Herencia, Polimorfismo y Abstracción en Java

Domina los pilares fundamentales de la programación orientada a objetos en Java con ejemplos prácticos y detallados para estudiantes universitarios y desarrolladores junior.

Conceptos Clave que Aprenderemos

01

Herencia en Java

Comprender y aplicar la herencia como mecanismo de reutilización de código y extensibilidad en la programación orientada a objetos.

02

Modificadores de Acceso

Implementar y gestionar correctamente los modificadores de acceso (public, private, protected) en contextos de herencia.

03

Conversión de Tipos

Utilizar constructores y técnicas de conversión como upcasting y downcasting de manera segura y eficiente.

04

Abstracción Avanzada

Aplicar clases abstractas y métodos abstractos para diseñar jerarquías flexibles y mantenibles.

05

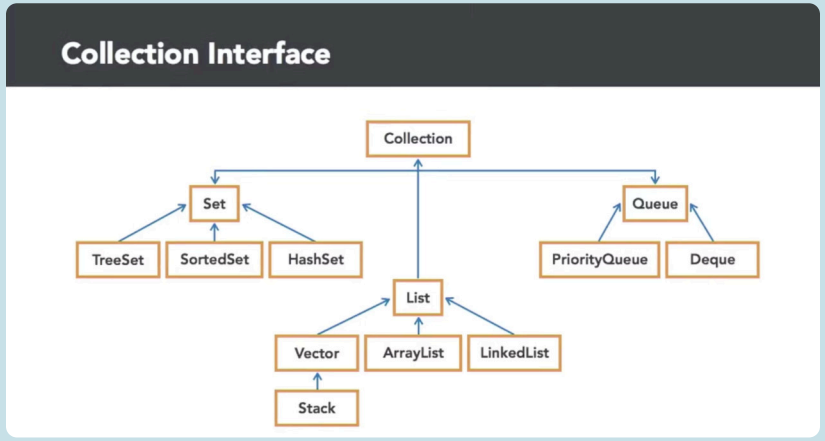
Polimorfismo Práctico

Desarrollar soluciones robustas aplicando polimorfismo y sobrescritura de métodos en proyectos reales.

¿Qué es la Herencia?

La herencia es un mecanismo fundamental en la POO que permite crear nuevas clases (subclases o clases derivadas) a partir de clases existentes (superclases o clases base). La subclase hereda los atributos y métodos de la superclase, promoviendo la **reutilización de código** y la **extensibilidad**.

Este concepto permite establecer relaciones jerárquicas entre clases, donde las subclases especializan o extienden el comportamiento de sus superclases, manteniendo la coherencia del diseño orientado a objetos.



Beneficios Fundamentales de la Herencia

Reutilización de Código

Evita la duplicación de código al heredar atributos y métodos de la superclase, reduciendo el tiempo de desarrollo y mantenimiento.

