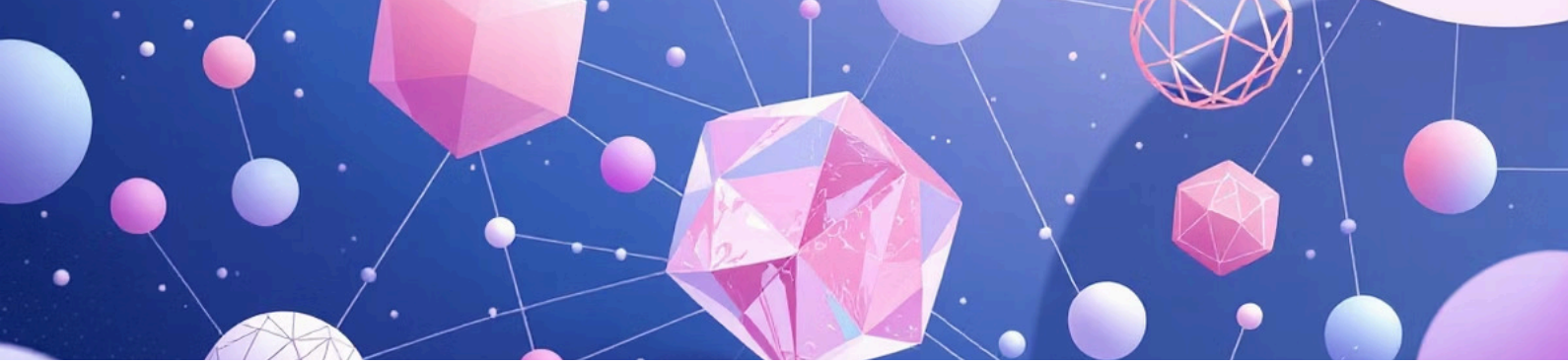
PROMETEO

Unidad 5: Programación Orientada a Objetos

Programación

Técnico Superior de DAM / DAW



La Programación Orientada a Objetos (POO) es uno de los paradigmas más influyentes en el desarrollo de software moderno. Frente a los modelos procedimentales tradicionales —que estructuraban el código en funciones y datos separados—, la POO propone un enfoque más natural: modelar el software como una colección de objetos que representan entidades del mundo real, con su propio estado y comportamiento.

En la vida real, interactuamos constantemente con objetos: un coche tiene un color, un modelo, una velocidad, y puede acelerar o frenar. De forma análoga, en programación, un objeto tiene atributos (datos) y métodos (acciones). Este enfoque hace que el código sea más modular, reutilizable y mantenible.

En Java —uno de los lenguajes más representativos de la POO— los objetos se crean a partir de clases, que funcionan como plantillas o moldes. Una clase define qué atributos y métodos tendrán los objetos que se creen a partir de ella. Por ejemplo:

```
class Libro {
    String titulo;
    String autor;
    int paginas;
    boolean disponible;

    // Constructor
    Libro(String titulo, String autor, int paginas) {
        this.titulo = titulo;
        this.autor = autor;
        this.paginas = paginas;
        this.disponible = true;
    }

    // Método
    void prestar() {
        if (disponible) {
            disponible = false;
            System.out.println("Libro prestado: " + titulo);
        } else {
            System.out.println("El libro ya está prestado.");}}}
```

Aquí, `Libro` es la clase; y cada vez que usamos `new Libro("1984", "Orwell", 328)`, creamos un objeto específico con su propio estado.

Los tres pilares de la POO aplicados a esta sesión

Encapsulación: los atributos suelen declararse `private` para proteger el estado interno. El acceso controlado se realiza mediante métodos getters y setters.

Abstracción: la clase simplifica la complejidad del mundo real y ofrece solo los datos y comportamientos relevantes.

Modularidad: cada clase representa una pieza independiente del sistema, lo que facilita su mantenimiento y ampliación.

El papel de los constructores

Un constructor es un método especial que se ejecuta automáticamente al crear un objeto. Su función es inicializar el estado del objeto y garantizar que comienza en condiciones válidas. En Java, lleva el mismo nombre que la clase y no tiene tipo de retorno. Por ejemplo:

```
Libro miLibro = new Libro("Cien años de soledad", "García Márquez", 417);
```

Este código crea un objeto `miLibro` con atributos inicializados. Si se necesita personalizar la creación (por ejemplo, con o sin disponibilidad), se pueden definir múltiples constructores (sobrecarga).

Métodos: el comportamiento de los objetos

Los métodos definen qué puede hacer un objeto. Pueden ser:

- **Públicos (`public`):** accesibles desde fuera de la clase.
- **Privados (`private`):** solo accesibles dentro de la clase.
- **Estáticos (`static`):** pertenecen a la clase y no a instancias concretas.

