

Polimorfismo via Herencia

Programación Orientada a Objetos (CC2008) - 2026

Polimorfismo via Herencia

Semestre 02, 2026

Polimorfismo

El polimorfismo es la capacidad de un mismo método de comportarse de forma distinta según el tipo real del objeto que lo ejecuta.

El término proviene del griego poly, muchos, y morphé, formas. Una sola llamada en el código, muchas respuestas posibles.

En Java el mecanismo se conoce como invocación virtual de métodos. La máquina virtual ejecuta el método del objeto real, no el del tipo con que se declaró la variable.

Las tres piezas

El polimorfismo via herencia requiere tres elementos, y si falta uno el mecanismo no opera.

1. Una superclase que declare el método que todas las subclases responderán.

2. Subclases que sobrescriban ese método con `@Override` , conservando la misma firma.
3. Una referencia del tipo de la superclase apuntando al objeto de la subclase.

La tercera es la que se pasa por alto. Sin una variable, un parámetro o una colección declarada con el tipo general, no hay nada que resolver en tiempo de ejecución.

Enunciado

Se construirá un sistema de nómina para una empresa que paga a su personal bajo tres esquemas distintos. El sistema calcula el pago del mes de cada empleado, imprime su recibo y totaliza el costo de la nómina.

- El monto por pagar depende del esquema de contratación de cada empleado.
- El recorrido de la nómina es idéntico para todos, sin importar el esquema.

Esquemas de pago considerados:

- Asalariado: recibe un monto fijo cada mes, sin importar las horas.
- Por hora: cobra una tarifa por hora, y lo que pasa de 40 horas se paga al 150 por ciento.
- Comisionista: recibe un porcentaje de las ventas del mes.

Restricción de diseño: agregar un esquema nuevo no debe obligar a modificar el código que recorre la nómina.

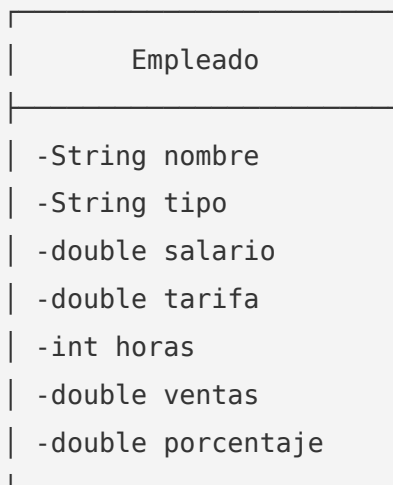
El diseño sin herencia

La salida inmediata es una sola clase con un campo que indica el esquema de cada empleado.

```

public class Empleado {
    private String nombre;
    private String tipo;          // asalariado...
    private double salario;
    private double tarifa;
    private int horas;
    private double ventas;
    private double porcentaje;
}

```



Cada objeto carga los campos de todos los esquemas y utiliza apenas dos o tres.

El cálculo del pago se convierte en una cadena de condicionales que consulta el esquema antes de decidir la fórmula.

```

// dentro de la clase Empleado
public double calcularPago() {
    if (this.tipo.equals("asalariado")) {
        return this.salario;
    } else if (this.tipo.equals("por_hora")) {
        return this.tarifa * this.horas;
    } else if (this.tipo.equals("comisionista")) {
        return this.ventas * this.porcentaje;
    }
}