Extensibilidad

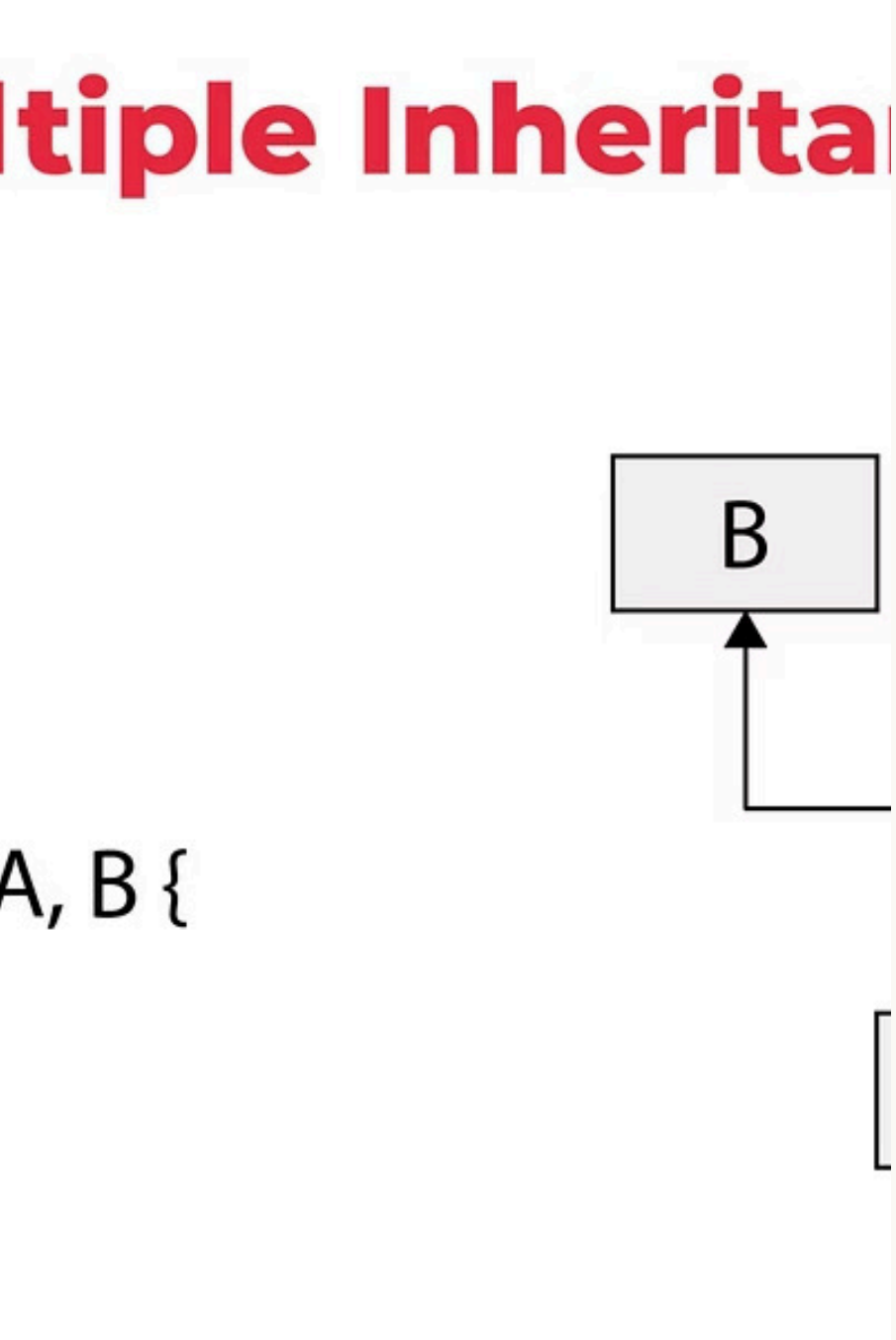
Facilita la adición de nuevas funcionalidades creando subclases que extienden el comportamiento de la superclase sin modificar el código existente.

Mantenibilidad

Simplifica el mantenimiento del código al permitir modificaciones centralizadas en la superclase que se propagan automáticamente.

Organización

Promueve una estructura jerárquica del código que facilita su comprensión, gestión y documentación para equipos de desarrollo.



Tipos de Herencia: Simple vs Múltiple

Herencia Simple

Una clase solo puede heredar de una superclase. **Java solo permite herencia simple en clases**, garantizando simplicidad y evitando conflictos de herencia múltiple como el "problema del diamante".

