«Talento Tech»

Back-End

# Java

Clase 07

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

# Clase 7: Herencia y polimorfismo

## Índice



---

## Objetivos de la clase

- Comprender el concepto de **herencia** en Java: qué es una superclase, una subclase y cómo se relacionan.
- Aprender a crear subclases que extiendan una superclase utilizando `extends`.
- Comprender el uso de `super` y `this` para acceder a la superclase y al propio objeto respectivamente.
- Explorar el **polimorfismo**: la habilidad de tratar diferentes subclases a través de una referencia de la superclase.
- Entender la **sobrescritura de métodos** (override) para especializar comportamientos en subclases.
- Introducir **clases abstractas** e **interfaces** como herramientas para definir contratos y aprovechar la reutilización y extensibilidad.

---

○

Buenos Aires aprende
Agencia de Habilidades para el Futuro

## Bienvenida a TechLab

Talento Lab

¡Les damos nuevamente la bienvenida a **TechLab**! Hasta ahora, hemos comprendido la esencia de la POO a través de la encapsulación, la creación de clases y objetos, la modularización y la colaboración entre ellos. Sin embargo, el verdadero poder de la POO se despliega cuando aplicamos herencia y polimorfismo.

La **herencia** permite crear jerarquías de clases donde las subclases especializan el comportamiento y estado de una clase base. El **polimorfismo** permite tratar objetos de distintas subclases como si fuesen instancias de su superclase, habilitando la intercambiabilidad y la extensibilidad del sistema.

En TechLab, la cantidad de productos, usuarios y usuarias y lógica de negocios crece. Con herencia y polimorfismo, el equipo podrá diseñar soluciones más extensibles, evitando duplicar código y facilitando la incorporación de nuevas categorías o comportamientos sin romper la arquitectura existente.

---

## Concepto de herencia en Java

La herencia es un mecanismo que permite que una clase (subclase) extienda otra clase (superclase), heredando sus atributos y métodos. La subclase puede agregar nuevos atributos y métodos, o sobrescribir métodos existentes.

Beneficios de la herencia:

- **Reutilización de código:** evita duplicación entre clases con características comunes.
- **Organización jerárquica:** permite reflejar el dominio del problema en estructuras más naturales.
- **Extensibilidad:** facilita la creación de nuevos tipos de objetos basados en clases existentes.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

### Problema: categorías de productos especializadas

**Contexto:** Silvia necesita que, además de `Producto`, existan `Bebida` con un atributo `volumenEnLitros` y `Comida` con un atributo `fechaVencimiento`. Estos tipos de productos deben manejarse de manera distinta sin duplicar el código base.

**Solución:** Crear subclases `Bebida` y `Comida` que extiendan `Producto`, heredando atributos como `nombre`, `precio`, `cantidadEnStock` y agregando sus propios atributos.

---

## Creación de subclases y superclases

En Java la herencia se declara con la palabra clave `extends`. Por ejemplo:

```
public class Bebida extends Producto {
   private double volumenEnLitros;

   public Bebida(String nombre, double precio, int cantidadEnStock,
double volumenEnLitros) {
        super(nombre, precio, cantidadEnStock);
        this.volumenEnLitros = volumenEnLitros;
   }
}
```

`Bebida` hereda de `Producto`. Al crear el constructor, llamamos a `super(...)` para invocar el constructor de la superclase. De este modo, `Bebida` adquiere los atributos y métodos de `Producto`, pudiendo agregar o especializar su comportamiento.

### Problema: Creando una jerarquía de productos

**Contexto:** Antes, `Producto` era una clase que abarcaba todo tipo de ítems. Ahora, con herencia, Matías y Sabrina crean `Bebida` y `Comida` como subclases. `Bebida` hereda atributos básicos de `Producto` y añade `volumenEnLitros`. `Comida` hereda lo mismo pero añade `fechaVencimiento`.

