# Herencia y Polimorfismo en Java: Clases, Interfaces y su Relación

La Programación Orientada a Objetos (POO) se basa en varios pilares, y la **herencia** es uno de los más importantes. En Java, la herencia permite que las clases compartan atributos y comportamientos, promoviendo la **reutilización de código** y estableciendo relaciones lógicas entre ellas. Este concepto va de la mano con el **polimorfismo**, la capacidad de los objetos de diferentes clases de ser tratados de manera uniforme a través de una superclase o interfaz común.

## ¿Qué es la Herencia?

La **herencia** es un mecanismo por el cual una clase (la **subclase** o clase hija) puede adquirir las propiedades (atributos) y comportamientos (métodos) de otra clase (la **superclase** o clase padre). Esto modela una relación "es un" (is-a).

### Beneficios Clave:

* **Reutilización de Código:** Evita la duplicidad, ya que las características comunes se definen una sola vez en la superclase.
* **Organización Lógica:** Estructura tu código de forma jerárquica, facilitando la comprensión y el mantenimiento.
* **Extensibilidad:** Facilita la adición de nuevas funcionalidades sin modificar el código existente.

### Implementación en Java: extends

En Java, usamos la palabra clave **extends** para establecer la herencia entre clases. Una subclase solo puede extender una única superclase.

Java

class Subclase extends Superclase {  
 // Miembros específicos de la subclase  
}

### Ejemplo Simple: Animales

Imaginemos que tenemos diferentes animales que comparten la característica de tener un nombre y de poder comer().

Java

// Superclase: Animal  
class Animal {  
 String nombre;  
  
 public Animal(String nombre) {  
 this.nombre = nombre;  
 }  
  
 public void comer() {  
 System.out.println(nombre + " está comiendo.");  
 }  
}  
  
// Subclase: Perro  
class Perro extends Animal {  
 public Perro(String nombre) {  
 super(nombre); // Llama al constructor de la superclase Animal  
 }  
  
 public void ladrar() { // Método específico de Perro  
 System.out.println(nombre + " está ladrando: ¡Guau guau!");  
 }  
}  
  
// Subclase: Gato  
class Gato extends Animal {  
 public Gato(String nombre) {  
 super(nombre); // Llama al constructor de la superclase Animal  
 }  
  
 public void maullar() { // Método específico de Gato  
 System.out.println(nombre + " está maullando: ¡Miau miau!");  
 }  
}  
  
public class Granja {  
 public static void main(String[] args) {  
 Perro miPerro = new Perro("Buddy");  
 miPerro.comer(); // Heredado de Animal  
 miPerro.ladrar(); // Específico de Perro  
  
 Gato miGato = new Gato("Whiskers");  
 miGato.comer(); // Heredado de Animal  
 miGato.maullar(); // Específico de Gato  
 }  
}

**Explicación:** Perro y Gato heredan nombre y comer() de Animal. Cada uno añade su propio comportamiento único. La llamada a super(nombre) en los constructores de las subclases es vital para inicializar la parte heredada.

## Polimorfismo y Sobrescritura de Métodos (@Override)

El **polimorfismo** (del griego "muchas formas") permite que objetos de diferentes clases sean tratados como objetos de una superclase común o una interfaz común. Esto se logra a menudo mediante la **sobrescritura de métodos**.

### Sobrescritura de Métodos: @Override

Una subclase puede proporcionar su propia implementación para un método que ya está definido en su superclase. Esto se conoce como **sobrescritura** (method overriding). La anotación @Override es opcional pero muy recomendable para indicar que un método está sobrescribiendo uno de la superclase, ayudando a detectar errores.

### Ejemplo: Formas Geométricas

Consideremos una Forma genérica que puede ser dibujada(). Círculo y Cuadrado son formas específicas que se dibujan de manera diferente.