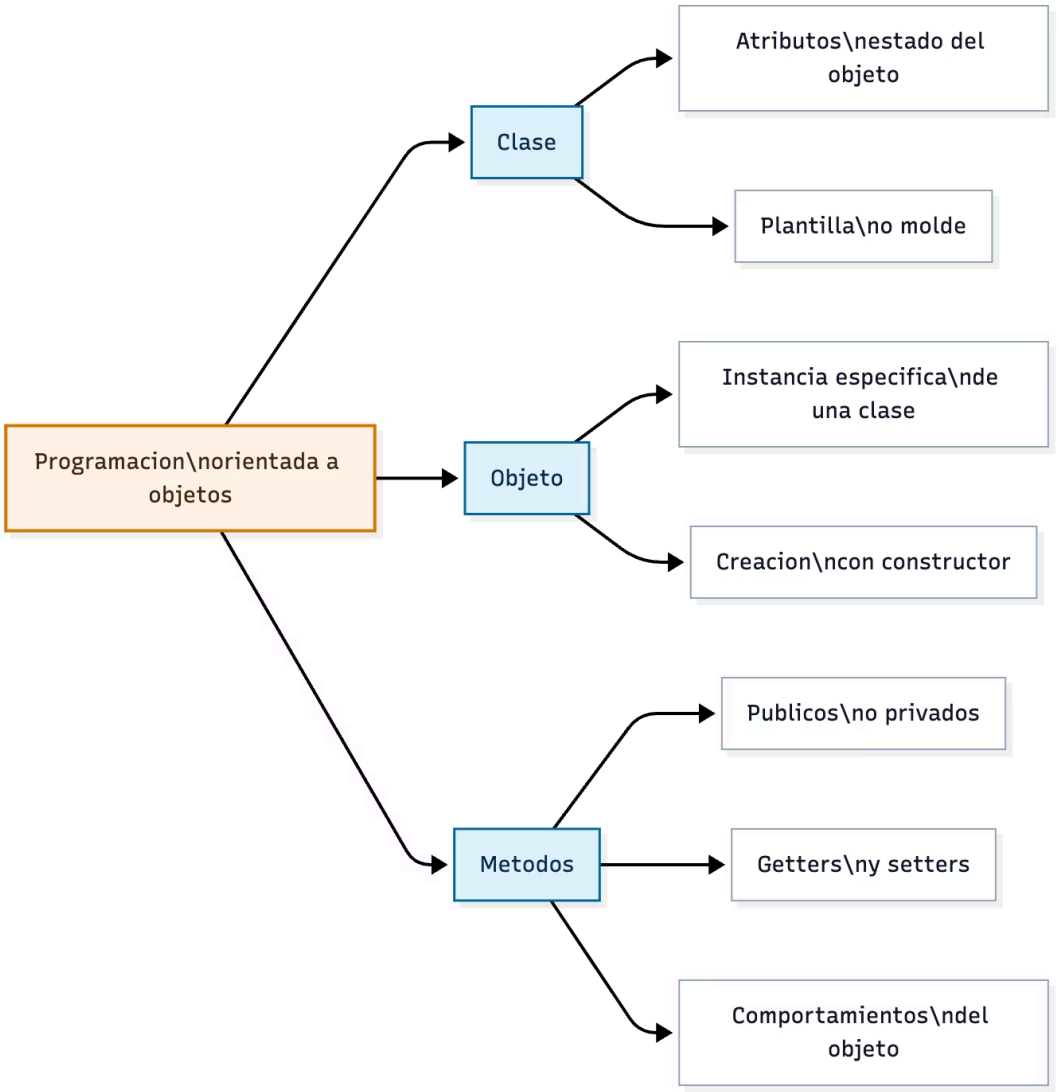
Los métodos permiten a los objetos interactuar y modificar su estado interno. Por ejemplo, un método `devolver()` podría cambiar `disponible` a `true`.

01	02	03	04
Clase = plantilla	Objeto = instancia con datos concretos	Constructor = inicializador del objeto	Método = comportamiento que define sus acciones

Este modelo mental permite construir sistemas más ordenados, escalables y fáciles de comprender.

Esquema visual

Diagrama conceptual de la Programación Orientada a Objetos



Descripción del esquema:

En la parte superior, el nodo central representa la POO como paradigma general.

De él derivan tres ramas:

Clase: simboliza la definición del modelo. Contiene atributos (estado) y actúa como plantilla.

Objeto: representa una instancia concreta creada a partir de la clase mediante un constructor.

Métodos: representan el comportamiento o conjunto de acciones que el objeto puede realizar.

Los nodos secundarios muestran detalles importantes como visibilidad (público/privado) y encapsulación (getters/setters). Este esquema ayuda a visualizar cómo las tres piezas encajan y colaboran para crear software modular y reutilizable.

Caso de estudio – Sistema de usuarios de Netflix

Contexto

Netflix opera a escala global con más de 230 millones de suscriptores. Gestionar esa cantidad de usuarios implica un sistema altamente modular, mantenible y escalable. Cada usuario, desde el punto de vista del software, es un objeto con su propio estado y comportamiento.

Estrategia

Netflix utiliza un modelo basado en clases y objetos para representar entidades como Usuario, Contenido, Suscripción o HistorialVisualización. Ejemplo simplificado en pseudocódigo Java:

```
class Usuario {
    String nombre;
    String email;
    String tipoSuscripcion;
    List historial;

    Usuario(String nombre, String email, String tipoSuscripcion) {
        this.nombre = nombre;
        this.email = email;
        this.tipoSuscripcion = tipoSuscripcion;
        this.historial = new ArrayList<>(); }

    void reproducirContenido(String titulo) {
        historial.add(titulo);
        System.out.println(nombre + " está viendo: " + titulo);}

    void calcularRecomendaciones() {
        // Simula recomendaciones según historial
        System.out.println("Recomendaciones para " + nombre +
            ": contenido similar a " +
            historial.get(historial.size() - 1));}}
```

Cada vez que un nuevo usuario se registra, el sistema ejecuta un constructor que inicializa sus datos. Posteriormente, los métodos como `reproducirContenido()` o `calcularRecomendaciones()` modifican o consultan el estado individual del objeto.