**Solución:**

```
public class Comida extends Producto {
   private String fechaVencimiento;

   public Comida(String nombre, double precio, int cantidadEnStock,
String fechaVencimiento) {
        super(nombre, precio, cantidadEnStock);
        this.fechaVencimiento = fechaVencimiento;
   }
}
```

Ahora, `Bebida` y `Comida` reutilizan código de `Producto` evitando duplicaciones.

---

## Uso de super y this

`this` hace referencia al objeto actual, mientras que `super` hace referencia a la superclase. Dentro del constructor de una subclase, se puede usar `super(...)` para llamar al constructor de la superclase. También se puede usar `super.metodo()` para acceder a un método sobrescrito de la superclase.

### Problema: reutilizar código del constructor y métodos de la superclase

**Contexto:** `Bebida` debe inicializar `nombre`, `precio` y `cantidadEnStock` como lo hace `Producto`. Gracias a `super(...)`, no es necesario repetir ese código, solo se delega el trabajo al constructor de `Producto`.

**Solución:**

```
public Bebida(String nombre, double precio, int cantidadEnStock, double
volumen) {
    super(nombre, precio, cantidadEnStock);
    this.volumenEnLitros = volumen;
}
```

Se evita duplicación y se centraliza la lógica común en la superclase.

---

## Polimorfismo y sobrescritura de métodos

El **polimorfismo** permite usar una referencia de la superclase para apuntar a objetos de sus subclases. Por ejemplo, podemos tener un `Producto p = new Bebida(...);` y `Producto q = new Comida(...);`. Aunque `p` y `q` sean `Producto`, cuando invoquemos métodos sobre ellos, se ejecutará la versión sobrescrita en su subclase correspondiente.

La **sobrescritura (override)** consiste en redefinir un método heredado de la superclase, adaptando su comportamiento en la subclase. Se utiliza la anotación `@Override` para mayor claridad.

```
public class Bebida extends Producto {
    // ...
    @Override
    public double calcularPrecioFinal() {
        // Por ejemplo, las bebidas tienen un descuento del 5%
        return getPrecio() * 0.95;
    }
}
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

### Problema: distintos cálculos de descuento según el tipo de producto

**Contexto:** Silvia pide que las bebidas tengan un descuento distinto al de las comidas. Con polimorfismo, Matías y Sabrina pueden tratar a todos los productos como `Producto`, pero cada subclase implementa `calcularPrecioFinal()` a su manera.

**Solución:**

- `Producto` define un método `calcularPrecioFinal()` por defecto.
- `Bebida` y `Comida` sobrescriben este método.
- Cuando se iteran productos en un `ArrayList<Producto>`, cada uno ejecuta su lógica especial sin `if` ni `switch`.

---

## Clases abstractas e interfaces

Una **clase abstracta** es una clase que no puede instanciarse directamente. Se utiliza para proveer una base común para subclases, definiendo comportamiento estándar y declarando métodos abstractos que las subclases deben implementar. Un método abstracto no tiene cuerpo en la clase abstracta.

Las **interfaces** definen un conjunto de métodos que una clase debe implementar, sin proveer implementación. Son una forma de lograr polimorfismo sin herencia múltiple, permitiendo que una clase implemente múltiples interfaces.

```
public abstract class ProductoBase {
   private String nombre;

   public ProductoBase(String nombre) {
       this.nombre = nombre;
   }
   public abstract double calcularPrecioFinal();
}
```

Las subclases de `ProductoBase` deben implementar `calcularPrecioFinal()`.

Las interfaces, en cambio, definen un contrato:

```
public interface Descontable {
    double aplicarDescuento(double porcentaje);
}
```

Cualquier clase que implemente `Descontable` debe proveer `aplicarDescuento`.

## Problema: definir un contrato de métodos para ciertas operaciones

**Contexto:** Silvia quiere que todos los productos tengan un método para aplicar descuentos, pero no todos los productos calculan el descuento de la misma forma. Con una interfaz `Descontable`, se fuerza a las subclases a implementar `aplicarDescuento(...)`.

**Solución:**

- Crear la interfaz `Descontable`.
- Que `Bebida` y `Comida` implementen `Descontable` y definan su propia lógica de descuento.

Este mecanismo permite flexibilidad y asegura que todas las clases que “prometen” ser descontables cumplan con el contrato.

---

Buenos Aires aprende
Agencia de Habilidades para el Futuro

## Situación inicial en TechLab

Talento Lab

Silvia, la Product Owner, observa que actualmente hay una sola clase `Producto`, pero el catálogo ha crecido y cada tipo de producto tiene características y reglas distintas. Por ejemplo, Té y Café. Alguna requiere cálculos de descuento particulares, otras tienen restricciones de stock o fecha de vencimiento.

Matías y Sabrina se dan cuenta de que no pueden seguir agregando `if` y `switch` dentro de `Producto` para cada caso especial. Necesitan una forma más elegante de modelar las diferencias: la **herencia** les permitirá crear una clase base `Producto` y luego `Te` y `Café` como subclases que especialicen el comportamiento. El polimorfismo les permitirá tratar a todos ellos como `Producto` al mismo tiempo que cada uno sabe hacer su trabajo de forma independiente.

Además, en algunos casos se requerirá definir un comportamiento sin implementar completamente la lógica en la superclase, o establecer un contrato de métodos que las subclases deben cumplir. Aquí entran en juego las clases abstractas y las interfaces.

Para cumplir con necesidades el equipo de desarrollo (Matías, Sabrina) y vos deberán:

---

## Ejercicios prácticos

1. **Herencia básica:**
    - Creá la clase abstracta `Producto` con un atributo `nombre` y un método abstracto `calcularPrecioFinal()`.
    - Creá `Te` y `Café` que extiendan `Producto`.
    - Implementá `calcularPrecioFinal()` en cada subclase.

2. **Polimorfismo:**
    - Creá un `ArrayList<Producto>` y agrega instancias de `Te` y `Cafe`.
    - Iterá sobre la lista y llamá a `calcularPrecioFinal()` en cada uno. Observá cómo se ejecuta la versión correspondiente a cada subclase.
3. **Uso de super y override:**
    - En `Cafe`, agregá un constructor que llame a `super(...)` para inicializar atributos comunes.
    - Sobrescribí algún método de `Producto` y llamá a `super` en su interior para reutilizar parte de la lógica.
4. **Interfaces:**
    - Creá la interfaz `Descontable` con el método `aplicarDescuento(double porcentaje)`.
    - Hacé que `Te` y `Cafe` la implementen.
    - Probá aplicar descuentos a diferentes productos.

---

## Materiales y recursos adicionales

- Documentación Oficial de Java: Herencia y Polimorfismo
- Tutoriales de Oracle sobre Clases Abstractas e Interfaces
- Videos recomendados en YouTube sobre herencia, polimorfismo, clases abstractas e interfaces.

---

## Preguntas para reflexionar

- ¿Cómo la herencia evita la duplicación de código y facilita la extensibilidad del sistema en TechLab?
- ¿En qué casos usarías una clase abstracta versus una interfaz?
- ¿Cómo el polimorfismo simplifica el manejo de colecciones que contienen distintos tipos de objetos relacionados?

---

Buenos Aires aprende
Agencia de Habilidades para el Futuro

## Próximos pasos

En la próxima clase, profundizaremos en temas más avanzados relacionados con la POO, como patrones de diseño o manejo de errores en jerarquías de clases. Mientras tanto, practicá creando jerarquías, sobrescribiendo métodos y aplicando interfaces. Esto te permitirá sentar bases sólidas para un proyecto orientado a objetos maduro y escalable en TechLab.

**¡Felicitaciones por llegar hasta aquí!** Con herencia y polimorfismo, ahora tenés las herramientas para diseñar sistemas más potentes, reutilizables y mantenibles.


Buenos Aires Ciudad