Herencia Múltiple

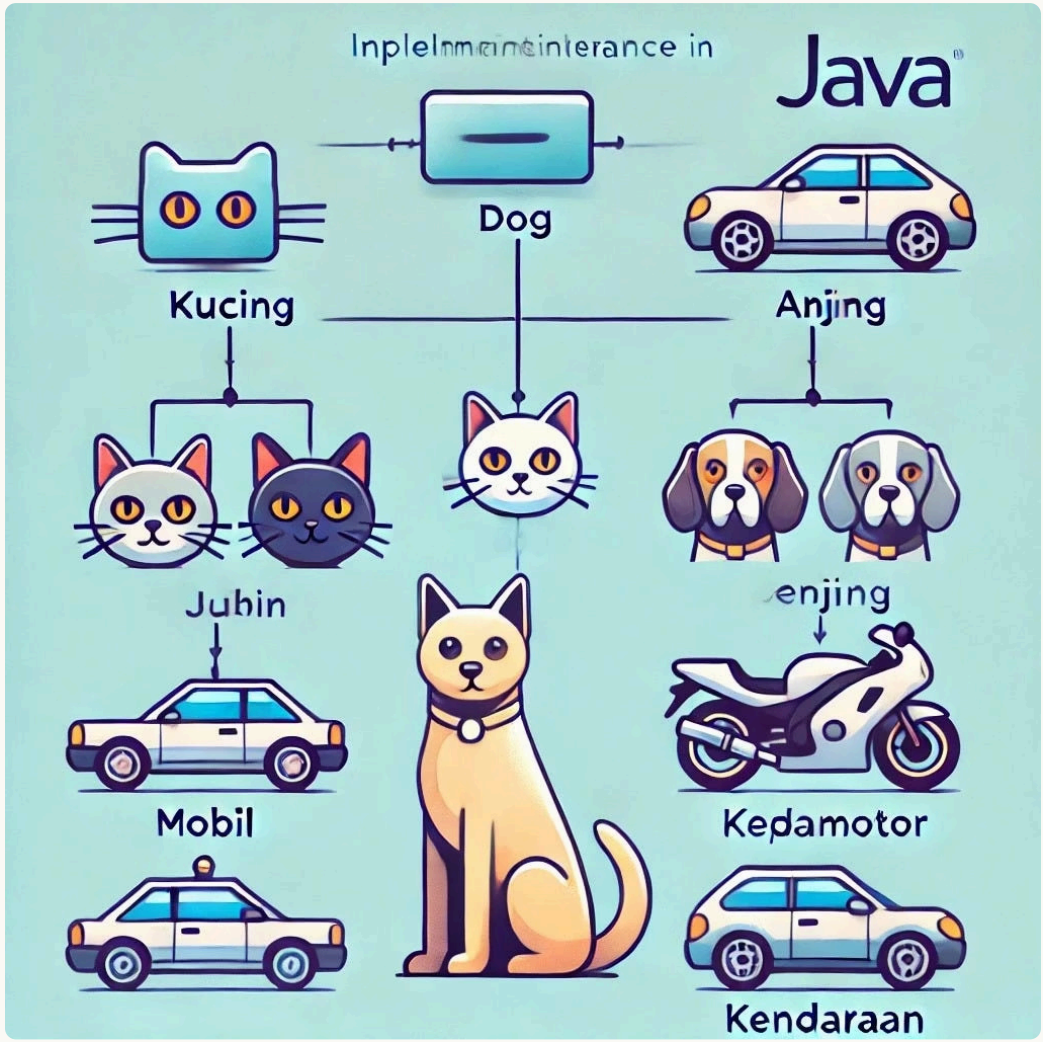
Una clase puede heredar de múltiples superclases. **Java no permite herencia múltiple en clases, pero sí en interfaces**, ofreciendo flexibilidad controlada mediante contratos bien definidos.