Java

// Superclase: Forma  
class Forma {  
 public void dibujar() {  
 System.out.println("Dibujando una forma genérica.");  
 }  
}  
  
// Subclase: Circulo  
class Circulo extends Forma {  
 @Override // Sobrescritura del método dibujar()  
 public void dibujar() {  
 System.out.println("Dibujando un círculo.");  
 }  
 public void calcularRadio() { /\* ... \*/ } // Método específico  
}  
  
// Subclase: Cuadrado  
class Cuadrado extends Forma {  
 @Override // Sobrescritura del método dibujar()  
 public void dibujar() {  
 System.out.println("Dibujando un cuadrado.");  
 }  
 public void calcularLado() { /\* ... \*/ } // Método específico  
}  
  
public class Dibujo {  
 public static void main(String[] args) {  
 Forma miCirculo = new Circulo(); // Polimorfismo: una Forma puede ser un Circulo  
 miCirculo.dibujar(); // Ejecuta el dibujar() de Circulo  
  
 Forma miCuadrado = new Cuadrado(); // Polimorfismo: una Forma puede ser un Cuadrado  
 miCuadrado.dibujar(); // Ejecuta el dibujar() de Cuadrado  
 }  
}

**Explicación:** Aunque miCirculo y miCuadrado son de tipo Forma, el método dibujar() ejecutado es el de la clase real del objeto (Circulo o Cuadrado), demostrando el polimorfismo.

## Clases Abstractas y Finales: Controlando la Herencia

Java nos ofrece modificadores para controlar cómo se comportan las clases en una jerarquía de herencia.

### Clases Abstractas (abstract class)

Una **clase abstracta** es una clase que **no puede ser instanciada directamente** (no puedes crear objetos de ella). Sirve como una plantilla o base para otras clases que deben implementar sus métodos abstractos.

* **Propósito:** Define una estructura común para un grupo de subclases, forzándolas a implementar ciertos métodos. Ideal cuando el concepto de la superclase es demasiado general para ser un objeto concreto (ej., "Figura" no es una figura específica).
* **Métodos Abstractos:** Se declaran con public abstract void miMetodo(); y no tienen implementación. Las subclases *deben* sobrescribirlos.
* **Métodos Concretos:** Una clase abstracta puede tener métodos con implementación normal.

#### Ejemplo: Figuras con Área

Si una Figura abstracta tiene un método calcularArea() abstracto, cada figura concreta (Rectángulo, Triángulo) deberá proporcionar su propia implementación.

Java

// Superclase Abstracta: Figura  
abstract class Figura {  
 String color;  
  
 public Figura(String color) {  
 this.color = color;  
 }  
 public abstract double calcularArea(); // Método abstracto  
 public void mostrarColor() { // Método concreto  
 System.out.println("Esta figura es de color: " + color);  
 }  
}  
  
// Subclase Concreta: Rectangulo  
class Rectangulo extends Figura {  
 double largo, ancho;  
 public Rectangulo(String color, double largo, double ancho) {  
 super(color); this.largo = largo; this.ancho = ancho;  
 }  
 @Override  
 public double calcularArea() {  
 return largo \* ancho; // Implementación obligatoria  
 }  
}  
  
public class DemostracionAbstracta {  
 public static void main(String[] args) {  
 // Figura miFigura = new Figura("Blanco"); // ¡ERROR! No se puede instanciar Figura  
 Rectangulo rect = new Rectangulo("Rojo", 5, 3);  
 rect.mostrarColor();  
 System.out.println("Área: " + rect.calcularArea());  
 }  
}

**Explicación:** No podemos crear un objeto Figura directamente. Rectangulo debe implementar calcularArea() para ser una clase concreta.

### Clases Finales (final class)

Una **clase final** es una clase que **no puede ser extendida** por ninguna otra clase.

* **Propósito:** Se usa para prevenir la herencia y asegurar que la implementación de la clase no sea modificada o sobreescrita por subclases.
* **Uso Común:** Para clases de utilidad inmutables o clases fundamentales que no deben ser alteradas, como la clase String en Java.

#### Ejemplo: Utilidades Matemáticas

Una clase de utilidad matemática podría ser final para asegurar que nadie modifique su comportamiento.