Resultado

Gracias a este enfoque:

- Cada usuario tiene un estado independiente, lo que permite una experiencia personalizada.
- El sistema puede manejar millones de objetos Usuario simultáneamente sin interferencias entre ellos.
- La lógica del código es reutilizable y escalable: se pueden añadir nuevos tipos de usuarios o funcionalidades (por ejemplo, perfiles familiares) sin reescribir el sistema completo.

Netflix demuestra cómo la POO permite traducir el mundo real a código estructurado, optimizando la personalización y la eficiencia operativa a gran escala.

Herramientas y consejos básicos

Usa convenciones de nombres correctas:

Clases → PascalCase: `CuentaBancaria`, `EmpleadoEmpresa`.

Métodos → camelCase: `calcularInteres()`, `obtenerSaldo()`.

Esto mejora la legibilidad y mantiene coherencia con los estándares de Java.

Sobrescribe toString() para depurar fácilmente:

```
@Override
public String toString() {
   return "Libro{titulo='" + titulo +
         "', autor='" + autor + "'}";}
```

Esto facilita ver el contenido de un objeto sin necesidad de inspeccionarlo atributo por atributo.

Aplica encapsulación estricta:

Declara los atributos como `private` y usa getters/setters para controlarlos:

```
private int edad;

public void setEdad(int edad) {
   if (edad < 0)
      throw new IllegalArgumentException("Edad no puede ser negativa");
   this.edad = edad;}
```

Así evitas estados inválidos dentro del objeto.

Utiliza herramientas profesionales:

- **IntelliJ IDEA o Eclipse** para desarrollo orientado a objetos con autocompletado y visualización UML.
- **PlantUML o StarUML** para generar diagramas de clases.
- **JUnit** para probar métodos y verificar que los constructores inicializan correctamente.

Planifica tus clases antes de codificar:

Un error común es programar sin diseño previo. Dibuja tus relaciones (qué clase usa a cuál, qué atributos comparten) para evitar duplicaciones o acoplamientos innecesarios.

Mitos y realidades

❌ Mito: "La POO es solo para aplicaciones grandes y complejas."

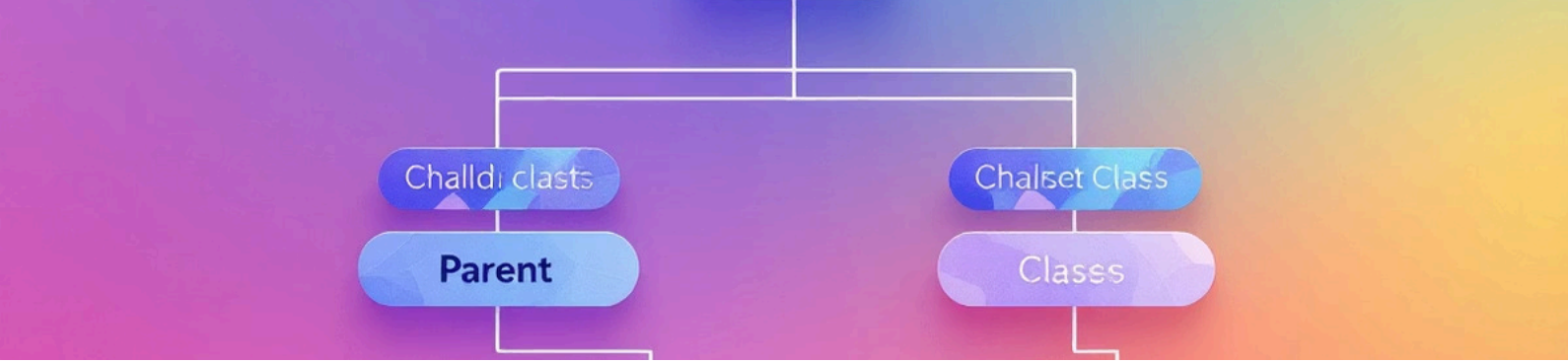
→ **FALSO.** Incluso en proyectos pequeños, estructurar el código en clases y objetos facilita la comprensión, la reutilización y la escalabilidad. Un programa de agenda personal, por ejemplo, puede tener clases simples como Contacto, Evento y Calendario.

❌ Mito: "Cuantas más clases, mejor diseño."

→ **FALSO.** Un buen diseño no depende de la cantidad, sino de la claridad y responsabilidad única de cada clase. Demasiadas clases pequeñas pueden fragmentar la lógica; muy pocas, concentrar demasiado poder en una sola. Regla práctica: cada clase debe tener una sola responsabilidad principal, siguiendo el principio SRP (Single Responsibility Principle).