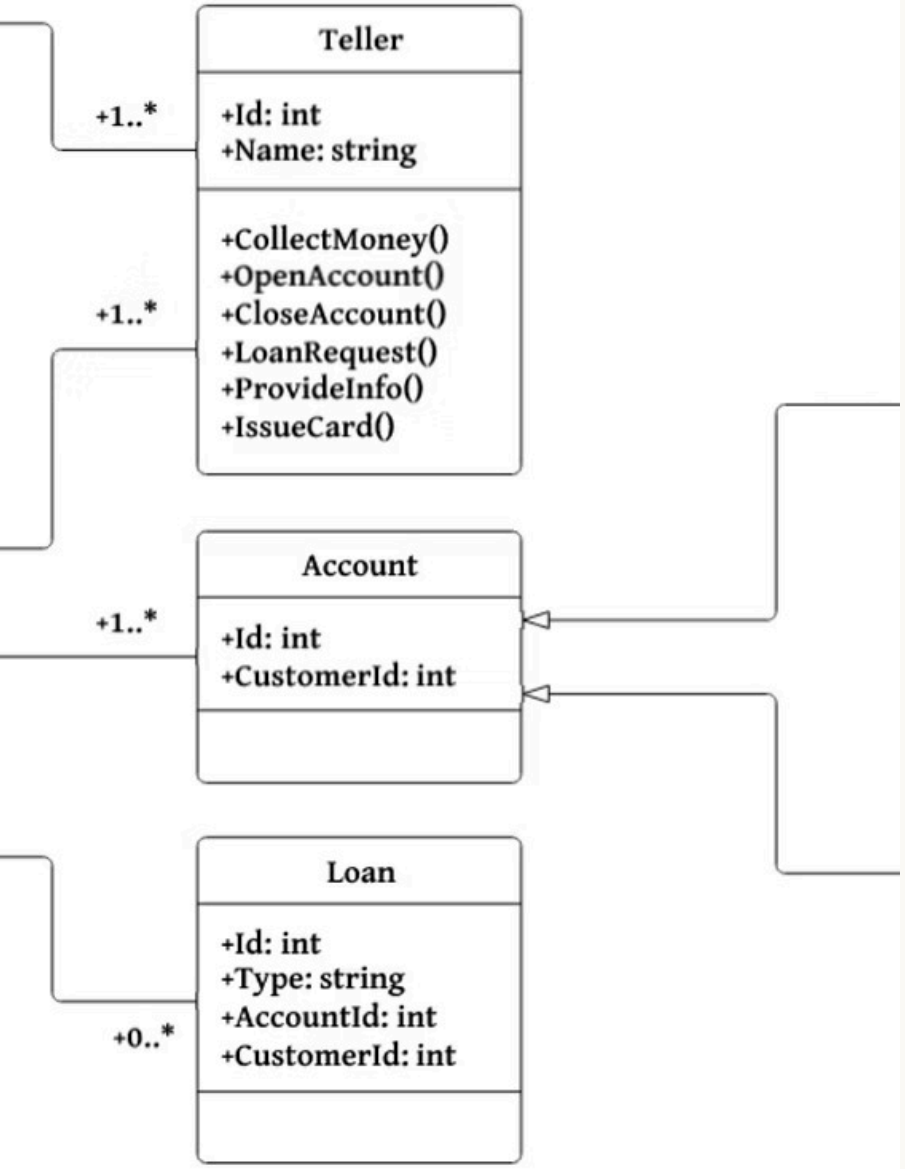
El Principio "is-a" en Herencia

La herencia se basa en el principio **"is-a" (es un)**. Una subclase es un tipo específico de superclase.

Este principio fundamental garantiza que la relación de herencia sea lógicamente consistente y semánticamente correcta.

- Un Perro **es un** Animal
- Un Coche **es un** Vehículo
- Un Estudiante **es una** Persona





Modelado con Diagramas UML

En un diagrama de clases UML, la herencia se representa con una **flecha con la punta vacía** que apunta desde la subclase hacia la superclase.

Esta representación visual ayuda a los desarrolladores a comprender rápidamente las relaciones jerárquicas entre clases, facilitando el diseño y la comunicación del equipo de desarrollo sobre la arquitectura del sistema.

Ejemplo Básico de Herencia en Java

```
// Superclase
public class Animal {
    protected String nombre;

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

    public void hacerSonido() {
        System.out.println("Sonido genérico de animal");
    }
}

// Subclase
public class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre); // Llama al constructor de la superclase
    }

    @Override
    public void hacerSonido() {
        System.out.println("¡Guau!");
    }
}
```

La palabra clave **extends** establece la relación de herencia, mientras que **super()** invoca el constructor de la superclase.