Java

final class UtileriaMatematica {  
 private UtileriaMatematica() {} // Constructor privado para evitar instanciación  
 public static int sumar(int a, int b) {  
 return a + b;  
 }  
}  
  
// class MiNuevaUtilidad extends UtileriaMatematica { } // ¡ERROR DE COMPILACIÓN!  
  
public class DemostracionFinal {  
 public static void main(String[] args) {  
 System.out.println("Suma: " + UtileriaMatematica.sumar(7, 8));  
 }  
}

**Explicación:** Intentar extender UtileriaMatematica resultaría en un error de compilación.

## Interfaces: Contratos de Comportamiento

Las **interfaces** son un concepto crucial que complementa la herencia de clases en Java. A diferencia de las clases (incluso las abstractas), una interfaz es un **contrato puro de comportamiento**.

* **Propósito:** Define un conjunto de métodos que una clase *debe* implementar si quiere adherirse a ese contrato. Permiten el **polimorfismo de comportamiento** y resuelven el problema de la herencia múltiple (que Java no permite para clases).
* **implements:** Las clases usan la palabra clave **implements** para indicar que están adoptando una o más interfaces. Una clase puede implementar múltiples interfaces.
* **Contenido:**
  + Hasta Java 8: Solo podían tener métodos abstractos (implícitamente public abstract) y constantes (implícitamente public static final).
  + Desde Java 8: Pueden tener métodos default (con implementación por defecto) y métodos static.

### Ejemplo: Comportamientos de Movimiento

Un Animal puede comer(), pero algunos animales pueden Volar y otros pueden Nadar.

Java

// Interfaz: Volador  
interface Volador {  
 void despegar();  
 void aterrizar();  
 void volar();  
}  
  
// Interfaz: Nadador  
interface Nadador {  
 void sumergirse();  
 void emerger();  
}  
  
// Un Pajaro hereda de Animal e implementa Volador  
class Pajaro extends Animal implements Volador {  
 public Pajaro(String nombre) { super(nombre); }  
 @Override public void despegar() { System.out.println(nombre + " bate sus alas y despega."); }  
 @Override public void aterrizar() { System.out.println(nombre + " aterriza."); }  
 @Override public void volar() { System.out.println(nombre + " está volando."); }  
}  
  
// Un Pato hereda de Animal e implementa Volador Y Nadador  
class Pato extends Animal implements Volador, Nadador {  
 public Pato(String nombre) { super(nombre); }  
 @Override public void despegar() { System.out.println(nombre + " despega del agua."); }  
 @Override public void aterrizar() { System.out.println(nombre + " aterriza en el agua."); }  
 @Override public void volar() { System.out.println(nombre + " está volando bajo."); }  
 @Override public void sumergirse() { System.out.println(nombre + " se sumerge."); }  
 @Override public void emerger() { System.out.println(nombre + " emerge."); }  
}  
  
public class DemostracionInterfaces {  
 public static void main(String[] args) {  
 Pajaro aguila = new Pajaro("Águila");  
 aguila.comer(); // De Animal  
 aguila.volar(); // De Volador  
  
 Pato donald = new Pato("Donald");  
 donald.comer(); // De Animal  
 donald.despegar(); // De Volador  
 donald.sumergirse(); // De Nadador  
  
 // Polimorfismo con interfaces:  
 Volador unVolador = new Pato("Ganso"); // Un Pato es un Volador  
 unVolador.volar();  
 // unVolador.sumergirse(); // ¡ERROR! No se puede llamar a un método de Nadador  
 // si la variable es de tipo Volador.  
 }  
}

**Explicación:** Pajaro y Pato implementan Volador, lo que les obliga a definir despegar(), aterrizar() y volar(). Pato también implementa Nadador, añadiendo esos comportamientos. El polimorfismo permite tratar a un Pato como un Volador o un Nadador según la necesidad.

## Puntos Clave para Recordar:

* **extends:** Para herencia de clases (solo una superclase).
* **implements:** Para implementar interfaces (múltiples interfaces).
* **super():** Llama al constructor de la superclase desde una subclase.
* **@Override:** Anotación para indicar la sobrescritura de un método.
* **Clases Abstractas (abstract class):** Plantillas que no pueden ser instanciadas; pueden tener métodos abstractos (sin cuerpo) y concretos.
* **Clases Finales (final class):** No pueden ser extendidas, lo que asegura la inmutabilidad de su diseño.
* **Interfaces (interface):** Definen contratos de comportamiento; las clases que las implementan deben cumplir ese contrato.