Resumen final

- **POO:** paradigma basado en objetos con estado (atributos) y comportamiento (métodos).
- **Clase:** plantilla o modelo; **objeto:** instancia creada con `new`.
- **Constructor:** inicializa el estado del objeto.
- **Métodos:** definen el comportamiento; pueden ser públicos, privados o estáticos.
- **Encapsulación:** protege datos mediante getters/setters.
- **Caso Netflix:** millones de objetos Usuario independientes para personalización global.
- **Buenas prácticas:** nombres descriptivos, validaciones, `toString()` y diseño modular.



Sesión 12 – Herencia y Polimorfismo

La herencia y el polimorfismo son dos de los pilares más potentes de la Programación Orientada a Objetos (POO). Ambos conceptos permiten reutilizar código, extender comportamientos y crear estructuras jerárquicas más naturales que reflejan cómo clasificamos las cosas en el mundo real.

Herencia: Reutilizar y Especializar

La herencia permite crear nuevas clases basadas en otras ya existentes. La clase base o superclase define características comunes, mientras las subclases heredan estos elementos y pueden añadir o modificar comportamientos. En Java, se utiliza la palabra clave `extends`:

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

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

    void presentarse() {
        System.out.println("Hola, soy " + nombre)   }}

class Estudiante extends Persona {
    String carrera;

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

    @Override
    void presentarse() {
        System.out.println("Hola, soy " + nombre + " y estudio " + carrera);    }}
```

En este ejemplo, la clase `Estudiante` hereda atributos y métodos de `Persona`, pero sobrescribe (`@Override`) el método `presentarse()` para adaptarlo a su propio contexto. De este modo, no se repite código y cada clase conserva una responsabilidad clara.

Polimorfismo: Muchas Formas, Un Solo Comportamiento

El término polimorfismo proviene del griego *poli* (muchos) y *morfos* (formas). En POO significa que una misma acción puede tener diferentes comportamientos según el tipo de objeto que la ejecute.

Por ejemplo, si tenemos una lista de `Persona` que contiene tanto `Estudiantes` como `Profesores`, y llamamos al método `presentarse()` de cada uno, el resultado dependerá del tipo real del objeto:

```
List<Persona> personas = List.of(
    new Estudiante("Laura", 21, "Ingeniería"),
    new Profesor("Carlos", 45, "Matemáticas"));

for (Persona p : personas) {p.presentarse(); // Cada clase ejecuta su propia versión del método}
```

Aunque el código trata a todos como `Persona`, el método correcto se selecciona en tiempo de ejecución (esto se llama *dynamic binding* o enlace dinámico). El programa se vuelve así más extensible y flexible, capaz de procesar distintos tipos de objetos sin conocerlos de antemano.

La palabra clave super

Cuando una subclase necesita reutilizar parte del comportamiento de la superclase, puede usar `super`:

- `super()` invoca el constructor padre.
- `super.metodo()` llama a una versión anterior de un método, útil cuando se sobrescribe.

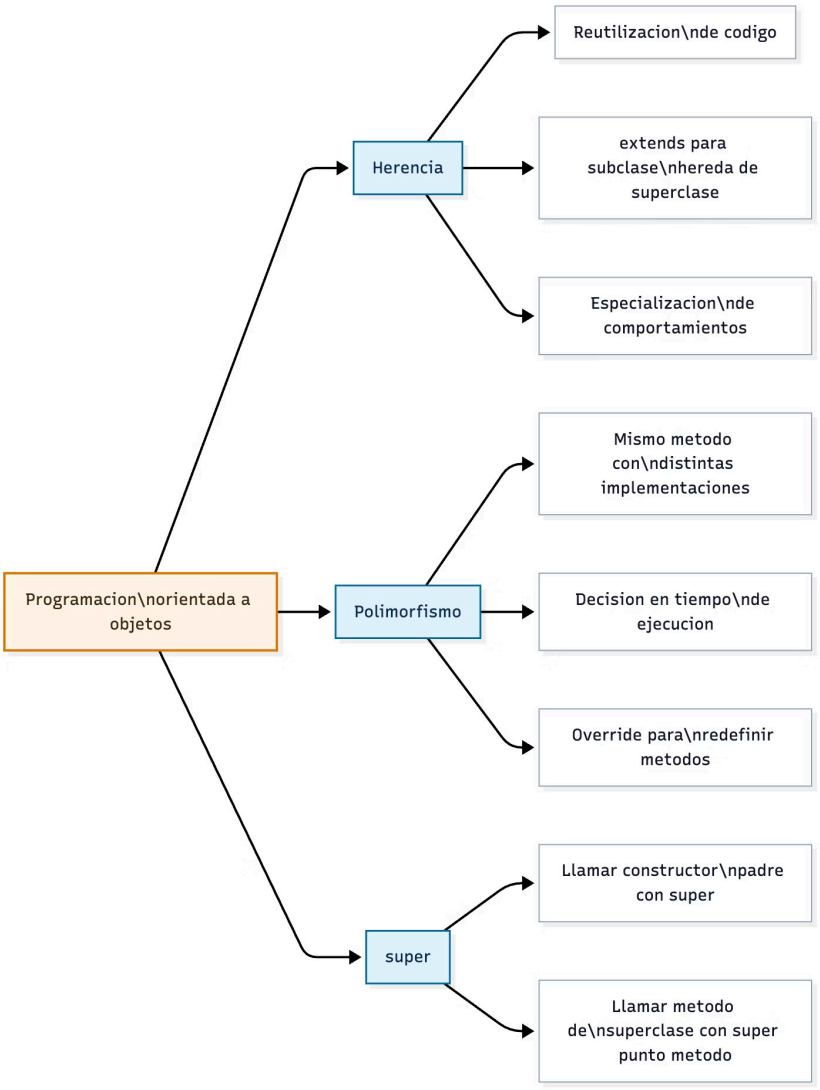
Por ejemplo:

```
@Override
void presentarse() {super.presentarse();
    System.out.println("Además, soy estudiante de " + carrera);}
```

Esto mantiene la coherencia entre clases y evita duplicar código. En conjunto, herencia y polimorfismo forman la base de la extensibilidad del software orientado a objetos: el código puede evolucionar sin romper lo existente.

Esquema visual

Diagrama conceptual: Herencia y Polimorfismo en Java



Descripción del esquema:

Herencia conecta clases en jerarquías. Cada subclase hereda datos y comportamientos de la clase padre, que pueden ampliarse o sobrescribirse.

Polimorfismo se representa como la capacidad de que distintos objetos respondan de manera diferente al mismo método. Es el motor de la flexibilidad en los programas orientados a objetos.

super actúa como enlace entre niveles de la jerarquía, permitiendo reutilizar constructores y lógica del padre sin duplicar código.

El conjunto crea un sistema coherente, mantenible y escalable.

Caso de estudio – Sistema de empleados GOOGLE

Contexto

Google gestiona una plantilla superior a 150.000 empleados de diferentes perfiles: ingenieros, diseñadores, gestores de producto, managers... Cada uno comparte ciertas características comunes (nombre, identificación, salario base), pero también tiene comportamientos específicos.

Estrategia

Para mantener su sistema de nóminas eficiente, Google modela sus empleados con una jerarquía de clases. La clase base `Empleado` representa los elementos comunes, mientras que las subclases `Ingeniero`, `Manager` y `Diseñador` especializan su comportamiento.

Ejemplo simplificado:

```
abstract class Empleado {
    String nombre;
    double salarioBase;

    Empleado(String nombre, double salarioBase) {
        this.nombre = nombre;
        this.salarioBase = salarioBase;
    }

    abstract double calcularBono(); // Método polimórfico
}

class Ingeniero extends Empleado {
    int bugsResueltos;

    Ingeniero(String n, double s, int b) {
        super(n, s);
        this.bugsResueltos = b;
```

```
 }

    @Override
    double calcularBono() {
        return salarioBase * 0.10 + bugsResueltos * 2;
    }
}

class Manager extends Empleado {
    int equipos;

    Manager(String n, double s, int e) {
        super(n, s);
        this.equipos = e;
    }

    @Override
    double calcularBono() {
        return salarioBase * 0.15 + equipos * 500;
    }
}
```

Resultado

El sistema de nóminas puede procesar a todos los empleados de forma uniforme:

```
List<Empleado> lista = List.of(
    new Ingeniero("Ana", 50000, 120),
    new Manager("Luis", 80000, 3)
);

double totalBonos = 0;
for (Empleado e : lista) {
    totalBonos += e.calcularBono(); // Polimorfismo en acción
}
```

El mismo método `calcularBono()` produce resultados diferentes según el tipo de empleado, sin necesidad de condicionales. Este diseño permite que Google añada nuevos tipos de empleados (por ejemplo, `DiseñadorUX`) sin modificar el código existente. El sistema escala y se mantiene con facilidad, garantizando coherencia y extensibilidad.

Herramientas y consejos básicos

Usa @Override siempre que sobrescribas un método

Evita errores tipográficos y asegura que el método realmente existe en la clase padre. Si te equivocas en la firma, el compilador avisará:

```
@Override
void calcularBono() { ... }
```

Aplica el principio "IS-A" (es-un) antes de heredar

Usa herencia solo cuando la relación entre clases sea realmente jerárquica. Ejemplo: un `Perro` es un `Animal`. En cambio, un `Coche` tiene un `Motor`, no es un `Motor` → en ese caso se usa composición, no herencia.

Combina herencia con composición

Una subclase puede tener objetos de otras clases como atributos. Por ejemplo, un `Empleado` puede tener un `Departamento`, sin necesidad de heredar de él.

Implementa correctamente equals() y hashCode()

Si tus objetos heredados se usan en colecciones (como `HashMap` o `Set`), redefine estos métodos para evitar duplicados o errores de comparación.

Visualiza jerarquías con UML

Herramientas como IntelliJ UML Viewer, StarUML o Lucidchart te permiten ver relaciones entre clases (hereda de, usa, compone), ayudando a detectar redundancias.

Evita herencias profundas

Más de tres niveles de herencia suelen indicar un mal diseño. Es preferible crear clases más simples y utilizar interfaces o composición para extender comportamientos.

Mitos y realidades

❌ Mito 1: "La herencia siempre mejora el diseño del código."

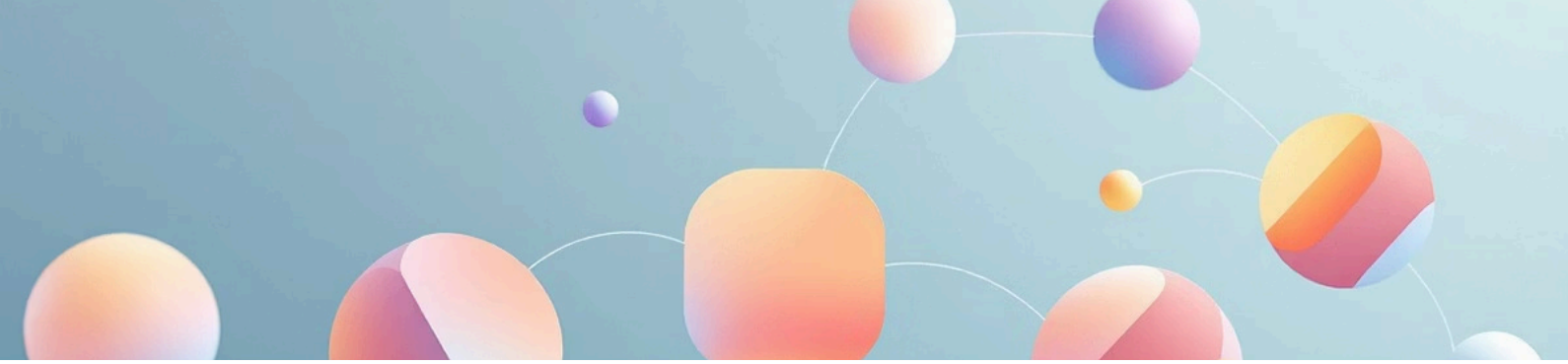
→ **FALSO.** Un uso excesivo de herencia genera acoplamiento fuerte: si cambias algo en la superclase, puedes romper todo el sistema. **Realidad:** la herencia debe usarse solo cuando existe una relación estable y lógica "es-un". Si no, es mejor usar composición o interfaces.

❌ Mito 2: "El polimorfismo solo existe con herencia."

→ **FALSO.** En Java, el polimorfismo también puede lograrse mediante interfaces. Una clase puede implementar múltiples interfaces y comportarse de formas distintas según el contexto. Esto ofrece una forma más flexible de extender comportamiento sin la rigidez de una jerarquía de herencia única.

Resumen final

- **Herencia:** permite crear clases hijas a partir de clases padre (`extends`), reutilizando y especializando código.
- **Polimorfismo:** mismo método, diferentes comportamientos según el tipo real del objeto (decidido en tiempo de ejecución).
- **super:** accede a constructores y métodos de la superclase.
- **@Override:** asegura la correcta sobrescritura de métodos.
- **Ejemplo Google:** 150.000+ empleados procesados polimórficamente con `calcularBono()`.
- **Buena práctica:** usar herencia solo si la relación "es-un" es clara; preferir composición en los demás casos.



Sesión 13 – Interfaces y Clases Abstractas

La Programación Orientada a Objetos (POO) no se limita a heredar y crear clases. A medida que las aplicaciones crecen, necesitamos formas más flexibles de organizar el comportamiento. Aquí entran en juego dos elementos esenciales: las **interfaces** y las **clases abstractas**. Ambas permiten definir contratos comunes entre diferentes clases, pero lo hacen de manera distinta. Comprender cuándo usar cada una es clave para diseñar software reutilizable, mantenible y fácil de ampliar.

Interfaces: el contrato de comportamiento

Una `interface` define qué debe hacer una clase, pero no cómo. Es un contrato: si una clase implementa una interface, se compromete a tener los métodos definidos en ella. Ejemplo:

```
interface Pagable {
    void procesarPago(double monto);
    boolean validarMetodo();
}
```

Cualquier clase que implemente `Pagable` debe incluir esos métodos:

```
class TarjetaCredito implements Pagable {
    public void procesarPago(double monto) {
        System.out.println("Pago con tarjeta de " + monto + " € procesado");
    }

    public boolean validarMetodo() {
        return true; }}
```

Gracias a las interfaces, diferentes clases pueden compartir comportamientos comunes sin estar relacionadas jerárquicamente. Por ejemplo, tanto `TarjetaCredito` como `PayPalBalance` o `TransferenciaBancaria` pueden implementar `Pagable`, aunque no hereden una de otra.

En Java, se utiliza la palabra clave `implements` para establecer esa relación:

```
class PayPalBalance implements Pagable { ... }
```

Desde Java 8, las interfaces pueden tener métodos `default` (con implementación) y métodos `static` (útiles sin crear instancias). Esto amplía su poder y las hace más parecidas a clases abstractas, pero sin estado interno.

Clases abstractas: el punto intermedio

Una clase abstracta no puede instanciarse directamente. Sirve como modelo parcial que agrupa código común, pero deja algunos métodos sin definir. Se declara con la palabra clave `abstract`:

```
abstract class Empleado {
    String nombre;
    double salarioBase;

    Empleado(String nombre, double salarioBase) {
        this.nombre = nombre;
        this.salarioBase = salarioBase;
    }

    abstract double calcularBono(); // Obligatorio implementar
}
```

Las clases hijas deben implementar los métodos abstractos:

```
class Ingeniero extends Empleado {
    Ingeniero(String n, double s) {
        super(n, s);
    }