Modificadores de Acceso en Java

public

Accesible desde cualquier parte del programa, proporcionando visibilidad completa.

private

Accesible solo desde dentro de la misma clase, garantizando encapsulación estricta.

protected

Accesible desde la misma clase, subclases y clases del mismo paquete.

Sin modificador

Accesible solo desde las clases del mismo paquete (package-private).

Modificadores de Acceso en Herencia

```
public class Animal {
    public String nombrePublico;
    private String nombrePrivado;
    protected String nombreProtegido;
    String nombrePaquete; // Sin modificador

    public Animal(String nombre) {
        this.nombrePublico = nombre;
        this.nombrePrivado = nombre;
        this.nombreProtegido = nombre;
        this.nombrePaquete = nombre;
    }

    public String getNombrePrivado() {
        return nombrePrivado;
    }
}
```

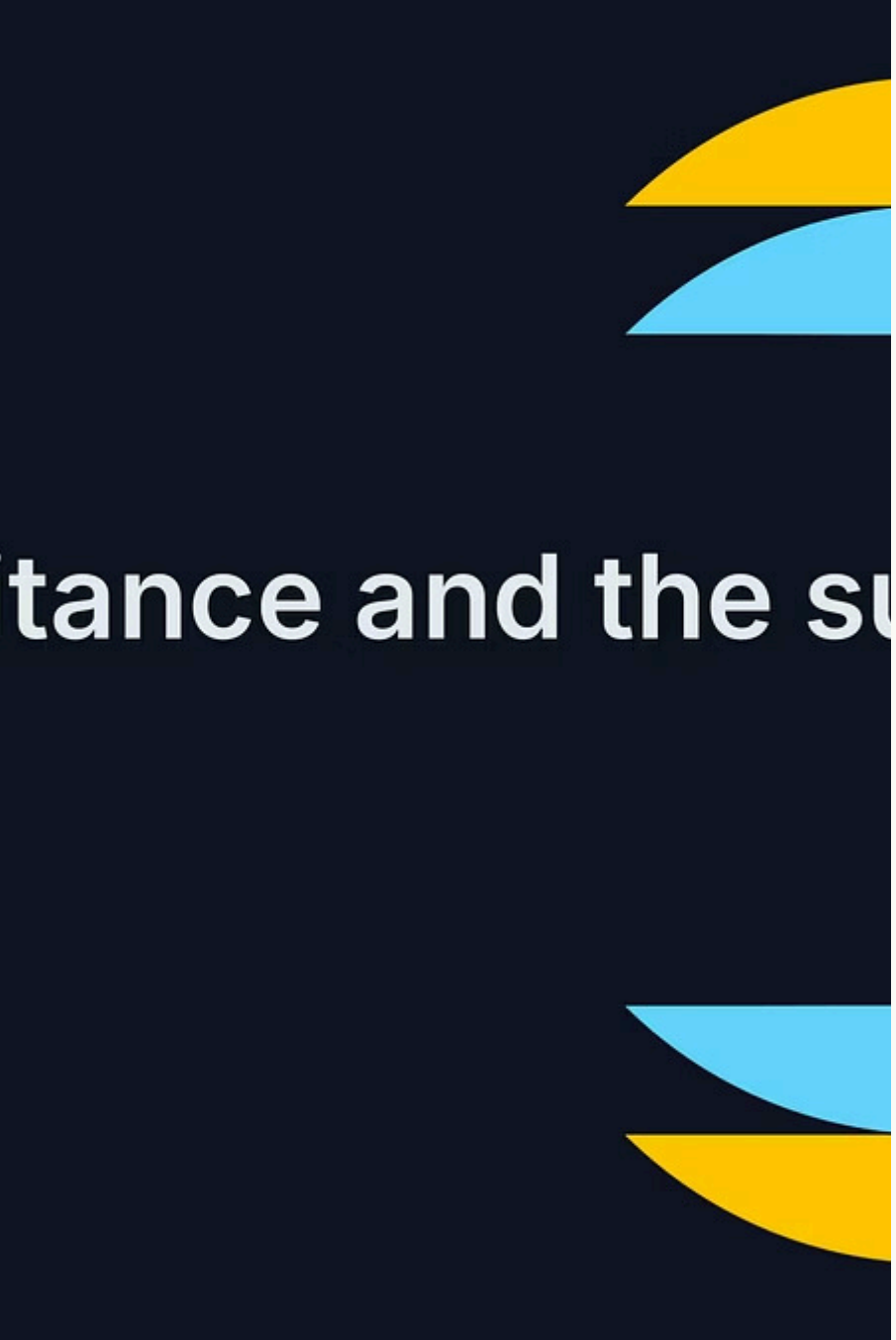
Los modificadores de acceso en herencia determinan qué miembros de la superclase son accesibles desde las subclases, controlando la visibilidad y el encapsulamiento.

