hg) de una persona, y determine si tiene bajo riesgo, riesgo moderado o alto riesgo cardiovascular.

Por ejemplo, si se ingresa Colesterol: 200 mg/dL, Presión Arterial: 120 mm Hg, Edad: 45, el programa calculará:

$$ICR = (200 * 120) \div (45 * 100) = 5.33$$

Dado que la edad está entre 31 y 50 años y el IRC es 5.33, se determina que la persona tiene **Riesgo moderado**.

Ejercicio #5: Simulación del índice de riesgo cardiovascular

Crear un proyecto con el nombre 05_IndiceRiesgo, crear dos Packages Entidades y Formularios. En Package Entidades crear una clase Cardiovascular con todos los atributos y métodos del enunciado del problema. En el Package Formularios crear un formularios frmElectrodomestico.

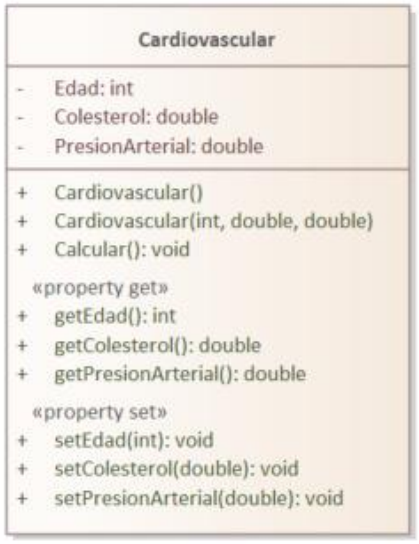


Figura 17: Diagrama de la clase Cardiovascular en UML
Fuente: Elaboración propia.

1. Implementación de la clase Cardiovascular

```
package Entidades;

import javax.swing.JOptionPane;

public class Cardiovascular
{

   private int Edad;
   private double Colesterol;
   private double PresionArterial;

   public Cardiovascular ()
   {

   }

   public Cardiovascular (int Edad, double Colesterol, double
            PresionArterial)
   {
      this.Edad = Edad;
      this.Colesterol = Colesterol;
      this.PresionArterial = PresionArterial;
   }

   public int getEdad()
```

```
{
   return Edad;
}



public void setEdad(int newVal)
{
   Edad = newVal;
}

public double getColesterol()
{
   return Colesterol;
}

public void setColesterol(double newVal)
{
   Colesterol = newVal;
}

public double getPresionArterial()
{
   return PresionArterial;
}

public void setPresionArterial(double newVal)
{
   PresionArterial = newVal;
}

public void Calcular(int eda, double cole, double pe)
{
double IRC;

if (eda<=30)
{
   IRC = ((cole*pe)/eda)/100;

   if (IRC<=3.0)
   {
      JOptionPane.showMessageDialog(null, "IRC: " + IRC + "\n"+ "Bajo
            riesgo");
   }
   else if (IRC>=3.1 && IRC<=6.0)
   {
```

```
      JOptionPane.showMessageDialog(null, "IRC: " + IRC + "\n"+ "Riesgo
            Moderado");
   }
            else {
      JOptionPane.showMessageDialog(null, "IRC: " + IRC + "\n"+ "Alto
            riesgo");
   }
}
if (eda>=31 && eda<=50)
{
   IRC = ((cole*pe)/eda)/100;
   if (IRC<=4.0)
   {
      JOptionPane.showMessageDialog(null,"IRC: " + IRC + "\n"+  "Bajo
            riesgo");
   }

        else if (IRC>=4.1 && IRC<=7.0)
        {
      JOptionPane.showMessageDialog(null, "IRC: " + IRC + "\n"+ "Riesgo
            Moderado");
   }
   Else
   {
      JOptionPane.showMessageDialog(null, "IRC: " + IRC + "\n"+ "Alto
            riesgo");
   }
}
if (eda>50)
{
  IRC = ((cole*pe)/eda)/100;
   if (IRC<=5.0)
   {
      JOptionPane.showMessageDialog(null,"IRC: " + IRC + "\n"+  "Bajo
            riesgo");
   }
   else if (IRC>=5.1 && IRC<=8.0)
   {
      JOptionPane.showMessageDialog(null,"IRC: " + IRC + "\n"+  "Riesgo
            Moderado");
   }
   Else
   {
      JOptionPane.showMessageDialog(null,"IRC: " + IRC + "\n"+  "Alto
            riesgo");
   }
}
```

```
 }

}
```

2. Implementación del frmCardiovascular

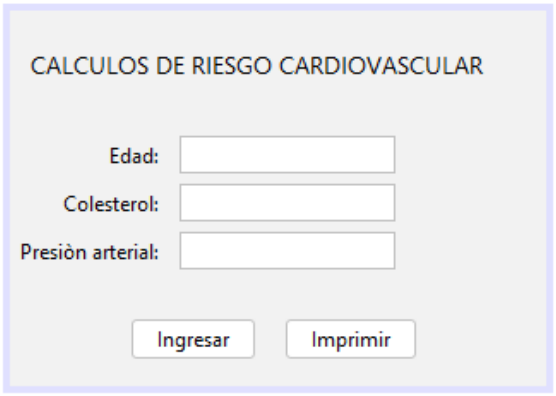


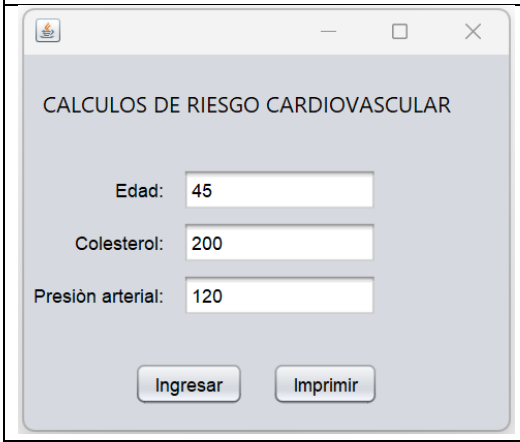
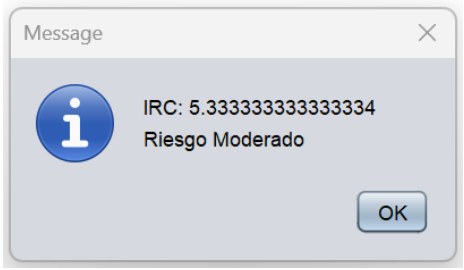
Figura 18: Diseño del JFrame frmCardiovascular.
Fuente: Elaboración propia.

```
package Formularios;
import Entidades.Cardiovascular;

public class frm_Cardiovascular  extends javax.swing.JFrame
{

   public frm_Cardiovascular ()
   {
      initComponents();
   }
         Cardiovascular  ob;
         int edad;
         double col, pa;

           private void jButton1ActionPerformed(java.awt.event.ActionEvent
           evt)
         {
      // TODO add your handling code here:
      edad = Integer.parseInt(jTextField1.getText());
      col = Integer.parseInt(jTextField2.getText());
      pa = Integer.parseInt(jTextField3.getText());
      ob= new Cardiovascular(edad,col,pa);
   }

   private void jButton3ActionPerformed(java.awt.event.ActionEvent evt)
```

```
        {
    ob.Calcular(edad, col, pa);

   }

}
```

3. Salida del sistema.

Tabla 12: Presentación de resultados de la clase Cardiovascular.

Interfaz de entrada	Interfaz de salida
	

RESUMEN DEL CAPITULO

En este capítulo, se desarrollaron habilidades esenciales para modelar clases y objetos utilizando diagramas UML, enfocándose en la correcta identificación de atributos, métodos y relaciones entre clases. Se aprendió a diferenciar tipos de relaciones como asociación, agregación, composición y herencia, y se mostró cómo trasladar estos modelos a código Java. La práctica constante y el uso de herramientas visuales fortalecen la capacidad de diseñar software robusto, mantenible y orientado a objetos.

AUTOEVALUACIÓN

En todas las preguntas de selecciona múltiple un literal es la respuesta correcta.

1. ¿Qué es una clase en el contexto de la programación orientada a objetos?

   a) Un algoritmo específico
   b) Una variable con múltiples valores
   c) Un modelo o plantilla para crear objetos
   d) Un tipo de dato primitivo

2. ¿Qué representa un objeto en programación orientada a objetos?
   a) Una línea de código
   a) Un valor lógico
   b) Una instancia de una clase
   c) Un tipo de error

3. ¿Cuáles de los siguientes enunciados con respecto a la POO, es el correcto?
   a) Los objetos físicos son entidades visibles como por ejemplo componentes de una interfaz gráfica
   b) Los objetos lógicos son entidades no visibles como por ejemplo la abstracción de un trabajador en una empresa
   c) El proceso de instanciación consiste en identificar o se seleccionar entidades u objetos reales con similares características
   d) El proceso de instanciación consiste en crear objetos en aplicaciones informáticas

4. ¿Qué es una clase?
   a) Un tipo de dato definido por el usuario que tiene a estructura de datos y las operaciones asociadas con esos datos
   b) Es un modelo que describe uno o más objetos del mundo real con similares atributos y comportamiento
   c) Presenta una interfaz (métodos públicos) para poder interactuar con el exterior
   d) Es una entidad que contiene unos atributos o características particulares (datos) y un comportamiento

5. ¿Cuál es el propósito principal del encapsulamiento?
   a) Hacer que el código sea más corto

b) Aumentar la complejidad del programa
c) Ocultar detalles de implementación y proteger datos
d) Duplicar clases

6. ¿Qué permite la abstracción en la programación orientada a objetos?
a) Heredar comportamientos sin restricciones
b) Definir comportamientos sin necesidad de implementar los detalles
c) Crear múltiples objetos con el mismo nombre
d) Ejecutar código sin compilar

7. ¿Qué característica no corresponde a la programación orientada a objetos?
a) Encapsulamiento
b) Herencia
c) Iteración
d) Polimorfismo

8. ¿Seleccione la respuesta correcta con respecto a los constructores?
a) Los constructores son métodos abstractos que llevan el mismo nombre de la clase
b) Tienen ningún valor de retorno void y no se heredan.
c) Se invocan automáticamente al momento de crear el objeto
d) Su función es asignar memoria a la clase e inicializar sus datos

9. ¿Qué es UML?
a) Lenguaje de modelado de datos que se usa para la codificación de los programas a desarrollarse
b) Es un conjunto de herramientas, que permite modelar (analizar y diseñar) sistemas orientados a objetos, no es un lenguaje de programación ni un método de desarrollo
c) Lenguaje modelamiento unificado que permite a los desarrolladores a codificar sistemas en una forma más eficiente
d) Herramienta de software asistida por computadora

10. ¿Cuáles de los siguientes enunciados con respecto al encapsulamiento, es el correcto?
a) Los métodos set son procedimientos que devuelven un valor de un atributo encapsulado
b) Los métodos set son funciones que no retornan ningún un valor de un atributo encapsulado
c) Los métodos get son funciones que devuelven un valor de un atributo encapsulado

d) Los métodos get son procedimientos que asignan un valor a un atributo encapsulado

EJERCICIOS PROPUESTOS

Realizar los siguientes problemas utilizando la programación orientada a objetos con atributos privados y formularios.

Ejercicio 1: Una empresa calcula sus sueldos multiplicando al valor de la hora y la cantidad de horas que trabajó cada empleado. Además, si el empleado trabajó más de 100 horas lo premian con $100.- y si trabajó más de 200 horas el premio es de $250.-.

La empresa le solicitó a Ud., futuro programador, un programa para poder ingresar por teclado el valor de la hora y la cantidad de horas trabajadas por un empleado y luego determinar e informar por pantalla el sueldo que corresponda abonarle.

