# Herencia

Semestre 2, 2025

## Usos

Reutilización de código: evita duplicación.

Modelado natural de jerarquías: ejemplo, **Animal → Perro → Pastor Alemán**.

Extensibilidad: se pueden crear nuevas clases a partir de otras ya existentes.

Permite **polimorfismo**.

La palabra **polimorfismo** proviene del griego poly (muchos) y morphé (formas). En Java, significa que un mismo objeto puede adoptar múltiples formas dependiendo del contexto de uso.

**Comparación con composición:**

- **Herencia:** modela relación “es-un” (is-a).
- **Composición:** modela relación “tiene-un” (has-a).

## Concepto

"¡Tan linda tu hija! es igualita a ti"

"Vos si no podes negar a tu hijo, es tu viva imágen"

Una clase (subclase) puede **heredar atributos y métodos** de otra (superclase).

Se establece con la palabra clave `extends`.

```

  class Persona {
```

```
3
4   String nombre;
5
6   public void saludar() {
7
8   System.out.println("Hola, soy " + nombre);
9
10  }
11
12  }
13
14
15
16  class Estudiante extends Persona {
17
18  String carnet;
19
20  }
21
```

Aquí, `Estudiante` hereda `nombre` y `saludar()` de `Persona`.

## Jerarquía de clases

En Java, todas las clases heredan directa o indirectamente de `Object`.

Esto significa que cualquier objeto en Java tiene disponibles métodos como:

- `toString()` – representación en texto.
- `equals()` – comparación de igualdad.
- `getClass()` – obtener el tipo en tiempo de ejecución.

Ejemplo: `Object → Persona → Estudiante`.

Proposito:

- Organizar las clases en **niveles lógicos**.
- Permitir que las clases compartan **atributos y comportamientos comunes**.

- Facilitar el **polimorfismo**, al tratar objetos diferentes bajo un mismo tipo general.

## Acceso a miembros heredados

Una subclase **hereda** los miembros `public` y `protected`.

Los miembros `private` **no son heredados**, pero sí accesibles a través de getters/setters.

```

class Padre {

private int secreto;

protected int protegido;

public int publico;

}


class Hijo extends Padre {

public void mostrar() {

// secreto; // no accesible

System.out.println(protegido); // accesible

System.out.println(publico); // accesible

}

}

```

## Override

La **sobrescritura de métodos** ocurre cuando una subclase redefine el comportamiento de un método heredado de la superclase.

El método mantinen la **misma firma** (nombre, parámetros y tipo de retorno compatible).

Se utiliza para **personalizar o especializar** el comportamiento heredado.

La anotación `@Override` no es obligatoria, pero ayuda al compilador a detectar errores.

La resolución del método ocurre en **tiempo de ejecución** (polimorfismo dinámico).

No se pueden sobrescribir métodos `final`, `static` o `private`.

```

class Persona {

public void saludar() {

System.out.println("Hola");

}

}


class Estudiante extends Persona {

@Override

public void saludar() {

System.out.println("Hola, soy estudiante");

}

}

```

# Super

La palabra clave `super` se utiliza en herencia para referirse explícitamente a la superclase.

Permite invocar al **constructor de la superclase** desde la subclase.

Permite acceder a **métodos o atributos** de la superclase que han sido sobrescritos o están ocultos.

Útil para extender comportamientos heredados sin perder la lógica original.

```
1
2   class Persona {
3
4   public void saludar() {
5
6   System.out.println("Hola");
7
8   }
9
10  }
11
12
13
14  class Estudiante extends Persona {
15
16  @Override
17
18  public void saludar() {
19
20  super.saludar(); // invoca método de superclase
21
22  System.out.println("... y también estudiante");
23
24  }
25
26  }
27
28
29
30  class Profesor extends Persona {
31
32  public Profesor() {
```

```
33
34  super(); // invoca al constructor de Persona
35
36  }
37
38  }
39
```

## Polimorfismo

El **polimorfismo** es la capacidad de un objeto de adoptar múltiples formas.

Esto permite que el mismo método tenga diferentes comportamientos según el tipo real del objeto en tiempo de ejecución.

Una referencia de la **superclase** puede apuntar a un objeto de cualquier **subclase**.

Permite escribir código más **genérico y extensible**, trabajando con la superclase pero obteniendo el comportamiento de la subclase.

```
1
2   class Persona {
3
4   public void saludar() {
5
6   System.out.println("Hola");
7
8   }
9
10  }
11
12
13
14  class Estudiante extends Persona {
15
16  @Override
17
18  public void saludar() {
19
20  System.out.println("Hola, soy estudiante");
```

```
}
}


class Profesor extends Persona {
@Override
public void saludar() {
System.out.println("Buenos días, soy profesor");
}
}


public class Main {
public static void main(String[] args) {
Persona p1 = new Estudiante();
Persona p2 = new Profesor();


p1.saludar(); // Hola, soy estudiante
p2.saludar(); // Buenos días, soy profesor
}
}

```

## Upcasting y Downcasting

```

```

```
2   Persona p = new Estudiante(); // upcasting implícito
3
4
5
6   Estudiante e = (Estudiante) p; // downcasting explícito
7
```

- **Upcasting:** seguro, siempre válido.
- **Downcasting:** puede lanzar `ClassCastException` si la referencia no es del tipo esperado.

## Arreglos y polimorfismo

Una de las ventajas del polimorfismo es poder trabajar con colecciones de la superclase y almacenar objetos de distintas subclases. Esto facilita el manejo uniforme de diferentes tipos de objetos.

```
1
2   import java.util.ArrayList;
3
4
5
6   public class Main {
7
8   public static void main(String[] args) {
9
10  ArrayList<Persona> personas = new ArrayList<>();
11
12  personas.add(new Estudiante());
13
14  personas.add(new Profesor());
15
16  personas.add(new Estudiante());
17
18
19
20  for (Persona p : personas) {
21
22  // Cada objeto ejecuta su propia versión
23
24  p.saludar();
```

```
25
26  }
27
28  }
29
30  }
31
```

Solo un `ArrayList<Persona>` maneja objetos de diferentes subclases gracias al polimorfismo.

## Clases abstractas y herencia

Una clase abstracta **no puede instanciarse**.

Puede contener métodos abstractos (sin implementación).

Las subclases deben implementar los métodos abstractos.

Es una especie de contato. Yo (subclase) me comprometo a implementar los métodos que se definen en la clase abastracta.

```
1
2   abstract class Animal {
3
4   public abstract void hacerSonido();
5
6   }
7
8
9
10  class Perro extends Animal {
11
12  @Override
13
14  public void hacerSonido() {
15
16  System.out.println("Guau");
17
18  }
19
20  }
21
```

# Final

`final class` → no puede ser extendida.

`final` en un método → no puede ser sobreescrito.

```

final class Constante {

// no se puede extender

}


class Base {

public final void metodo() {}

}


class Derivada extends Base {

// @Override metodo() ❌ error

}

```

# Buenas prácticas

Usen herencia solo si existe relación is-a.

Prefieran composición si es un caso “tiene-un”.

Eviten jerarquías profundas y complejas.

Documenten las clases base para que sea claro qué debe heredarse.

# Limitaciones

No hay herencia múltiple de clases en Java.

Puede generar acoplamiento excesivo.

Difícil de mantener en jerarquías muy profundas.

**Alternativa:** interfaces y composición.