Encapsulamiento y Seguridad

Utilizar **private** y **protected** correctamente es fundamental para garantizar el encapsulamiento y la seguridad de los datos en jerarquías de herencia.

- **Private:** Oculta implementación interna
- **Protected:** Permite acceso controlado a subclases
- **Public:** Define la interfaz pública de la clase

Esta estrategia protege la integridad de los datos mientras proporciona flexibilidad para la extensión.



Constructores en Herencia

Cuando se crea una instancia de una subclase, **primero se llama al constructor de la superclase** para inicializar los atributos heredados. Esto se realiza utilizando la palabra clave **super()**.

Este mecanismo garantiza que la inicialización siga el orden jerárquico correcto, asegurando que todos los componentes heredados estén correctamente configurados antes de inicializar los componentes específicos de la subclase.

Ejemplo de Constructores en Herencia

```
public class Animal {
    protected String nombre;
    protected int edad;

    public Animal(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        System.out.println("Constructor de Animal");
    }
}

public class Perro extends Animal {
    private String raza;

    public Perro(String nombre, int edad, String raza) {
        super(nombre, edad); // Llama al constructor de Animal
        this.raza = raza;
        System.out.println("Constructor de Perro");
    }
}
```

La llamada a **super()** debe ser la primera línea del constructor de la subclase.

Upcasting: Conversión Ascendente

1

Subclase

Objeto específico con características detalladas

2

Superclase

Referencia genérica que permite tratamiento uniforme

El **upcasting** es la conversión automática y segura de un objeto de una subclase a un tipo de superclase. Esta conversión permite generalizar instancias y tratarlas de manera uniforme, ya que un objeto de la subclase siempre es un objeto de la superclase.

```
Animal animal = new Perro("Fido", 3, "Labrador"); // Upcasting automático
```

Downcasting y el Operador instanceof

El **downcasting** es la conversión de un objeto de una superclase a un tipo de subclase. Esta conversión debe realizarse con precaución y verificación previa.

1

Verificación con instanceof

Comprobar el tipo real del objeto antes de realizar el casting

2

Casting Explícito

Realizar la conversión de tipo de manera explícita

3

Manejo de Excepciones

Estar preparado para manejar ClassCastException si es necesario

Ejemplo de Downcasting Seguro

```
public class Main {
    public static void main(String[] args) {
        Animal animal = new Perro("Fido", 3, "Labrador");

        if (animal instanceof Perro) {
            Perro perro = (Perro) animal; // Downcasting seguro
            System.out.println("El nombre del perro es: " + perro.nombre);
        } else {
            System.out.println("El animal no es un perro.");
        }
    }
}
```