Ejercicio 2: Cálculo del salario basado en horas trabajadas

Una empresa paga a sus empleados de la siguiente forma:

- Las primeras 40 horas de trabajo se pagan a la tarifa normal.
- Las siguientes 10 horas (de 41 a 50) se pagan a una tarifa del 50% más.
- Las horas adicionales (más de 50 horas) se pagan al doble de la tarifa normal.

Escribe un programa que calcule el salario semanal de un empleado, dado el número de horas trabajadas y la tarifa por hora.

Ejercicio 3: Cálculo del precio de boletos de avión según la temporada

Una aerolínea ofrece precios diferentes para los boletos según la temporada:

- Temporada baja: $300 por boleto.
- Temporada media: $400 por boleto.
- Temporada alta: $600 por boleto.

Además, si el pasajero es mayor de 65 años, recibe un 10% de descuento sobre el precio. Crea un programa que, dada la temporada y la edad del pasajero, calcule el precio final del boleto

Ejercicio 4: Sistema de gestión de pedidos con descuentos

Una tienda ofrece descuentos en función del total de la compra:

- Si el total es mayor o igual a $1000, se aplica un descuento del 10%.
- Si está entre $500 y $999, se aplica un descuento del 5%.
- Si es menor de $500, no se aplica descuento.

Además, si el cliente es miembro del club de fidelidad, se le aplica un 5% adicional de descuento. Escribe un programa que calcule el total a pagar teniendo en cuenta ambos descuentos.

Ejercicio 5: Venta de boletos de tren con distintas tarifas según el destino y tipo de vagón

Una empresa de venta de boletos de tren tiene distintas tarifas según el destino, el tipo de vagón elegido por el pasajero (turista, preferente o VIP) y la cantidad de boletos comprados. La siguiente tabla indica los precios a pagar por cada tipo de vagón en destino:

Destino	Vagón Turista	Vagón Preferente	Vagón VIP
Ciudad A	$50	$70	$90
Ciudad B	$40	$60	$80
Ciudad C	$30	$50	$70

Además:

- Si el pasajero elige un vagón **preferente**, se cobra un 10% adicional sobre el valor del vagón turista.
- Si el pasajero elige un vagón **VIP**, se cobra un 20% adicional sobre el valor del vagón turista.
- Si el pasajero compra **4 o más boletos**, se aplica un descuento del **10%** sobre el total.

El administrador de la empresa necesita un programa que permita ingresar los siguientes datos:

- Número del destino (1: Ciudad A, 2: Ciudad B, 3: Ciudad C).
- Tipo de vagón (1: Turista, 2: Preferente, 3: VIP).
- Cantidad de boletos solicitados.

El programa deberá calcular el importe total a pagar por la compra, aplicando el recargo correspondiente según el vagón elegido y el descuento si aplica.

REFERENCIA BIBLIOGRÁFICA

Alan, A., & Haley Wixom, B. (2020). Systems Analysis and Design: An Object-Oriented Approach with UML (6.ª ed.). Wiley.

Bloch, J. (2018). Java efectivo mejores prácticas (3.ª ed.). Addison-Wesley Professional.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). The Unified Modeling Language User Guide (2.ª ed.).

Deitel, P., & Deitel, H. (2022). Java como programar (11.ª ed.). Person.

Larman, C. (2020). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall.

Mazón Olivo, B. (2015). Fundamentos de Programación Orientada a Objetos en java (1.ª ed.). Ediciones Utmach.

Moreno Pérez, J. C. (2015). Certificado de profesionalidad, programación orientada a objetos. Rama.

Sierra, K., & Bates, B. (2022). OCA/OCP Java SE 8 Programmer Certification (2.ª ed).

CAPÍTULO III: HERENCIA Y POLIMORFISMO

Objetivos de la unidad

- Identificar qué es la herencia y el polimorfismo en la programación orientada a objetos mediante mecanismos clave para la reutilización y extensión de código para el desarrollo de aplicaciones modulares, escalables y mantenibles.
- Aplicar el *UML* para representar mediante diagramas la herencia de clases, sus atributos, métodos y relaciones.
- Desarrollar programas orientados a objetos, eficientes y escalables, mediante el uso adecuado de la herencia, con el propósito de fomentar la reutilización de código y la optimización del diseño de software.

En este tercer capítulo se centra en dos de los pilares más importantes de la *POO*: herencia y polimorfismo. Aprenderás a extender clases, reutilizar atributos y métodos, y a diseñar jerarquías lógicas. Estudiaremos herencia simple y compuesta, analizaremos el uso del operador super y exploraremos ejemplos de sobrecarga y sobrescritura de métodos. Además, entenderás cómo el polimorfismo permite que un mismo método se comporte de manera distinta según el contexto, brindando flexibilidad y escalabilidad a los programas.

Preguntas de Enfoque

- ¿Por qué es importante la herencia en la reutilización del código?
- ¿Qué beneficios ofrece el polimorfismo en el desarrollo de aplicaciones escalables?
- ¿Cuáles son las diferencias entre sobrecarga y sobrescritura de métodos?
- ¿Cómo se representan las relaciones de herencia en *UML*?
- ¿Qué errores comunes se deben evitar al implementar herencia en Java?

"Heredar no es copiar, es construir sobre lo que ya funciona para crear soluciones más limpias, poderosas y sostenibles."
Robert C. Martin

3. Herencia y polimorfismo

La combinación de la herencia y el polimorfismo en la programación orientada a objetos permite diseñar sistemas más flexibles y adaptables. La herencia establece una estructura jerárquica que organiza las clases de manera lógica, facilitando la reutilización y extensión del código. Por su parte, el polimorfismo proporciona la capacidad de interactuar con objetos de diferentes clases de manera uniforme, permitiendo que una misma interfaz se comporte de manera específica según el tipo de objeto que la implemente.

3.1 Introducción a la herencia

Definiciones y propósitos en la POO

Según(Deitel & Deitel, 2022), la herencia permite construir jerarquías lógicas de clases, facilitando el diseño modular y el mantenimiento del software. Para (Horstmann, 2022), La herencia es un mecanismo que permite que una clase hija herede atributos y métodos de una clase padre. Esto promueve la reutilización del código y la expansión de funcionalidades, simplificando el mantenimiento.

La herencia es el segundo pilar de la programación orientada a objetos, es un mecanismo que permite crear nuevas clases a partir de otras ya existentes. Esto favorece la reutilización del código, ya que evita tener que reescribir atributos y métodos comunes.

Ventajas de la Herencia

- ✓ Reutilización de código**:** Permite usar atributos y métodos ya definidos en una clase base sin volver a escribirlos.
- ✓ Facilita la extensión del software: Se pueden agregar nuevas funcionalidades creando subclases sin modificar el código original.
- ✓ Organización jerárquica: Ayuda a estructurar el código en niveles de generalidad, como una jerarquía lógica de clases.
- ✓ Mejora la mantenibilidad: Al centralizar comportamientos comunes, los cambios se hacen en un solo lugar y se reflejan en todas las clases hijas.

Desventajas de la Herencia

- ✗ Dependencia fuerte entre clases: Cambios en la clase padre pueden afectar el comportamiento de las subclases, incluso sin que lo notes de inmediato.

- ✗ Rígida estructura jerárquica: Un diseño mal planificado puede generar jerarquías innecesarias o difíciles de mantener.
- ✗ Dificultad para entender jerarquías complejas: Cuando hay muchas clases relacionadas, el código puede volverse confuso para otros programadores (o para ti en el futuro).
- ✗ Puede limitar la flexibilidad: A veces, usar composición (usar objetos dentro de objetos) es más flexible que heredar.

3.2 Tipos de herencia

Herencia Simple

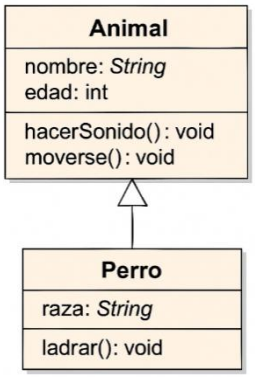


Figura 19: Diagrama UML de Herencia simple.
Fuente: Elaboración propia.

Cuando una clase hija hereda directamente de una clase padre, java implementa un modelo de herencia simple, lo que significa que una clase puede heredar directamente de una única superclase utilizando la palabra clave *extends*. Este diseño evita la complejidad y ambigüedad asociadas con la herencia múltiple.

Herencia Compuesta

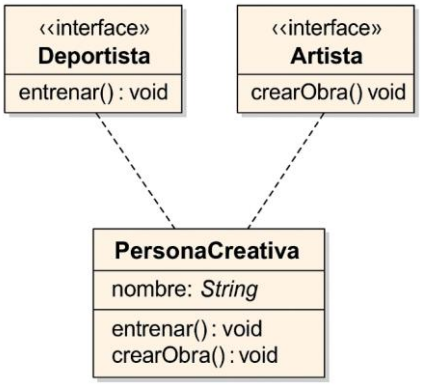


Figura 19: Diagrama UML de herencia múltiple.

Fuente: Elaboración propia.

Cuando una clase hija hereda directamente de dos o más clase padre, java no permite la herencia múltiple de clases. para evitar ambigüedades. Sin embargo, Java ofrece una solución elegante a través del uso de interfaces. Una clase puede implementar múltiples interfaces, lo que permite simular la herencia múltiple de comportamientos.

3.3 Implementación de herencia en java

En Java, se utiliza la palabra clave extends para establecer esta relación.

```
class Estudiante extends  Persona {...}
```

Constructores en la  **herencia**

En Java, los constructores no se heredan de la clase padre. Por esta razón, cuando una clase hija extiende a una clase padre, es su responsabilidad asegurarse de que los atributos heredados sean correctamente inicializados. Esto se logra mediante el uso de la palabra clave super, que permite llamar explícitamente a un constructor de la superclase desde la subclase.

Super permite acceder a constructores y métodos de la clase padre, lo cual es fundamental cuando se redefine comportamiento en la subclase(Sommerville, 2021). Además, dentro de una misma clase, se puede usar *this* para invocar otro constructor de la misma clase, facilitando la sobrecarga de constructores.

3.4 Diagrama de herencia en UML

En UML, la herencia se muestra con una flecha blanca desde la clase hija hacia la clase padre. Este modelo favorece el análisis estructurado de jerarquías de clases (Pérez & Ramírez, 2024).

Figura 21: Diagrama UML de la Herencia.
Fuente: Elaboración propia.

Las nuevas clases (Fijo, Consultor) de la Figura 19. Son denominadas derivadas, hijas o subclases, a partir de la clase(Persona) ya existente, llamadas base, padre o superclase".

De esta manera los objetos pueden construirse en base a otros objetos ya creados.

Propagación de Atributos y Métodos

Cuando una nueva clase deriva de una clase padre, hereda automáticamente todos los atributos y métodos públicos y protegidos de la clase padre. Esto significa que puede utilizar directamente estos elementos sin necesidad de redefinirlos. Sin embargo, los atributos privados de la clase padre no son accesibles directamente desde la clase hija.

Para interactuar con estos atributos, se utilizan métodos públicos conocidos como *getters* y *setters*, que permiten leer y modificar los valores de manera controlada(Gamma & Helm, 2021). Este enfoque sigue el principio de encapsulamiento, que busca proteger los datos internos de una clase y exponer solo lo necesario. En la Figura 21 la clase derivada Fijo hereda todos los atributos y métodos de la clase base Persona.

Tabla 13: Atributos y métodos de la clase Fijo.

Clase Fijo	Total atributos y métodos
Atributos	Cedula, Nombres, Apellidos, Sexo, FechaNacimiento, Estado, Sueldo, Comisiones.
Métodos	Persona(), Persona(….), Fijo(), Fijo(…), ImprimirPersona(), CalcularSueldo(), Imprimir().