    @Override
    double calcularBono() {
        return salarioBase * 0.1;
    }}
```

La diferencia clave es que una clase abstracta puede contener tanto métodos implementados como abstractos, y también mantener estado (atributos con valores). Por tanto, se usa cuando hay comportamiento compartido entre subclases.

Interfaces vs. Clases abstractas

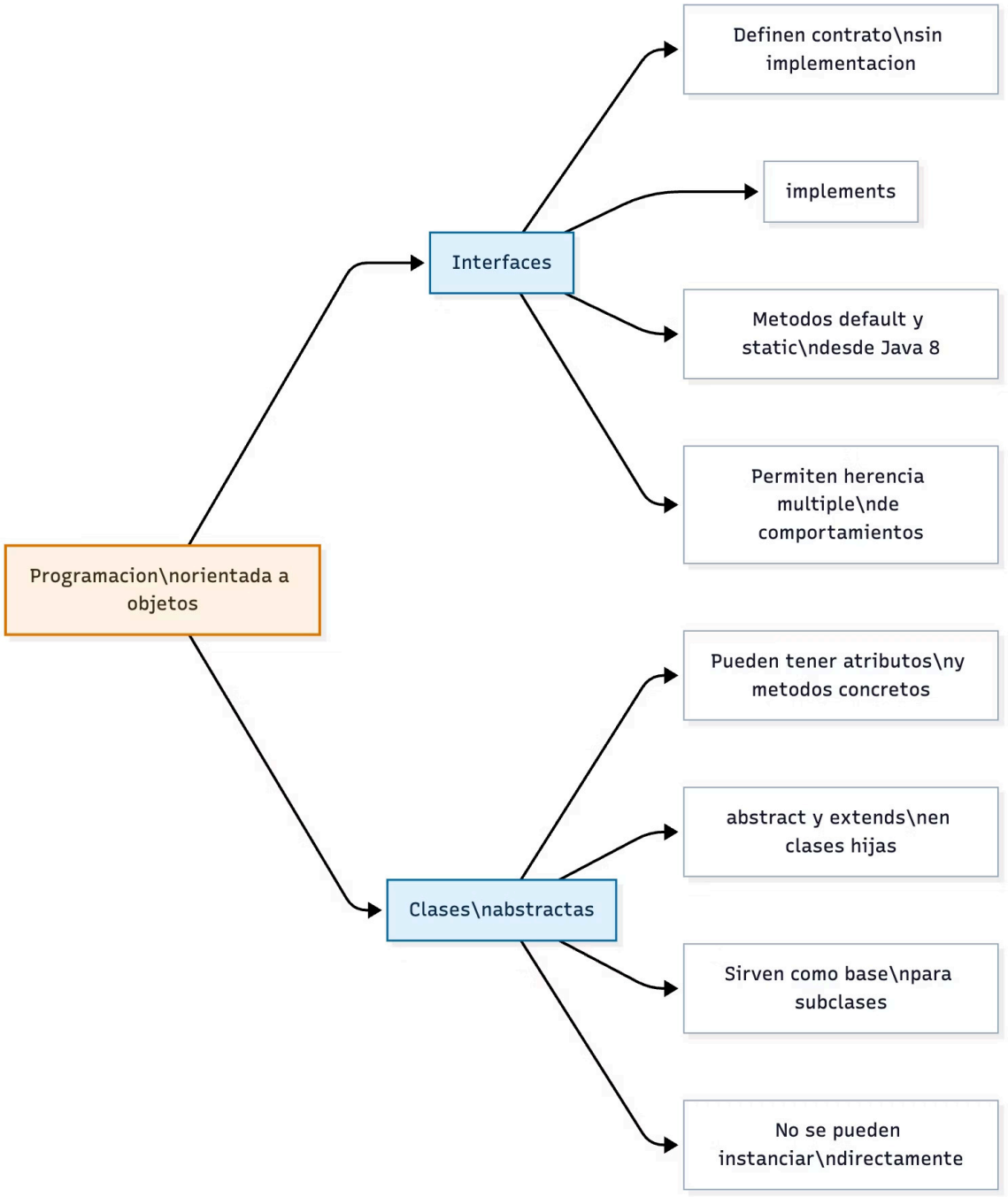
Característica	Interface	Clase abstracta
Palabra clave	`implements`	`extends`
Estado interno	No puede tener atributos de instancia	Sí puede tener atributos
Métodos	Todos abstractos (o default/static desde Java 8)	Pueden ser abstractos y concretos
Herencia	Múltiple (una clase puede implementar varias interfaces)	Única (solo puede heredar de una clase abstracta)
Uso recomendado	Para definir qué hace una clase	Para definir qué es una clase base con lógica común

En resumen:

- Usa **interfaces** para definir capacidades o roles (por ejemplo, `Serializable`, `Comparable`, `Runnable`).
- Usa **clases abstractas** para compartir código y estructura común entre clases emparentadas.

Esquema visual

Diagrama conceptual: Interfaces y clases abstractas en Java



Descripción del esquema:

En el lado izquierdo, las **interfaces** representan contratos: las clases que las implementan se comprometen a cumplir con ciertos métodos, sin importar su posición en la jerarquía.

En el lado derecho, las **clases abstractas** sirven de plantillas parciales, con atributos y comportamientos comunes para sus subclases.

Ambas fluyen desde el nodo central "Programación Orientada a Objetos", porque ambas fomentan la reutilización y el polimorfismo, pero desde perspectivas diferentes.

Caso de estudio – Arquitectura de pagos de PayPal

Contexto

PayPal gestiona más de 19 mil millones de transacciones anuales a través de múltiples métodos de pago: tarjetas de crédito, saldo interno, transferencias, criptomonedas, etc. Cada método de pago tiene sus propias validaciones y comisiones, pero todos deben integrarse dentro del mismo sistema y responder al mismo conjunto de operaciones: procesar, validar y calcular comisión.

Estrategia

Para mantener el sistema escalable y modular, PayPal aplica la combinación de interfaces e implementación polimórfica. Define la interface común:

```
interface MetodoPago {
    void procesar(double monto);
    boolean validar();
    double calcularComision(double
monto);
}
```

Implementa clases específicas:

```
class TarjetaCredito implements MetodoPago {
    public void procesar(double monto) {
        System.out.println("Pago con tarjeta: " + monto + "€");
    }
    public boolean validar() {
        return true; // Validaciones CVV, fecha...
    }
    public double calcularComision(double monto) {
        return monto * 0.03; // 3% de comisión}}

class PayPalBalance implements MetodoPago {
    public void procesar(double monto) {
        System.out.println("Pago desde saldo PayPal: " + monto + "€");
    }
    public boolean validar() {
        return true; // Verifica saldo disponible
    }
    public double calcularComision(double monto) {
        return monto * 0.02; // 2% de comisión}}
```

Gestiona los pagos de forma unificada:

```
MetodoPago metodo = new TarjetaCredito();
metodo.procesar(100);
metodo.calcularComision(100);
```

Resultado

El sistema de PayPal puede añadir nuevos métodos de pago sin modificar el código existente. Solo se necesita implementar la interface `MetodoPago`. Esto garantiza bajo acoplamiento y alta cohesión, principios esenciales del diseño orientado a objetos.

Además, PayPal combina interfaces con clases abstractas para gestionar comportamientos comunes. Por ejemplo, una clase `PagoBase` podría incluir atributos como `monto`, `fecha`, `usuario` y métodos concretos para registrar logs, mientras las subclases definen las operaciones específicas.

Esta estructura híbrida combina la flexibilidad de las interfaces con la reutilización de las clases abstractas.

Herramientas y consejos

1 Aplica interfaces para definir capacidades

Por ejemplo:

- **Comparable:** permite ordenar objetos.
- **Serializable:** hace persistibles los objetos.
- **Runnable:** define tareas que pueden ejecutarse en hilos.

Implementarlas correctamente evita duplicar código y permite integrarte fácilmente con las librerías de Java.

2 Usa clases abstractas cuando exista comportamiento compartido

Si varias clases comparten atributos o lógica base, colócalos en una clase abstracta. Ejemplo: `Vehiculo` podría tener `velocidad`, `combustible` y método `arrancar()`, mientras `Coche` y `Moto` implementan `calcularConsumo()`.

3 Evita abusar de la herencia múltiple mediante interfaces

Aunque una clase puede implementar muchas interfaces, cada una debe tener un propósito claro. Un exceso de interfaces genera confusión y dificultad de mantenimiento.

4 Emplea default y static sabiamente

- Usa `default` para añadir funcionalidad genérica a una interface sin romper implementaciones previas.
- Usa `static` para utilidades relacionadas, por ejemplo:

```
interface UtilPago {static boolean validarMonto(double monto) {return monto > 0;}}
```

5 Visualiza relaciones de implementación

Herramientas como IntelliJ UML, Visual Paradigm o PlantUML te ayudan a identificar si tu diseño mezcla correctamente interfaces y clases abstractas o si estás sobrecargando la jerarquía.

6 Aplica pruebas unitarias

Usa JUnit o Mockito para probar que todas las clases que implementan una interface cumplen correctamente los contratos definidos.

Mitos y realidades

❌ Mito 1: "Las interfaces solo pueden tener métodos abstractos."

→ **FALSO.** Desde Java 8, las interfaces pueden contener métodos `default` y `static` con implementación, lo que permite agregar nueva funcionalidad sin romper el código existente. **Realidad:** esto hace que las interfaces sean más poderosas y compatibles hacia atrás, permitiendo evolución controlada del software.

❌ Mito 2: "Las clases abstractas son obsoletas con las interfaces modernas."

→ **FALSO.** Aunque las interfaces han evolucionado, las clases abstractas siguen siendo esenciales cuando necesitas compartir código base (atributos o métodos comunes) entre subclases. **Realidad:** ambas herramientas se complementan; la interface define el *qué*, la clase abstracta el *cómo* parcial.

6. Resumen final

- **Interface:** define un contrato (qué debe hacer una clase). Se implementa con `implements`.
- **Clase abstracta:** punto medio entre clase e interface; combina código común con métodos abstractos.
- **Métodos default y static:** introducidos en Java 8 para ampliar interfaces sin romper compatibilidad.
- **PayPal:** usa interfaces (`MetodoPago`) para gestionar múltiples métodos de pago con polimorfismo.
- **Diferencia clave:** una clase puede implementar múltiples interfaces, pero solo heredar de una clase abstracta.
- **Uso ideal:** interface = comportamiento; abstracta = estructura compartida