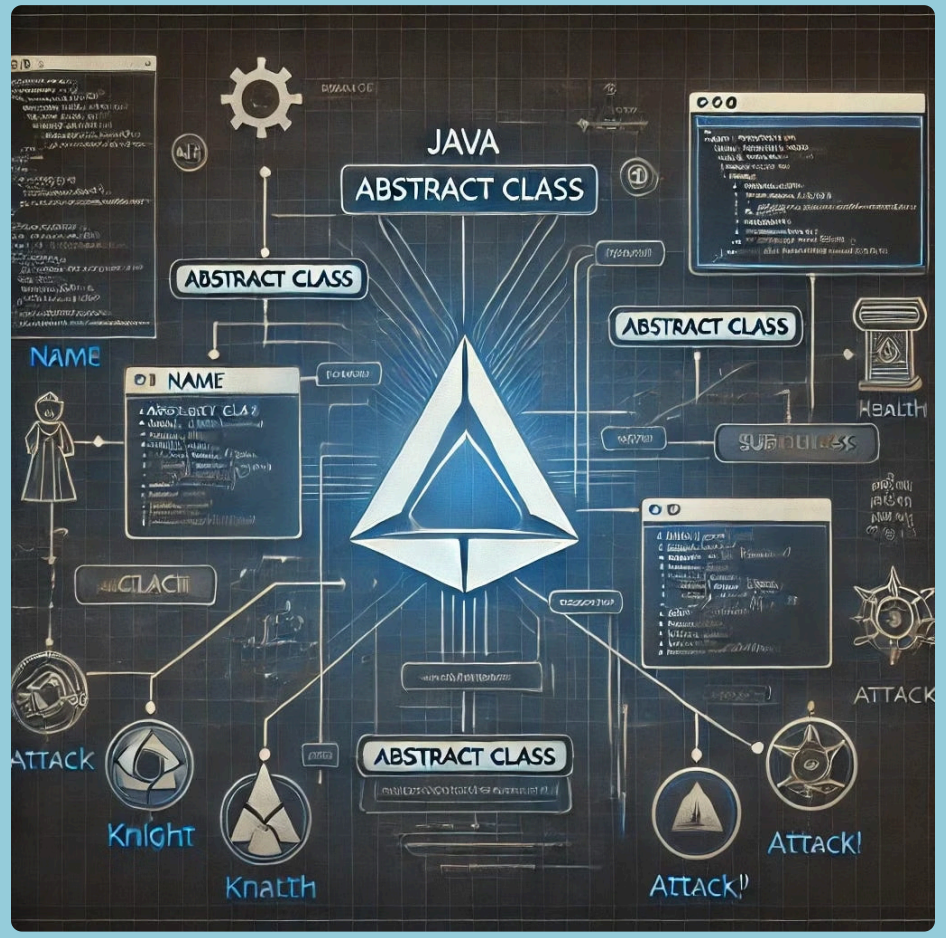
El operador **instanceof** es esencial para realizar downcasting seguro, evitando excepciones en tiempo de ejecución y garantizando la robustez del código.

Introducción a las Clases Abstractas

Una **clase abstracta** es una clase que no se puede instanciar directamente. Sirve como plantilla para crear otras clases (subclases).

Puede contener:

- **Métodos abstractos:** Sin implementación
- **Métodos concretos:** Con implementación
- **Atributos:** Variables de instancia
- **Constructores:** Para inicialización



Métodos Abstractos y Ejemplo Práctico

```
public abstract class Figura {
    protected String color;

    public Figura(String color) {
        this.color = color;
    }

    // Método abstracto - debe implementarse en subclases
    public abstract double calcularArea();

    // Método concreto - implementación compartida
    public void imprimirColor() {
        System.out.println("Color: " + color);
    }
}
```

Los **métodos abstractos** definen contratos que las subclases deben cumplir, while los métodos concretos proporcionan funcionalidad compartida. Esta combinación ofrece flexibilidad y consistencia.

Beneficios de la Abstracción

Flexibilidad

Permite definir estructuras base reutilizables que se pueden adaptar a diferentes situaciones específicas manteniendo coherencia arquitectural.