La nueva clase Fijo tiene en total ocho atributos y siete métodos.

3.5 Implementación de herencia en java

Ejercicio 6: Creación de la herencia

Crear un proyecto con el nombre 06_HerenciaSimple, crear dos Packages Entidades y formularios. En Package Entidades crear una clase Persona con todos los atributos y métodos. Luego crear otra clase Estudiante heredando de la clase persona. Como se muestra en la Figura 22.

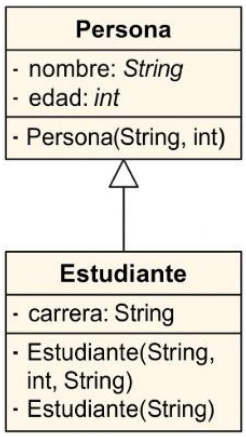


Figura 20: Diagrama UML de herencia.

1. Implementación de la clase Persona

```
// Clase base
class Persona {
   String nombre;
   int edad;

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

2. Implementación de la clase Estudiante

```
// Clase derivada
class Estudiante extends Persona {
   String carrera;
   public Estudiante(String nombre, int edad, String carrera) {
      super(nombre, edad); // Llama al constructor de la clase padre
      this.carrera = carrera;
   }

   // Otro constructor usando this
    public Estudiante(String nombre){
    this(nombre, 18, "Software"); // Llama al otro constructor de la   misma
                                                          clase
   }
}
```

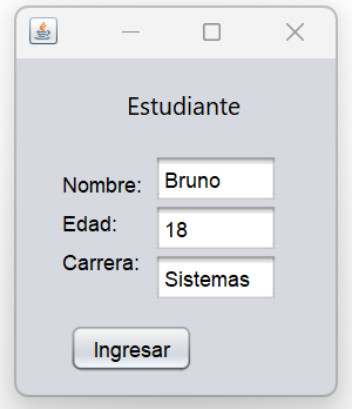
- En la clase derivada Estudiante en el constructor super (nombre, edad) se utiliza para inicializar los atributos heredados de la clase Persona.
- En el segundo constructor se utiliza this(nombre, 18, "Software") para llamar a otro constructor de la misma clase, facilitando la sobrecarga de constructores con valores por defecto.

3. Implementación del frmEstudiante.

```
public class frmEstudiante extends javax.swing.JFrame {

   public frmEstudiante() {
      initComponents();
   }
   private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
      // TODO add your handling code here:
      String nom=jTextField1.getText();
      int eda=Integer.parseInt(jTextField2.getText());
      String car= jTextField3.getText();
      Estudiante oe= new Estudiante(nom, eda, car);
      oe.ImprimirEstudiante();
   }
}
```

4. Salida del sistema.

Tabla 14: Presentación de resultados de la clase Cardiovascular.

Interfaz de entrada	Interfaz de salida
	

3.6 Buenas prácticas en herencia

Principio de sustitución de Liskov.

El Principio de Sustitución de Liskov **(**LSP**,** Liskov Substitution Principle**)** es uno de los cinco principios SOLID de la programación orientada a objetos, formulado por Barbara Liskov en 1987.

Definición formal

Si **S** es un subtipo de **T**, entonces los objetos de tipo **T** en un programa pueden ser reemplazados por objetos de tipo **S** sin **alterar** el correcto funcionamiento del programa.

En otras palabras, una clase hija debe poder **sustituir** a su clase padre sin romper la lógica del sistema.

Implicaciones del LSP

- Herencia coherente: Las subclases deben respetar el comportamiento esperado de la clase base.
- Contrato de métodos: Una subclase no debe violar precondiciones ni relajar excesivamente las postcondiciones de los métodos heredados.
- Consistencia de invariantes: Las reglas que se cumplen en la clase padre también deben cumplirse en la clase hija.
- Polimorfismo seguro: El polimorfismo funciona correctamente solo si las subclases cumplen con LSP.

Ejemplo: clásico de violación

Supongamos que modelamos un Rectángulo y un Cuadrado para calcular el área:

Ejercicio 7: Calcular el área de un rectángulo y un cuadro.

Crear un proyecto con el nombre 07_LspFiguras, crear dos Packages Entidades y Formularios. En el Package Entidades crear las clase Rectángulo y Cuadrado con los siguientes atributos y métodos.

1. Implementación de la clase Rectángulo

```
package Entidades;

public class Rectangulo {
   protected int ancho;
   protected int alto;
   public void setAncho(int ancho) { this.ancho = ancho; }
   public void setAlto(int alto) { this.alto = alto; }
   public int getArea() { return ancho * alto; }
}
```

2. Implementación de la clase Cuadrado

```
package Entidades;

public class Cuadrado extends Rectangulo
{
   @Override
   public void setAncho(int ancho) {
      this.ancho = this.alto = ancho;
   }

   @Override
   public void setAlto(int alto) {
      this.ancho = this.alto = alto;
   }
   @Override
   public int getArea() { return ancho * alto; }

}
```

Si un método espera un Rectángulo, pero recibe un Cuadrado, podría romper la lógica porque los setters del cuadrado no respetan la independencia entre ancho y alto. Esto **viola LSP**.

El Principio de Sustitución de Liskov garantiza que:

- La herencia sea usada de forma coherente.
- Las subclases mantengan el comportamiento de la clase base.
- Se respete el polimorfismo sin generar efectos inesperados.

3.7 Introducción al polimorfismo

Definición y propósito

Segun(Li & Wang, 2023), el polimorfismo permite que objetos de diferentes clases respondan de forma distinta al mismo método. Esto mejora la flexibilidad del sistema y reduce acoplamientos innecesarios.

Para (Bailón & Baltazar, 2021), el polimorfismo se refiere a la capacidad de una variable de referencia para cambiar su comportamiento según la instancia del objeto que contiene. Esto permite que múltiples objetos de diferentes subclases sean tratados como objetos de una superclase única, seleccionando automáticamente los métodos apropiados basados en la subclase a la que pertenece.

El polimorfismo es el tercer pilar de la programación orientada a objetos, permite crear varios métodos con el mismo nombre tanto en la clase base como en la clase derivada, pero se comportan de manera diferente.

Ventajas del polimorfismo

- ✓ Reutilización de código: Permite utilizar métodos de la superclase en las subclases, reduciendo la redundancia.
- ✓ Flexibilidad: Facilita la extensión y modificación del código sin afectar otras partes del sistema.
- ✓ Mantenibilidad: Simplifica el mantenimiento del software al permitir cambios en la superclase que se reflejan en las subclases.
- ✓ Abstracción: Permite trabajar con objetos de diferentes clases de manera uniforme.

Desventaja del polimorfismo

- ✓ Complejidad: Puede aumentar la complejidad del sistema al tener múltiples implementaciones de un mismo método.
- ✓ Rendimiento: La resolución dinámica de métodos puede afectar el rendimiento en sistemas con recursos limitados.
- ✓ Depuración: Puede dificultar la depuración y el seguimiento del flujo del programa debido a la dinámica en tiempo de ejecución.

3.8 Diagrama de polimorfismo en UML

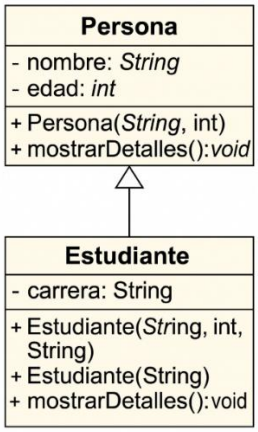


Figura 22: Diagrama UML de polimorfismo.
Fuente: Elaboración propia.

Observa que el método mostrarDetalles(): void aparece en ambas clases, lo que sugiere que estamos aplicando polimorfismo por sobrescritura (override) en programación orientada a objetos.

3.9 Implementación de polimorfismo

Método en la clase Persona

```
public void mostrarDetalles()
{
   JOptionPane.showMessageDialog(null,"Nombre: " + nombre);
   JOptionPane.showMessageDialog("Edad: " + edad);
}
```

Método en la clase Estudiante

```
@Override
public void mostrarDetalles() {
   super.mostrarDetalles(); // Llama al método de la clase padre
   JOptionPane.showMessageDialog(null,"Carrera: " + carrera);
}
```

Sobrecarga de métodos

La sobrecarga de métodos permite definir varios métodos con el mismo nombre, pero diferentes parámetros(Arora & Suri, 2023).

```
void mostrarDetalles() {
   // Método sin parámetros
}

void mostrarDetalles(String mensaje) {
   // Método con un parámetro String
}
```

En Java, la sobrecarga de métodos (*method overloading*) es un mecanismo que permite definir varios métodos con el mismo nombre dentro de una misma clase**,** pero con diferentes listas de parámetros (ya sea en cantidad, tipo o el orden de los mismos).

¿Cómo funciona?

Java selecciona cuál método ejecutar en tiempo de compilación, según los argumentos proporcionados en la llamada del método.

```
mostrarDetalles();          // Llama al primer método
mostrarDetalles("Hola Java");  // Llama al segundo método
```

Aquí tenemos dos métodos llamados mostrarDetalles, pero uno no recibe parámetros y el otro sí. El compilador sabrá cuál usar según cómo lo invoques:

¿Por qué es útil la sobrecarga?

- Permite mejorar la legibilidad del código.
- Evita nombres duplicados para métodos que hacen tareas similares con diferentes datos.
- Se utiliza frecuentemente en constructores sobrecargados **y** APIs de utilidad.

3.10 Implementación de herencia

Caso de estudio: Sistema de nómina utilizando herencia simple

Una compañía paga semanalmente a sus Trabajadores, quienes se dividen en cuatro tipos:

- **Fijos** que reciben un salario mensual, descantando 9.32 % del seguro social y anticipos realizados considerando que el anticipo no puede ser mayor al 50 % del salario mensual.
- **Contratados**, que perciben un sueldo por hora y pago por las horas trabajadas, si las horas trabajadas excedan a 160 horas, se le realiza el descuento seguro social.
- **Comisión**, que perciben una tarifa de comisión del porcentaje de sus ventas brutas.
- **Comisionados**, que obtienen un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base.

La gerencia ha decidido compensar a sus empleados que estén de cumpleaños en el mes de pago con una bonificación de $100 libres sin descuento. Si la edad del trabajador es de 60 años en adelante tiene otra bonificación de $50 por antigüedad.
Los atributos de la clase Persona son (Cedula, Nombres, Apellidos, Sexo, Fecha Nacimiento). Si considera más atributos a la clase puede agregar.
Se pide realizar lo siguiente:

- Diseñar el modelado de datos UML en Enterprise Architecter luego generar el código a java.
- Crear una función que permita calcular la edad a partir de la fecha de nacimiento.
- Crear una función que permita calcular la antigüedad a partir de la fecha de ingreso del trabajador.
- Crear una función para calcular los bonos solicitados por cumpleaños y antigüedad.
- Implementar el sistema con los requerimientos en java.

Ejercicio 8: Sistema de nómina aplicando herencia simple

Crear un proyecto con el nombre 08_SistemaNominaHerencia, crear dos Packages Entidades y Formularios. En el Package Entidades crear las clase Trabajador, Fijo, Contratado, Comisión y Comisionado con todos los atributos y métodos del enunciado del problema. En el Package Formularios un JFrame frmSistema.

Al analizar los requerimientos solicitados, llegamos a la abstracción del problema donde identificamos las principales entidades con sus atributos, métodos y relaciones como se muestra en la Figura 23.

Figura 21: Diagrama UML de caso de estudio.
Fuente: Elaboración propia.

1. Implementación de la clase Trabajador

```
package Entidades;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
```

```
public class Trabajador {

  //Atributos
  private String Cedula;
  private String Nombres;
  private String Apellido;
  private char Sexo;
  private Date FechaNaci;
  private Date FechaIngreso;
  //Metodos
  public Trabajador() {
  }
  public Trabajador(String Cedula, String Nombres, String Apellido, char
        Sexo, Date FechaNaci, Date FechaIngreso)
  {
     this.Cedula = Cedula;
     this.Nombres = Nombres;
     this.Apellido = Apellido;
     this.Sexo = Sexo;
     this.FechaNaci = FechaNaci;
     this.FechaIngreso = FechaIngreso;
  }
  public boolean Cumpleanios()
  {// verifica si esta de cumpleaños
     Calendar naci = new GregorianCalendar();
     naci.setTime(FechaNaci);
     Calendar today = new GregorianCalendar();
     today.setTime(new Date());
     //compara si nacio el mismo mes que today
     return naci.get(Calendar.MONTH) == today.get(Calendar.MONTH);
  }
  public int Antiguedad()
  {
     return CalcularAnios(FechaIngreso);
  }
  public int Edad() {
     return CalcularAnios(FechaNaci);
  }
  public int CalcularAnios(Date past)
  {
     Calendar p = new GregorianCalendar();
     p.setTime(past);//tiempo de la fecha pasada
     Calendar t = new GregorianCalendar();
     t.setTime(new Date()); //tiempo actual
     int anio = t.get(Calendar.YEAR) - p.get(Calendar.YEAR);//años=presente
           menos pasado
```

```
    if (t.get(Calendar.MONTH) < p.get(Calendar.MONTH))
    {
        anio--;
    }
    return anio;
}
public String MensajesBonos()
{

    String msg = "Bonos Aplicados:\n";
    if (Cumpleanios()) {
        msg += "- Bono Cumpleañero: 100\n";
    }
    if (Edad() >= 60) {
        msg += "- Bono por Edad: 50\n";
    }
    if (msg.equalsIgnoreCase("Bonos Aplicados:\n")) {
        msg = "Bonos Aplicados: Ninguno\n";
    }
    return msg;
}
public int Bono()
{
    int bono = 0;
    if (Cumpleanios()) {
        bono += 100;
    }
    if (Edad() >= 60) {
        bono += 50;
    }
    return bono;
}
public String getCedula()
{
    return Cedula;
}
public void setCedula(String newVal)
{
    Cedula = newVal;
}
public String getNombres()
{
    return Nombres;
}
public void setNombres(String newVal)
{
```

```
        Nombres = newVal;
    }
    public String getApellido()
    {
        return Apellido;
    }
    public void setApellido(String newVal)
    {
        Apellido = newVal;
    }

    public Date getFechaNaci()
    {
        return FechaNaci;
    }
    public void setFechaNaci(Date newVal)
    {
        FechaNaci = newVal;
    }
    public char getSexo()
    {
        return Sexo;
    }
    public void setSexo(char newVal)
    {
        Sexo = newVal;
    }
    public String ImprimirTrabajador()
    {

        return "Cédula: " + Cedula + "\n"
                + "Nombre: " + Nombres + " "+ Apellido + "\n"
                + "Sexo:" + Sexo + "\n"
                + "Edad :" + Edad() + "\n"
                + "Antiguedad en la compañia:" + Antiguedad() + "\n";
    }

}
```

2. Implementación de la clase Fijo

```
package Entidades;

import java.util.Date;
import javax.swing.JOptionPane;

public class Fijo extends Trabajador
```

```
{
   private double SalarioMensual;
   private double Anticipos;

   public Fijo()
   {
      super();
   }

   public Fijo(double SalarioMensual, double Anticipos, String Cedula, String
       Nombres, String Apellido, char Sexo, Date FechaNaci, Date
       FechaIngreso)
   {
      super(Cedula, Nombres, Apellido, Sexo, FechaNaci, FechaIngreso);
      this.SalarioMensual = SalarioMensual;
      this.Anticipos = Anticipos;

   }


   public void Imprimir()
   {
      JOptionPane.showMessageDialog(null,ImprimirTrabajador()
            +"Salario Mensual: " + SalarioMensual+ "\n"
            + "Anticipos Solicitados : " + Anticipos + "\n"
            + MensajesBonos()
            + "Sueldo Total:" + CalcularSueldo()+ "\n");
   }

   public double CalcularSueldo()
   {
      return (SalarioMensual -(SalarioMensual * 0.0932)) - Anticipos +
                        Bono();
   }

   public double getSalarioMensual()
   {
      return SalarioMensual;
   }

   public void setSalarioMensual(double newVal)
   {
      SalarioMensual = newVal;
   }

   public double getAnticipos()
   {
```

```
    return Anticipos;
  }

  public void setAnticipos(double newVal)
  {
    Anticipos = newVal;
  }

}
```

3. Implementación de la clase Contratado

```
package Entidades;

import java.util.Date;
import javax.swing.JOptionPane;

public class Contratado extends Trabajador
{

  private double Tarifa;
  private double Horas;
  public Contratado()
  {
    super();
  }


  public Contratado(double Tarifa, double Horas, String Cedula, String
    Nombres, String Apellido, char Sexo, Date FechaNaci, Date FechaIngreso
    )
  {
    super(Cedula, Nombres, Apellido, Sexo, FechaNaci, FechaIngreso);
    this.Tarifa = Tarifa;
    this.Horas = Horas;
  }
  public double getTarifa()
  {
    return Tarifa;
  }

  public void setTarifa(double newVal)
  {
    Tarifa = newVal;
  }

  public double getHoras()
```

```
    {
        return Horas;
    }

    public void setHoras(int newVal)
    {
        Horas = newVal;
    }

    public void Imprimir()
    {

        JOptionPane.showMessageDialog(null, ImprimirTrabajador()
                + "Horas Trabajadas: " + Horas + "\n"
                + "Tarifa por hora " + Tarifa + "\n"
                + MensajesBonos()
                + "Sueldo total: " + CalcularSueldo());
    }
    public double CalcularSueldo()
    {

        double sueldo = Horas * Tarifa;

        if (Horas > 160) //descuento del 9.32% del seguro social
        {
            sueldo -= sueldo * 0.0932;
        }

        return sueldo + Bono();
    }

}
```

4. Implementación de la clase Comisión

```
package Entidades;

import java.util.Date;
import javax.swing.JOptionPane;

public class Comision extends Trabajador
{

    protected double Porcentaje;
    protected double Ventas;

    public Comision() {
```

```
    super();
  }
  public Comision(double Comision, double Ventas, String Cedula, String
        Nombres, String Apellido, char Sexo, Date FechaNaci, Date
        FechaIngreso)
  {
    super(Cedula, Nombres, Apellido, Sexo, FechaNaci, FechaIngreso);
    this.Porcentaje = Comision;
    this.Ventas = Ventas;
  }
  public double getComision()
  {
    return Porcentaje;
  }
  public void setComision(double newVal)
  {
    Porcentaje = newVal;
  }
  public double getVentas()
  {
    return Ventas;
  }
  public void setVentas(double newVal)
  {
    Ventas = newVal;
  }
  public void Imprimir()
  {
    JOptionPane.showMessageDialog(null,ImprimirTrabajador()
        +"Porcentaje de comisión: " + Porcentaje + "%\n"
        + "Total de Ventas : " + Ventas + "\n"
        + MensajesBonos()
        + "Sueldo Total:" + CalcularSueldo());
  }
  public double CalcularSueldo()
  {
    return ((Porcentaje / 100) * Ventas)+ Bono();
  }
}
```

5. Implementación de la clase Comisionado

```
package Entidades;

import java.util.Date;
import javax.swing.JOptionPane;
```

```
public class Comisionado extends Comision
{

   private double SueldoBase;

   public Comisionado()
   {
      super();
   }

   public Comisionado(double SueldoBase, double Comision, double Ventas,
         String Cedula, String Nombres, String Apellido, char Sexo, Date
         FechaNaci, Date FechaIngreso)
   {
      super(Comision, Ventas, Cedula, Nombres, Apellido, Sexo, FechaNaci,
            FechaIngreso);
      this.SueldoBase = SueldoBase+(SueldoBase*0.10);
      //sueldo base mas el 10%
   }

   public double getSueldoBase()
   {
      return SueldoBase;
   }

   public void setSueldoBase(double newVal)
   {
      SueldoBase = newVal;
   }

   public double CalcularSuel()
   {
      return CalcularSueldo() + SueldoBase;
   }

   @Override
   public void Imprimir()
   {
       JOptionPane.showMessageDialog(null,ImprimirTrabajador()
            +"Porcentaje de comisión: " + Porcentaje + "%\n"
            + "Total Ventas : " + Ventas + "\n"
            + "Sueldo Base : " + SueldoBase + "\n"
            +MensajesBonos()
            + "Sueldo Total :" + CalcularSueldo() + "\n");

   }
```

```
}
```

6. Diseño del formulario frmSistema

Figura 22: Diseño del formulario JFrame frmSistema.
Fuente: Elaboración propia.

7. Implementación del JFrame frmSistema.

```
package Formulario;

import Entidades.Comision;
import Entidades.Comisionado;
import Entidades.Contratado;
import Entidades.Fijo;
import java.util.Date;
import javax.swing.JOptionPane;


public class frmSistema extends javax.swing.JFrame
{

   public frmSistema() {
      initComponents();
   }

   public void mostrartxt(boolean n)
   {
      jLabel10.setEnabled(n);
      jTextField6.setEnabled(n);
```

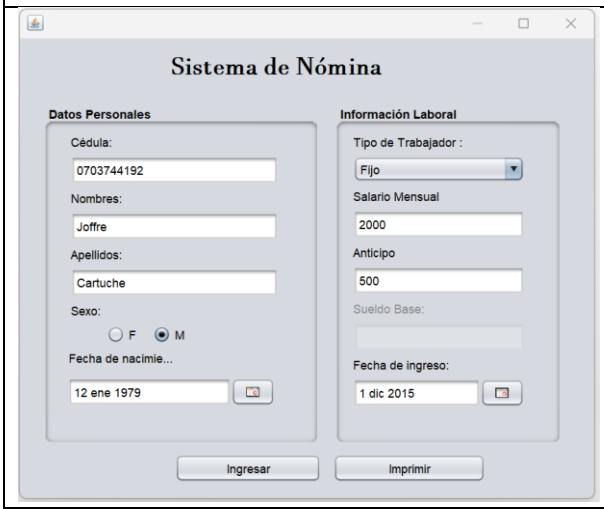
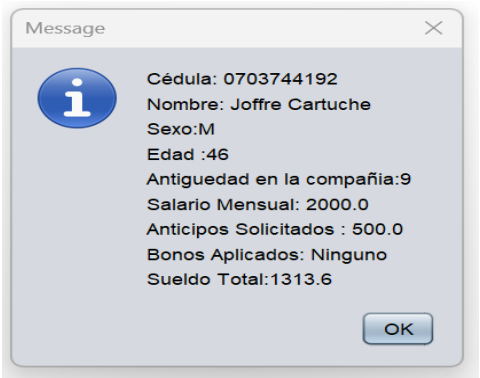
```
}
private void jComboBox1ActionPerformed(java.awt.event.ActionEvent evt)
{

 if(jComboBox1.getSelectedItem().toString().equalsIgnoreCase("Fijo"))    {

      jLabel8.setText("Salario Mensual");
      jLabel9.setText("Anticipo");
      mostrartxt(false);

 }else if
    (jComboBox1.getSelectedItem().toString().equalsIgnoreCase("Contrata
    do"))
{
      jLabel8.setText("Tarifa por hora");
      jLabel9.setText("Horas Trabajadas");
      mostrartxt(false);