Extensibilidad

Facilita la adición de nuevas clases que extiendan el comportamiento de la estructura base sin modificar código existente.

Mantenibilidad

Simplifica el mantenimiento del código al centralizar funcionalidad común y permitir modificaciones controladas en la jerarquía.

¿Qué es el Polimorfismo?

El **polimorfismo** es la capacidad de tratar objetos de diferentes clases de manera uniforme. Permite que un objeto tome muchas formas diferentes, habilitando flexibilidad y extensibilidad en el diseño de software.

Esta característica fundamental de la POO permite escribir código más genérico y reutilizable, donde una misma interfaz puede comportarse de manera diferente según el tipo específico del objeto que la implemente.

Tipos de Polimorfismo en Java

1

Sobrecarga (Overloading)

Definir múltiples métodos con el mismo nombre pero con diferentes listas de parámetros en la misma clase. Resolución en tiempo de compilación.

2

Coerción (Coercion)

Conversión implícita de un tipo de dato a otro realizada automáticamente por el compilador para mantener compatibilidad de tipos.

3

Por Herencia (Inheritance)

Utilizar herencia y sobrescritura de métodos para que objetos de diferentes clases respondan de manera específica a la misma llamada de método.

Sobrescritura de Métodos (@Override)

```
public class Animal {
 public void hacerSonido() {
 System.out.println("Sonido genérico de animal");
 }
}

public class Perro extends Animal {
 @Override // Anotación para indicar sobrescritura
 public void hacerSonido() {
 System.out.println("¡Guau!");
 }
}

public class Main {
 public static void main(String[] args) {
 Animal animal1 = new Animal();
 Animal animal2 = new Perro(); // Polimorfismo

 animal1.hacerSonido(); // "Sonido genérico de animal"
 animal2.hacerSonido(); // "¡Guau!" - versión sobrescrita
 }
}
```

Ejemplo Práctico: Sistema de Pagos

```
// Clase base para métodos de pago
class MetodoPago {
    public void realizarPago(double cantidad) {
        System.out.println("Realizando pago genérico de " + cantidad);
    }
}

class TarjetaCredito extends MetodoPago {
    private String numeroTarjeta;

    public TarjetaCredito(String numeroTarjeta) {
        this.numeroTarjeta = numeroTarjeta;
    }

    @Override
    public void realizarPago(double cantidad) {
        System.out.println("Pagando " + cantidad + " con tarjeta " + numeroTarjeta);
    }
}
```

Este ejemplo demuestra cómo el polimorfismo permite intercambiar implementaciones sin modificar el código cliente.

Sistema de Notificaciones Polimórfico

```
class ServicioNotificacion {
    public void enviarNotificacion(String mensaje, String destinatario) {
        System.out.println("Enviando notificación genérica a " + destinatario);
    }
}

class EmailService extends ServicioNotificacion {
    @Override
    public void enviarNotificacion(String mensaje, String destinatario) {
        System.out.println("Enviando email a " + destinatario + ": " + mensaje);
    }
}

class SMSService extends ServicioNotificacion {
    @Override
    public void enviarNotificacion(String mensaje, String destinatario) {
        System.out.println("Enviando SMS a " + destinatario + ": " + mensaje);
    }
}
```

El polimorfismo permite cambiar el comportamiento dinámicamente en tiempo de ejecución.

Mejores Prácticas y Conclusiones

Diseño Coherente

Aplica el principio "is-a" correctamente y mantén jerarquías lógicas que reflejen relaciones reales del dominio del problema.

Encapsulamiento

Utiliza modificadores de acceso apropiados para proteger la integridad de los datos mientras permites extensibilidad controlada.

Polimorfismo Efectivo

Aprovecha la sobrescritura de métodos y las clases abstractas para crear código flexible, mantenible y extensible.

¡La práctica constante es clave para dominar estos conceptos fundamentales de Java!