 }else if
    (jComboBox1.getSelectedItem().toString().equalsIgnoreCase("Comisió
    n"))
{
      jLabel8.setText("Porcentaje de Comisión");
      jLabel9.setText("Ventas Brutas");
      mostrartxt(false);

  }else
  {
      jLabel8.setText("Porcentaje de Comisión");
      jLabel9.setText("Ventas");
      mostrartxt(true);

   }

}//jComboBox1ActionPerformed

// Variables globales
Fijo of;
Contratado oc;
Comision ocm;
Comisionado ocmo;
String tip;
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{

   String cedula = jTextField1.getText();
```

```
    String nombres = jTextField2.getText();
    String apellidos = jTextField3.getText();
    char sexo = 'M';
    if (jRadioButton1.isSelected())
        {
       sexo = 'F';
    }
    double v1 = Double.parseDouble(jTextField4.getText());
    double v2 = Double.parseDouble(jTextField5.getText());
    Date Fechanaci = jDateChooser1.getDate();
    Date Fechaingreso = jDateChooser2.getDate();
    tip = jComboBox1.getSelectedItem().toString();
    switch(tip)
    {
       case "Fijo":
          if (v2 > (v1*0.50)
          {
              JOptionPane.showMessageDialog(null,"El anticipo no puede ser
                        mayor al salario");

          }
          else
          {
             of = new Fijo(v1, v2, cedula, nombres, apellidos, sexo, Fechanaci,
                        Fechaingreso);

          }
          break;
       case "Comisionados":
          double v3 = Double.parseDouble(jTextField6.getText());
          ocmo = new Comisionado(v3, v1, v2, cedula, nombres, apellidos,
                        sexo, Fechanaci, Fechaingreso);
          break;

       case "Comisión":
          ocm = new Comision(v1, v2, cedula, nombres, apellidos, sexo,
                        Fechanaci, Fechaingreso);
          break;
       case "Contratado":
           oc = new Contratado(v1, v2, cedula, nombres, apellidos, sexo,
                    Fechanaci, Fechaingreso);

    }

}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)
```

```
  {
     switch(tip)
     {
        case "Fijo":
             of.Imprimir();
             break;
        case "Comisionados":

            ocmo.Imprimir();
            break;

        case "Comisión":
            ocm.Imprimir();
            break;
        default:
           oc.Imprimir();
            break;
     }
  }
}
```

8. Salida del sistema.

Tabla 15: Presentación de resultados de sistema de nómina.

Interfaz de entrada	Interfaz de salida
	

RESUMEN DEL CAPITULO

En este capítulo, exploramos a fondo el segundo y tercer pilar fundamental de la programación orientada a objetos: la herencia y el polimorfismo. La herencia fue presentada como un mecanismo esencial para la reutilización de código, permitiendo que una clase hija aproveche atributos y métodos definidos en una clase padre. Se introdujeron los conceptos de herencia simple y compuesta, resaltando cómo Java utiliza *extends* para herencia directa de una clase e interfaces para simular herencia múltiple sin ambigüedades.

También se explicó cómo propagar atributos y métodos, el papel del modificador *private* y el uso de métodos *getters* y *setters* para respetar el principio de encapsulamiento. Se abordó el uso de super para acceder a constructores y métodos de la clase base, así como *this* para sobrecargar constructores dentro de la misma clase.

En cuanto al polimorfismo, se estudió su papel como facilitador del diseño flexible, permitiendo que distintos objetos respondan de manera específica a métodos compartidos. Se analizaron los conceptos de sobrecarga y sobrescritura de métodos, su sintaxis y utilidad en la práctica. A través de ejemplos reales y diagramas *UML*, se mostró cómo el polimorfismo permite trabajar con listas heterogéneas de objetos, manteniendo una interfaz común.

Finalmente, se destacaron las ventajas y desventajas tanto de la herencia como del polimorfismo, ofreciendo al estudiante una visión crítica y práctica de cuándo y cómo aplicar estos mecanismos en el desarrollo de software.

Este capítulo sienta las bases para el diseño de sistemas modulares, extensibles y mantenibles, aspectos clave en la ingeniería de software profesional.

AUTOEVALUACIÓN

En todas las preguntas de selecciona múltiple un literal es la respuesta correcta.

1. ¿Cuál es la palabra clave para aplicar herencia en Java?
   a) interface
   b) implements
   c) inherit
   d) extends

2. ¿Qué tipo de herencia permite una clase padre con múltiples hijos?
   a) Múltiple
   b) Jerárquica
   c) Simple
   d) Complicada

3. ¿Qué permite super()?
   a) Llamar al constructor padre
   b) Crear un método
   c) Eliminar atributos
   d) Declarar objetos

4. ¿Qué es sobrescritura?
   a) Crear un nuevo método
   b) Heredar atributos
   c) Modificar comportamiento heredado
   d) Repetir un método

5. ¿Qué anota el método sobrescrito?
   a) @Super
   b) @Override
   c) @Extends
   d) @Inherited

6. ¿Qué permite la sobrecarga?
   a) Variar parámetros
   b) Cambiar nombres
   c) Heredar clases
   d) Definir atributos

7. ¿Cuál es una ventaja del polimorfismo?
   a) Código duplicado
   b) Clases independientes
   c) Interfaces acopladas
   d) Comportamiento flexible

8. ¿Cuál de las siguientes es clase hija?
   a) public class Estudiante extends Persona
   b) public class Persona extends Estudiante
   c) class extends Persona
   d) class Persona

9. ¿Qué tipo de atributo no es accesible desde una subclase?
   a) protected
   b) public
   c) private
   d) static

10. ¿Qué se logra aplicando herencia y polimorfismo?
    a) Mayor duplicidad
    b) Complejidad innecesaria
    c) Reutilización y flexibilidad
    d) Acoplamiento rígido

EJERCICIO PROPUESTO

Caso de estudio congresistas

Declara una clase Procurador que herede de la clase Persona (IdCodigo, Cedula, Nombre, Apellido, FechaNacimiento, Estado), con un atributo provincia que representa, partido político, Declarar un método ProyectoPresentado. Crear dos clases concretas que hereden de Procurador: la clase Congresista y la clase Senador con sus respectivos métodos.
En el congreso se realiza el pago de sueldo de la siguiente manera:

- **Congresista,** que reciben un salario mensual, descontando 9.32 % del seguro social, adicional tiene una bonificación de $ 1000 por cada proyecto presentado (puede ser aprobado, revisión), cuando sea aprobado el proyecto, si no está aprobado no recibe dicha bonificación, puede actualizarse de revisión a aprobado.
- **Senador**, que perciben un sueldo mensual, más comisiones, descontando 9.32 % del seguro social, puede recibir un anticipo del 50% máximo de su sueldo mensual.

Si considera más atributos en las clases queda a responsabilidad del programador, se pide realizar lo siguiente:

- Diseñar el modelado de datos UML del problema planteado, generar el código en java.
- Crear las respectivas clases del problema que permita obtener el sueldo a cobrar según su tipo de procurador.

REFERENCIA BIBLIOGRÁFICA

Arora, A., & Suri, B. (2023). Effective use of polymorphism in Java. International Journal of Computer Applications, 185(8), 21-27.

Bailón, J., & Baltazar, A. (2021). Fundamentos de programación orientada a objetos. CIDE.

Cheng, Y., & Zhang, J. (2025). Code reusability and scalability using inheritance and polymorphism. Journal of Software Engineering Research, 14(2), 101-115.

Deitel, P., & Deitel, H. (2022). Java como programar (11.ª ed.). Person.

Gamma, E., & Helm, R. (2021). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Horstmann, C. (2022). Core Java Fundamentals (12.ª ed). Pearson.

Li, L., & Wang, X. (2023). Teaching object-oriented principles through real-world Java projects. ACM Transactions on Computing Education, 45, 23(3).

Martin, R. (2021). Clean Code Revisited: Best Practices for Maintainable Code (Prentice Hall).

Pérez, D., & Ramírez, L. (2024). Modelado UML y programación Java en ambientes educativos. Revista de Tecnología y Software.

Sommerville, I. (2021). Introducción a la ingeniería de software moderna (10.ª ed.). Pearson.

CAPÍTULO IV: CLASES ABSTRACTAS E INTERFACES

Objetivos de la unidad

- Comprender que son las clases abstractas y las interfaces en Java, identificando sus características fundamentales y sus diferencias clave dentro del paradigma de la programación orientada a objetos.
- Aplicar el *UML* para representar mediante diagramas, las clases abstractas e interfaces identificando sus atributos, métodos y relaciones de generalización y realización, con el fin de fortalecer el análisis y diseño de *software*.
- Desarrollar programas orientados a objetos que declaren prototipos de métodos en una clase mediante el uso de clases abstractas e interfaces, con el propósito de fomentar el razonamiento lógico y la reutilización de código para el diseño modular en la programación Java.

En este cuarto capítulo se centra en el cuarto pilar de la *POO*: clases abstractas, estudiaremos dos herramientas fundamentales del lenguaje Java que permiten organizar, reutilizar y mantener mejor el código: las clases abstractas y las interfaces. Veremos cómo estas estructuras nos ayudan a crear programas más flexibles y fáciles de extender. Aprenderemos cuándo y cómo utilizarlas, cómo representarlas mediante diagramas *UML*, y las diferencias clave entre ellas.

Preguntas de Enfoque

- ¿Qué son las clases abstractas y qué papel cumplen en la *POO*?
- ¿Cuáles son las diferencias clave entre una clase abstracta y una interfaz?
- ¿Cuándo conviene usar una clase abstracta y cuándo una interfaz?
- ¿Qué ventajas ofrece el uso de métodos abstractos?
- ¿Qué rol cumple el polimorfismo en el uso de clases abstractas e interfaces (Herrera, 2024)?

"Una buena abstracción es la que no solo oculta los detalles, sino que también revela la esencia". Martin Fowler

4.1 Introducción a la clase abstracta

Cuando diseñamos un programa en Java, muchas veces necesitamos crear estructuras que compartan características comunes pero que al mismo tiempo tengan comportamientos distintos. Por ejemplo, un empleado y un consultor tienen nombre y dirección, pero calculan su sueldo de forma diferente. Del mismo modo, un círculo y un cuadrado pueden calcular su área, pero con fórmulas distintas.

Para resolver estas situaciones, el lenguaje Java nos ofrece dos herramientas muy útiles: las clases abstractas y las interfaces. Ambas permiten crear modelos generales y establecer reglas que otras clases deben seguir. Las clases abstractas se utilizan cuando se desea compartir código entre varias clases relacionadas, mientras que las **i**nterfaces definen comportamientos comunes que cualquier clase puede adoptar, sin importar su jerarquía.

Definición y propósito de clase abstracta

Según (Roldán & Meneses, 2023). Una clase abstracta es una clase que no puede ser instanciada directamente y que sirve como plantilla base para otras clases. Se utiliza cuando varias clases comparten comportamientos y atributos, pero requieren implementar ciertos métodos a su manera. Estas clases pueden contener métodos abstractos**,** que son como instrucciones incompletas: se definen solo por su nombre y tipo, pero no tienen contenido, por lo que deben ser implementados por las clases hijas. También pueden tener métodos concretos**,** que sí incluyen una implementación completa y pueden ser usados tal como están en las clases derivadas.

¿Cuándo utilizar clases abstractas?

- Cuando varias clases comparten comportamiento común: Por ejemplo, todas las clases que representan empleados podrían heredar de una clase abstracta Empleado que contenga métodos como CalcularEdad(), CalcularAntiguedad(), CalcularSueldo(), etc.
- Cuando deseas establecer una estructura base: Las clases abstractas definen métodos que obligan a las subclases a proporcionar su propia implementación.
- Cuando necesitas combinar comportamiento general y comportamiento específico: Puedes definir métodos ya implementados junto con otros que deben completarse en las subclases.

- Cuando no tiene sentido instanciar la clase base directamente: Es útil en casos donde una clase representa algo muy general (como Animal, Documento, o Vehículo).
- Cuando quieres facilitar la evolución del diseño: Nuevas funcionalidades pueden agregarse a la clase base sin afectar las subclases, que automáticamente heredan lo nuevo si lo necesitan.

Características y uso de métodos abstractos y concretos

- Las clases abstractas pueden tener atributos, constructores y métodos concretos.
- Una clase puede ser declarada como abstracta, aunque no tenga ningún método abstracto. Esto significa que no es obligatorio incluir métodos sin implementación para que una clase sea abstracta. En muchos casos, se usa esta técnica para evitar que la clase sea instanciada directamente, es decir, que sirva únicamente como base para otras clases que sí estarán completas. Este enfoque ayuda a organizar mejor el diseño del software y a evitar errores en tiempo de ejecución.

Ventajas de las clases abstractas

- ✓ **Reutilización de código**: permiten definir métodos y atributos comunes que todas las subclases heredan automáticamente.
- ✓ **Consistencia en el diseño**: obligan a las subclases a implementar ciertos métodos clave, garantizando un comportamiento uniforme.
- ✓ **Flexibilidad**: combinan métodos abstractos (incompletos) y concretos (implementados), lo que facilita equilibrar lo general con lo específico.
- ✓ **Organización del software**: ayudan a estructurar jerarquías claras y lógicas, evitando instanciaciones indebidas de clases demasiado generales.
- ✓ **Facilidad de mantenimiento**: nuevas funcionalidades pueden añadirse a la clase base y heredarse automáticamente en las subclases.

Desventajas de las clases abstractas

- ✗ **Limitación de herencia**: en Java solo se puede heredar de una clase abstracta (herencia simple), lo que restringe la flexibilidad frente a las interfaces.
- ✗ **Posible rigidez del diseño**: una jerarquía mal diseñada puede generar dependencias fuertes y dificultar la extensión del sistema.
- ✗ **Sobrecarga innecesaria**: si una subclase no necesita todos los métodos heredados, puede terminar con código redundante.

- **Menor compatibilidad con clases no relacionadas**: a diferencia de las interfaces, las clases abstractas no permiten que clases de jerarquías diferentes compartan fácilmente comportamientos comunes.
- **Mayor complejidad para principiantes**: el concepto de "método sin implementación" puede ser confuso en etapas iniciales de aprendizaje

4.2 Creación de clases abstractas

Las clases abstractas permiten una jerarquía clara donde se puede forzar la implementación de comportamientos comunes(Jiménez, 2024) . En java la palabra clave abstract permite crear clase abstractas.

```
public abstract class Trabajador
{
  // Atributos

  //Métodos concretos

  //Métodos abstractos

}
```

4.3 Diagrama de clase abstracta en UML

En UML (Lenguaje Unificado de Modelado), una clase abstracta se representa de forma similar a una clase normal, pero con una diferencia importante: el nombre de la clase aparece en letra cursiva o itálica, lo que indica que no puede ser instanciada directamente. Además, los métodos abstractos también se escriben en cursiva, para señalar que no tienen implementación y deben ser definidos por las clases hijas. La relación de herencia entre clases se muestra mediante una flecha con cabeza blanca (no rellena) que va desde la clase hija hacia la clase padre, tal como se observa en la Figura 25.

Figura 23: Diagrama UML de clase abstracta.
Fuente: Elaboración propia.

La clase Trabajador, mostrada en la parte superior del diagrama UML, es una clase abstracta que actúa como superclase para Empleado y Consultor. A continuación, te explico sus características clave y su función en el diseño orientado a objetos:

Características de la clase Trabajador

- **Atributos compartidos (heredables)**: la clase Trabajador define atributos comunes que todo tipo de trabajador debe tener, como cedula, nombre, fecha Nacimiento, dirección, teléfono. Estos atributos son heredados automáticamente por las clases Empleado y Consultor.

- **Constructores sobrecargados**: la clase trabajador tiene dos constructores.

  - Uno vacío: Trabajador()
  - El otro con parámetros: Trabajador(String, String, Date, String, String)

  Esto permite inicializar objetos de subclases (Empleado, Consultor) con datos básicos del trabajador, reutilizando código más los atributos propios.

- **Métodos abstractos**: Se tiene declarado dos métodos abstractos

- Imprimir():void y CalcularSueldo():double
- Estos métodos no tienen implementación son abstractos, lo que obliga a las clases hijas (Empleado y Consultor) a implementarlo con su propia versión, según su necesidad.

Esto favorece el polimorfismo, ya que cada tipo de trabajador puede "Imprimirse" y "calcular el sueldo" de forma diferente.

- **Métodos concretos**: son los métodos propios de la clase Trabajador que están implementados, por ejemplo:

- Edad(): es un método que calcula la edad a través de la fecha de nacimiento
- ImprimirTrabajador() para imprimir los datos de la clase Trabajador
  Estos métodos pueden ser usados directamente por las subclases (Empleado y consultor), esto evita duplicación de código y centraliza la lógica común.

¿Por qué es abstracta la clase Trabajador?

Aunque tiene métodos implementados ImprimirTrabajador(), no se puede instanciar directamente. Es decir, no puedes crear objetos del tipo Trabajador, solo puedes crear objetos de las subclases (Empleado, Consultor) que implementen los métodos abstractos.

- **Generalización**: Trabajador encapsula lo que es común entre Empleado y Consultor.
- **Especialización**: Cada subclase agrega sus propios atributos y comportamientos (comisiones, tarifaHoraria, etc.).
- **Polimorfismo**: El método abstracto imprimirTrabajador() puede llamarse de manera genérica desde un arreglo de Trabajador, y se ejecutará la versión correspondiente según el tipo real del objeto.

La clase Trabajador representa un diseño bien estructurado que aplica herencia, abstracción y polimorfismo**.** Define lo que todo trabajador debe tener y hacer, pero deja detalles específicos a las subclases. Este enfoque reduce la redundancia, facilita el mantenimiento del código y mejora la organización del sistema.

4.4 Implementación de clases abstractas

Caso de estudio: pago de sueldo.

Para el siguiente caso de estudio utilizaremos el modelado UML de la clase abstractas trabajador de la Figura 25. Trabajador es una clase genérica que sirve para almacenar datos como la cedula, nombre, fecha de nacimiento, dirección y el número de teléfono.

- Empleado es una clase especializada para representar los empleados que tienen un sueldo mensual, más comisiones, menos impuestos cancelados.
- Consultor es una clase especializada para representar a aquellos trabajadores que cobran por horas (por ejemplo, registra el número de horas que ha trabajado un consultor y su tarifa horaria).
- Las clases **Empleado** y **Consultor**, además de los atributos y de las operaciones que definen, heredan de Trabajador todos sus atributos y operaciones.
- Implementar el sistema con los requerimientos en java.

Ejercicio 9: Sistema de pago de sueldo

Crear un proyecto con el nombre 09_SistemaPagoSueldo, crear dos Packages Entidades y Formularios. En el Package Entidades crear las clase Trabajador, Empleado y consultor con todos los atributos y métodos del enunciado del problema.

1. Implementación de la clase Trabajador.

```
package Entidades;


import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public abstract class Trabajador
{

   private String Cedula;
   private String Nombre;
   private Date FechadeNacimiento;
   private String Direccion ;
   private String Telefono ;
   public Trabajador ()
   {
   }
```

```
  public Trabajador(String Cedula, String Nombre, Date
          FechadeNacimiento, String Direccion, String Telefono)
  {
     this.Cedula = Cedula;
     this.Nombre = Nombre;
     this.FechadeNacimiento = FechadeNacimiento;
     this.Direccion = Direccion;
     this.Telefono = Telefono;
  }
  public int Edad()
  {
     Calendar cal = new GregorianCalendar();
     cal.setTime(FechadeNacimiento);
     Calendar hoy = new GregorianCalendar();
     hoy.setTime(new Date());
     int  anios = hoy.get(Calendar.YEAR) - cal.get(Calendar.YEAR);
     if ((hoy.get(Calendar.MONTH) < cal.get(Calendar.MONTH)) ||
          (hoy.get(Calendar.MONTH) == cal.get(Calendar.MONTH) &&
          hoy.get(Calendar.MONTH) < cal.get(Calendar.MONTH)))
     {
        anios--;
     }
     return anios;

  }
  public String ImprimirTrabajador()
  {
        return  "Cedula : "+this.Cedula+"\n"+
           "Nombre :"+this.Nombre+"\n"+
           "Direccion: "+this.Direccion+"\n"+
           "Telefono: "+this.Telefono+"\n"+
           "Edad :" + Edad()+"\n";
  }
  public abstract double CalcularSueldo();
  public abstract void Imprimir();

}
```

2. Implementación de la clase Empleado.

```
package Entidades;

import java.util.Date;
import javax.swing.JOptionPane;
```

```
public class Empleado extends Trabajador {

  private double Comisiones;
  private double Impuesto;
  private double Sueldo;
  public Empleado()
  {
     super();
  }
  public Empleado(String ced,String nom, Date fn, String dir, String tel,
          double com, double imp,double sue )
  {
     super(ced,nom,fn,dir,tel);
     this.Comisiones =com;
     this.Impuesto=imp;
     this.Sueldo = sue;
  }
  public double getComisiones()
  {
     return Comisiones;
  }
  public void setComisiones(double Comisiones)
  {
     this.Comisiones = Comisiones;
  }
  public double getImpuesto()
  {
     return Impuesto;
  }

    public void setImpuesto(double Impuesto)
  {
     this.Impuesto = Impuesto;
  }
  public double getSueldo()
  {
     return Sueldo;
  }
  public void setSueldo(double Sueldo)
  {
     this.Sueldo = Sueldo;
  }

  @Override
  public double CalcularSueldo()
  {
    return Sueldo - Impuesto + Comisiones;
```

```
    }

    @Override
    public void Imprimir()
    {
        JOptionPane.showMessageDialog(null, ImprimirTrabajador()+"\n"+
                    "Sueldo: "+this.Sueldo+"\n"+
                    "Impuesto: "+this.Impuesto+"\n"+
                    "Comisiones :" + this.Comisiones +"\n"+
                    "Sueldo cobrar: "+CalcularSueldo()+"\n");
    }
}
```

3. Implementación de la clase Consultor.

```
package Entidades;

import java.util.Date;
import javax.swing.JOptionPane;
public class Consultor extends Trabajador
{
    private double HorasTrabajadas ;
    private double TarifaHora;
    public Consultor()
    {
        super();
    }
    public Consultor(String ced,String nom, Date fn, String dir, String tel,
            double hor, double cos)
    {
            super(ced,nom,fn,dir,tel);
            this.HorasTrabajadas=hor;
            this.TarifaHora=cos;
    }



    public double getHorasTrabajadas()
    {
        return HorasTrabajadas;
    }
    public void setHorasTrabajadas(double HorasTrabajadas)
    {
        this.HorasTrabajadas = HorasTrabajadas;
    }
```

```
  public double getTarifahora()
  {
     return TarifaHora;
  }
  public void setTarifahora(double Tarifahora)
  {
     this.TarifaHora = Tarifahora;
  }
  @Override
  public double CalcularSueldo()
  {
      return HorasTrabajadas * TarifaHora;
  }
  @Override
  public void Imprimir()
  {
     JOptionPane.showMessageDialog(null, ImprimirTrabajador()+"\n"+
              "Horas Trabajadas: "+this.HorasTrabajadas+"\n"+
              "Tarifa por Hora: "+this.TarifaHora+"\n"+
              "Sueldo cobrar: "+CalcularSueldo()+"\n");
  }
}
```

4. Implementación del formulario frmTrabajador en el paquete Formularios.

Figura 6: Diseño del formulario JFrame frmTrabajador.
Fuente: Elaboración propia.

```
package Formulario;

import Entidades.Consultor;
import Entidades.Empleado;
import java.util.Date;

public class frmTrabajador extends javax.swing.JFrame
{

   public frmTrabajador() {
      initComponents();
   }
   Empleado oe;
   Consultor oc;
   private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
   {
      String ced = jTextField1.getText();
      String nom = jTextField1.getText();
      Date fecha = jDateChooser1.getDate();
      String dir = jTextField3.getText();
      String tel = jTextField4.getText();
      double v1 = Double.parseDouble(jTextField5.getText());
      double v2 = Double.parseDouble(jTextField6.getText());
      if (jComboBox1.getSelectedItem().toString().equals("Empleado"))
      {
         double imp = Double.parseDouble(jTextField7.getText());
         oe = new Empleado(ced,nom,fecha,dir,tel,v1,v2,imp);
      }
      if (jComboBox1.getSelectedItem().toString().equals("Consultor"))
                  {
         oc = new Consultor(ced,nom,fecha,dir,tel,v1,v2);
      }

   }

   private void jButton2ActionPerformed(java.awt.event.ActionEvent evt)
   {
      if (jComboBox1.getSelectedItem().toString().equals("Empleado")) {
         oe.Imprimir();
      }
      if (jComboBox1.getSelectedItem().toString().equals("Consultor"))
                  {
         oc.Imprimir();

      }
   }
```

```
  private void jComboBox1ActionPerformed(java.awt.event.ActionEvent evt)
    {
     // TODO add your handling code here:
     if (jComboBox1.getSelectedItem().toString().equals("Empleado"))
                {
        jLabel8.setText("Sueldo Mensual:");
        jLabel9.setText("Comisiones:");
        jTextField7.setEnabled(true);
     }
     if (jComboBox1.getSelectedItem().toString().equals("Consultor"))
     {
        jLabel8.setText("Horas Trabajadas:");
        jLabel9.setText("Tarifa por Hora:");
        jTextField7.setEnabled(false);
     }
   }
}
```

4.5 Interface

Es una estructura que define un conjunto de métodos abstractos que una clase debe implementar obligatoriamente. Representa un contrato porque establece un acuerdo formal: cualquier clase que la implemente se compromete a proporcionar una implementación concreta de todos los métodos declarados en la interfaz. Desde Java 8, las interfaces pueden incluir además métodos default, que tienen una implementación por defecto, y métodos static, que pertenecen a la propia interfaz y no a las clases que la implementan. Este enfoque promueve una mayor abstracción e independencia entre el qué hace una clase (la interfaz lo define) y el cómo lo hace (la clase concreta lo implementa), facilitando así la flexibilidad, el desacoplamiento y la reutilización del código (Guzmán, 2021).

Las interfaces no declaran métodos constructores, debido a que sólo se declaran constantes, describe un grupo de funciones relacionadas que pueden pertenecer a cualquier clase o estructura.

Ventajas de la interface

- ✓ **Herencia múltiple de comportamientos**: una clase puede implementar varias interfaces, lo que otorga gran flexibilidad al diseño.
- ✓ **Alto grado de abstracción**: separan completamente el *qué hace* una clase del *cómo lo hace*, promoviendo el desacoplamiento.
- ✓ **Facilitan el polimorfismo**: distintas clases que implementan la misma interfaz pueden ser tratadas como el mismo tipo.

- ✓ **Mayor reutilización y escalabilidad**: permiten que clases de jerarquías distintas compartan comportamientos comunes sin necesidad de forzar relaciones de herencia.
- ✓ **Evolución del lenguaje**: desde Java 8, las interfaces pueden contener métodos default y static, lo que reduce la duplicación de código y mejora la compatibilidad hacia atrás.

Desventajas de la interface

- ✗ **No permiten atributos de instancia**: solo se pueden declarar constantes, lo que limita su uso para compartir estado entre clases.
- ✗ **Complejidad en grandes sistemas**: demasiadas interfaces pueden dificultar la comprensión y mantenimiento del código.
- ✗ **Mayor esfuerzo inicial**: obligan a implementar todos los métodos declarados, incluso si algunos no se utilizan en ciertas clases.
- ✗ **Riesgo de abuso**: si se crean interfaces muy pequeñas o demasiado específicas, se fragmenta el diseño y se pierde claridad.
- ✗ **Menor compatibilidad con código heredado**: en versiones anteriores a Java 8, cualquier cambio en una interfaz obligaba a modificar todas las clases que la implementaban.

4.6 Diagrama de interface en UML

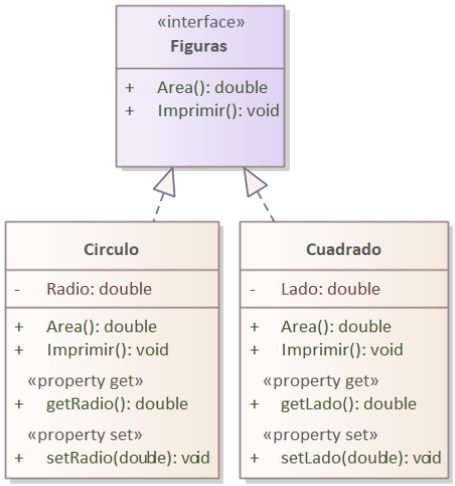


Figura 24: Diagrama UML de la interface Figuras.
Fuente: Elaboración propia.

La interfaz de Figuras 28 representa un contrato, no tiene instrucciones completas, sino solo una lista de métodos que las clases Circulo y Cuadrado obligatoriamente deben escribir el código para esos métodos. En este caso, la

interfaz indica que todas las figuras deben poder calcular su área y deben poder imprimirse.

Las clases Circulo y Cuadrado implementan la interfaz Figuras, lo que significa que están obligadas a definir cómo calcular el área y cómo imprimirse. Cada clase tiene su propia fórmula y forma de hacerlo, según su figura geométrica.

¿Qué significan las flechas?

Las flechas con líneas punteadas y una cabeza blanca indican que Circulo y Cuadrado están implementando la interfaz Figuras. No es una relación de herencia como con clases abstractas, sino una relación de **implementación de interfaz**.

4.7 Creación de interfaces

Pare crear una interface utilizamos la palabra reservada **interface** y luego los métodos abstractos.

```
public interface Figuras {
        public double Area();
        public void Imprimir();

}
```

4.8 Implementación de interfaces

Caso de estudio: Calcular el área de una figura geométrica.
Para el siguiente caso de estudio utilizaremos el modelado *UML* de la interface Figuras, la clase Circulo y Cuadrado de la Figura 26.

Ejercicio 10: Calcular el área del círculo o cuadrado

Crear un proyecto con el nombre 10_FiguraGeometrica, crear dos Packages Entidades y Formularios. En el Package Entidades crear la interface Figuras, la clase Circulo, Cuadrado con todos los atributos y métodos del enunciado del problema.

1. Creación de la interface Figura.

```
package Entidades;

public interface Figuras
{
        public double Area();
        public void Imprimir();
}
```

2. Creación de la clase Circulo implementando de la interface Figuras.

```
package Entidades;

import javax.swing.JOptionPane;

public class Circulo implements Figuras {

        private double Radio;
        public Circulo(double r)
        {
        this.Radio = r;
        }
        public Circulo()
        {

        }
        public double getRadio()
        {
                return Radio;
        }

        public void setRadio(double newVal)
        {
                Radio = newVal;
        }

    @Override
        public double Area()
        {
                return Math.PI * Radio * Radio;
        }
     @Override
     public void Imprimir()
        {
```

```
        JOptionPane.showMessageDialog(null,"Area  :" + Area());
    }
}
```

3. Creación de la clase Cuadrado implementando de la interface Figuras.

```
package Entidades;

import javax.swing.JOptionPane;

public class Cuadrado implements Figuras {

    private double Lado;
    public Cuadrado()
    {
    }
    public Cuadrado(double l)
    {
    this.Lado = l;
    }
    public double getLado()
    {
        return Lado;
    }
    public void setLado(double newVal)
    {
        Lado = newVal;
    }
   @Override
    public double Area()
    {
        return this.Lado * this.Lado;
    }
  @Override

  public void Imprimir()
    {
    JOptionPane.showMessageDialog(null,"Área del cuadrado es :" +
            Area());
    }

}
```

4. Implementación del formulario frmFigura en el paquete Formularios.

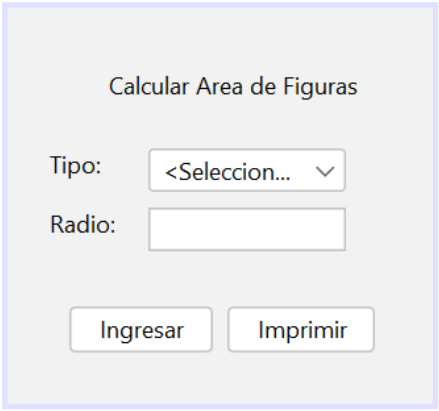


Figura 25: Diseño del formulario JFrame frmFigura.
Fuente: Elaboración propia.

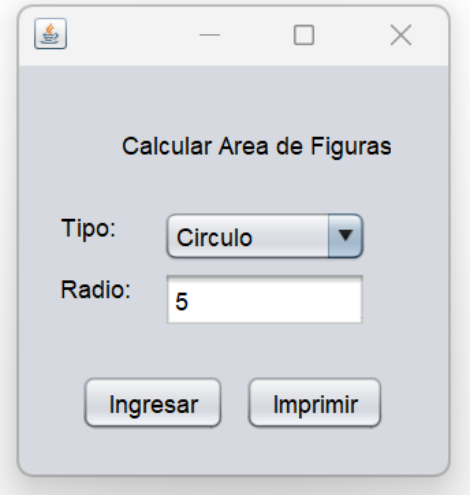
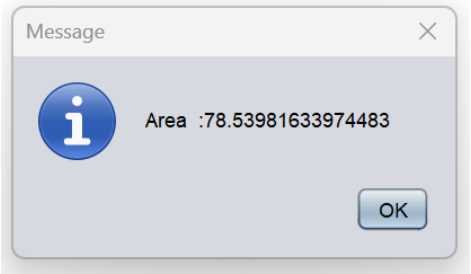
```
package Formulario;

import Entidades.Circulo;
import Entidades.Cuadrado;
import javax.swing.JOptionPane;

public class frmFiguras extends javax.swing.JFrame
{

   public frmFiguras() {
      initComponents();
   }

  // Variables globales
   Cuadrado occ;
   Circulo oc;

   private void cmbTipoActionPerformed(java.awt.event.ActionEvent evt)
     {
      if(cmbTipo.getSelectedItem().equals("Circulo"))
      {
         jLabel8.setText("Radio:");
      }
      if(cmbTipo.getSelectedItem().equals("Cuadrado"))
      {
         jLabel8.setText("Lado:");
      }

   }
```

```
private void btnIngresarActionPerformed(java.awt.event.ActionEvent evt)
      {

 if( !txtValor.getText().equals("") && jComboBox1.getSelectedIndex()>= 0
                    )
 {
  try{
     double v = Double.parseDouble(txtValor.getText());
     if(cmbTipo.getSelectedItem().equals("Circulo"))
     {
        oc = new Circulo(v);

     }
     if(cmbTipo.getSelectedItem().equals("Cuadrado"))
     {

        occ= new Cuadrado(v);
      }
      }catch(Exception ex)
      {
         JOptionPane.showMessageDialog(null, ex.getMessage());
      }

   }
   else
   JOptionPane.showMessageDialog(null,"Ingrese correctamente datos");
}

private void btnImprimirActionPerformed(java.awt.event.ActionEvent evt)
{

  try{

     if(cmbTipo.getSelectedItem().equals("Circulo"))
     {
        oc.Imprimir();

     }
     else
     {
        occ.Imprimir();
                    }
      }catch(Exception ex)
      {
         JOptionPane.showMessageDialog(null, ex.getMessage());
      }
```

```
    }
}
```

5. Salida del sistema.

Tabla 16: Presentación de resultados de clase Figuras

Interfaz de entrada	Interfaz de salida
	

4.9 Comparación: clases abstractas vs interfaces

Aspecto	Clases Abstractas	Interfaces
Definición	Plantilla base que puede contener métodos abstractos (sin implementación) y métodos concretos (con implementación). No se puede instanciar directamente.	Estructura que define un contrato: declara métodos abstractos que las clases deben implementar. Desde Java 8 admite métodos default y static.
Uso principal	Cuando varias clases comparten atributos y comportamiento común, pero necesitan implementar algunos métodos de manera diferente.	Cuando varias clases (incluso no relacionadas jerárquicamente) deben compartir un mismo comportamiento sin necesidad de heredar.
Herencia	Una clase solo puede heredar de una clase abstracta (herencia simple).	Una clase puede implementar múltiples interfaces (herencia múltiple de comportamiento).
Constructores	Pueden tener constructores para inicializar atributos comunes.	No pueden tener constructores.
Atributos	Pueden tener atributos de instancia y estáticos.	Solo pueden tener constantes (public static final).
Métodos	Pueden ser abstractos o concretos.	Antes de Java 8: solo métodos abstractos. Desde Java 8: permiten métodos default y static.
Ventajas	- Reutilización de código. - Consistencia en el diseño. - Flexibilidad (métodos abstractos + concretos). - Organización clara de jerarquías. - Facilita el mantenimiento.	- Soporta herencia múltiple. - Mayor abstracción y desacoplamiento. - Favorece el polimorfismo. - Reutilización en jerarquías distintas. - Métodos default y static reducen duplicación.

Desventajas	- Solo se puede heredar de una clase abstracta. - Puede generar rigidez en el diseño. - Riesgo de heredar métodos innecesarios. - No compatible con clases no relacionadas. - Complejo para principiantes.	- No permiten atributos de instancia. - Exceso de interfaces complica el diseño. - Implementación obligatoria de todos los métodos. - Riesgo de abuso por fragmentación. - En versiones antiguas, cambios forzaban modificar todas las clases.

RESUMEN DEL CAPITULO

Las clases abstractas permiten establecer plantillas comunes y forzar la implementación de ciertos métodos en las subclases. Las interfaces, por su parte, definen contratos que las clases pueden adoptar sin necesidad de compartir una jerarquía común. Ambos conceptos son pilares del diseño orientado a objetos moderno, promoviendo el uso de la abstracción y el polimorfismo. En este capítulo ha explorado el uso de clases abstractas e interfaces como recursos para diseñar sistemas flexibles y bien organizados. Gracias a estas herramientas, el desarrollador puede estructurar jerarquías claras, aplicar el principio de programación orientada a contratos y fomentar el uso efectivo del polimorfismo (Salinas, 2021).

AUTOEVALUACIÓN

En todas las preguntas de selecciona múltiple un literal es la respuesta correcta.

1. ¿Cuál de las siguientes afirmaciones es verdadera sobre las clases abstractas?
   a) No pueden contener métodos concretos
   b) Solo pueden tener métodos estáticos
   c) No pueden tener constructores
   d) Pueden tener métodos abstractos y concretos

2. ¿Cuál es el objetivo principal de una interfaz?
   a) Heredar comportamiento
   b) Establecer un contrato
   c) Definir atributos
   d) Crear objetos directamente

3. ¿Qué tipo de herencia permite una interfaz?
   a) Única
   b) Jerárquica
   c) Múltiple
   d) Lineal

4. ¿Cuál es la diferencia clave entre una clase abstracta y una interfaz?
   a) Las clases abstractas no pueden tener métodos
   b) Las interfaces tienen atributos privados
   c) Las clases abstractas permiten estado y las interfaces no
   d) Las interfaces no se usan en Java

5. ¿Qué palabra clave se usa para implementar una interfaz?
   a) extends
   b) interface
   c) inherits
   d) implements

6. ¿Qué método debe implementar una clase que hereda de una clase abstracta?
   a) Solo los métodos concretos

b) Solo los métodos de instancia
c) Todos los métodos abstractos
d) Ningún método

7. ¿Puede una interfaz tener atributos?
   a) Sí, de cualquier tipo
   b) No
   c) Sí, pero deben ser constantes (public static final)
   d) Sí, si son privados

8. ¿Qué ventaja tiene el uso de interfaces?
   a) Mayor acoplamiento
   b) Permite herencia múltiple
   c) Menor extensibilidad
   d) Dificulta la prueba del código

9. ¿Cuál de las siguientes estructuras no se puede instanciar?
   a) Clase normal
   b) Clase abstracta
   c) Clase interna
   d) Clase estática

10. ¿Qué versión de Java introdujo métodos default en interfaces?
    a) Java 6
    b) Java 7
    c) Java 8
    d) Java 11

EJERCICIO PROPUESTO

Caso de estudio congresistas

Declara una clase Procurador que herede de la clase Persona (IdCodigo, Cedula, Nombre, Apellido, FechaNacimiento, Estado), con un atributo provincia que representa, partido político, declarar un método ProyectoPresentado.
Crear dos clases concretas que hereden de Procurador: la clase Congresista y la clase Senador con sus respectivos métodos.
En el congreso se realiza el pago de sueldo de la siguiente manera:

- Congresista, que reciben un salario mensual, descontando 9.32 % del seguro social, adicional tiene una bonificación de $ 1000 por cada proyecto presentado (puede ser aprobado, revisión), cuando sea aprobado el proyecto, si no está aprobado no recibe dicha bonificación, puede actualizarse de revisión a aprobado.
- Senador, que perciben un sueldo mensual, más comisiones, descontando 9.32 % del seguro social, puede recibir un anticipo del 50% máximo de su sueldo mensual.

Si considera más atributos en las clases queda a responsabilidad del programador.

- Crear el modelado de datos UML del problema planteado, generar el código en java.
- Implementar en java las respectivas clases con sus atributos y métodos del problema que permita obtener el sueldo a cobrar según su tipo de legislador.

REFERENCIA BIBLIOGRÁFICA

Eckel, B. (2021). Pensando en Java. Prentice Hall.

Gómez, E., & Pinto, L. (2023). Interfaces y programación basada en contratos. Revista Colombiana de Informática., 18(2), 77–84.

Guzmán, H. (2021). UML y Java: Aplicación práctica para el diseño de software. Ra-Ma.

Herrera, H. (2024). Abstracción y polimorfismo en entornos educativos. Educación y Tecnología., 5(3), 90-102.

Jiménez, R. (2024). Clases abstractas y sus aplicaciones en arquitectura de software. Innovación y Tecnología, 9(3), 66-74.

López, C. (2025). Abstracción como principio del diseño orientado a objetos. Revista Latinoamericana de Computación., 15(1), 9-16.

Morán, D. (2022). Introducción al desarrollo de software en Java. (Alfaomega.).

Rodríguez, J., & Cevallos, F. (s. f.). Patrones de diseño orientados a interfaces. Journal de Ingeniería y Tecnología., 12-20.

Roldán, M., & Meneses, R. (2023). Comparación entre clases abstractas e interfaces. Revista Iberoamericana de Tecnologías Educativas, 55-62.

Salinas, R. (2021). Fundamentos de programación con Java. Ediciones Díaz de Santos.

Sierra, K., & Bates, B. (2022). Programación en Java (3.ª edición.). O'Reilly.

Vargas, A., & Peña, C. (2023). Diseño limpio y modularidad con interfaces. Boletín Latinoamericano de Software., 8(4), 44-53.

Vargas, M. (2022). Diseño orientado a objetos: Buenas prácticas en Java. Revista de Ingeniería de Software., 6(1), 34-40.

Joffre Jeorwin Cartuche Calva https://orcid.org/0000-0002-1633-2291
Ecuatoriano, Analista e Ingeniero en Sistemas Informáticos por la ESPOCH, con una Maestría en Ingeniería de Software por la ESPE. Candidato a Doctor en Dirección de Proyectos por la Universidad de Investigación e Innovación de México. Docente titular agregado en la Universidad Técnica de Machala, donde imparte asignaturas de pregrado como Programación Orientada a Objetos, Programación Avanzada, Programación Móvil con Despliegues en la Nube, y en posgrado Gestión de Proyectos de Software. Cuenta con experiencia en desarrollo de software, soluciones móviles e integradas, y gestión tecnológica. Sus intereses incluyen arquitectura de software, metodologías ágiles, inteligencia artificial aplicada, microservicios, contenedores, computación en la nube (AWS y Azure) y despliegue de datos. Su enfoque está orientado a la innovación tecnológica, el desarrollo de software escalable y la gestión eficiente de proyectos de TI.

Bertha Eugenia Mazón Olivo https://orcid.org/0000-0002-2749-8561
Es docente investigadora titular en la Universidad Técnica de Machala (UTMACH), Ecuador. Doctora en Tecnologías de la Información y las Comunicaciones por la Universidad de La Coruña, España. Es Magíster en Informática Aplicada e Ingeniera en Sistemas por la Escuela Superior Politécnica de Chimborazo (ESPOCH) y posee un Diplomado Internacional en Ingeniería de Datos por la Universidad Nacional de Ucayali, Perú. Actualmente cursa una Maestría en Big Data y Data Science en la Universidad Internacional de Valencia. Participa como docente de posgrado en UTMACH, Universidad de los Hemisferios y ESPOCH. Forma parte del grupo de investigación AutoMathTIC y dirige el proyecto sobre adopción de IA e IoT en el sector agropecuario de El Oro. Contribuyó al diseño de la carrera de Ciencia de Datos en UTMACH. Sus intereses incluyen IoT, Ciencia de Datos, Big Data, Machine Learning y Deep Learning. Ha publicado diversos artículos científicos, libros y capítulos.

Wilmer Braulio Rivas Asanza https://orcid.org/0000-0002-2239-3664
Es Ingeniero en Informática por la Universidad Católica de Cuenca, con una formación académica que incluye un Diplomado en Auditoría de TI por la Escuela Superior Politécnica del Litoral (ESPOL), un Diplomado en Inteligencia Artificial, una Maestría en Docencia y Gestión en la Educación Superior por la Universidad Estatal de Guayaquil y una Maestría en Gestión Estratégica de las Tecnologías de la Información por la Universidad Estatal de Cuenca. Además, es Doctor en Tecnologías de la Información y las Comunicaciones por la Universidad de La Coruña, España. Cuenta con experiencia profesional tanto en el sector público como en el privado. Es profesor titular en la Universidad Técnica de Machala durante aproximadamente 14 años, impartiendo asignaturas como Sistemas Operativos, Gestión de Centros de Cómputo e Inteligencia Artificial. Actualmente, cuenta con varias publicaciones y libros indexados que reflejan su trayectoria académica y profesional.

ISBN: 978-9942-53-026-4

9 789942 530264

Compás
capacitación e investigación