```

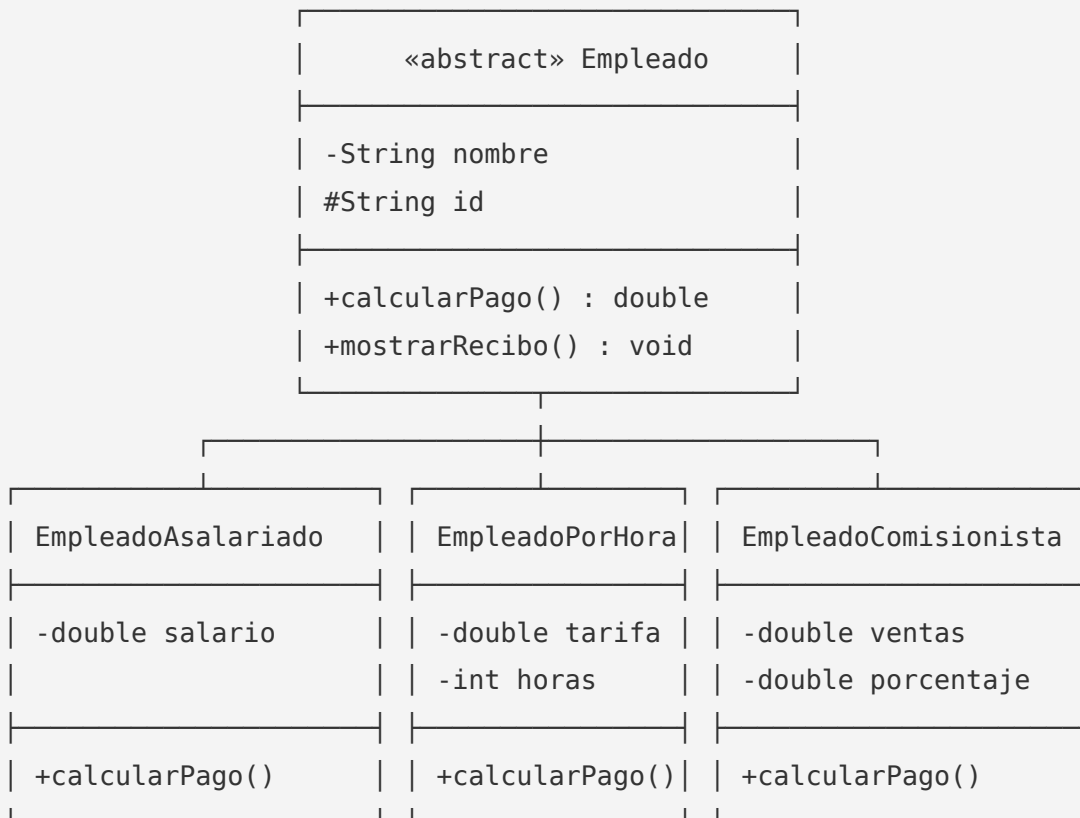
```
return 0;  
}
```

Lo que cuesta este diseño:

- Agregar un esquema obliga a modificar el método.
- El mismo `if` se repite en cada operación nueva, como el recibo, el reporte o el bono.
- Un esquema mal escrito compila igual y devuelve cero en producción.
- Nada obliga a un esquema nuevo a definir su fórmula.

La versión con polimorfismo traslada cada fórmula a la clase que le corresponde y elimina la cadena de condicionales.

La jerarquía de clases



Lectura del diagrama:

- El triángulo apunta a la superclase y se lee como una relación es-un.
- El signo numeral marca los atributos `protected`, visibles para la clase y sus subclases.
- La etiqueta `«abstract»` marca la clase que no se puede instanciar.
- El signo más marca los métodos públicos de la jerarquía.

`calcularPago` aparece en los cuatro recuadros con la misma firma, y la fórmula vive en la clase que le corresponde.

La superclase Empleado

```
public abstract class Empleado {  
    protected String nombre;  
    protected String id;  
  
    public Empleado(String nombre, String id) {  
        this.nombre = nombre;  
        this.id = id;  
    }  
  
    public abstract double calcularPago();  
  
    public void mostrarRecibo() {  
        System.out.println(this.id + " " + this.nombre  
            + " -> Q" + calcularPago());  
    }  
}
```

Decisiones de diseño:

- La clase es `abstract` porque un empleado sin esquema no existe en la empresa, de modo que `new Empleado(...)` no compila.

- `nombre` e `id` son `protected` para que las subclases los usen de forma directa, sin abrirlos al resto del programa.
- `calcularPago` es abstracto y obliga a cada subclase a definir su fórmula.
- `mostrarRecibo` se hereda tal cual y ninguna subclase necesita reescribirlo.

`mostrarRecibo` invoca a `calcularPago` sin conocer la versión que se ejecutará. Esa decisión la toma Java en tiempo de ejecución, según el objeto real.

EmpleadoAsalariado

```
public class EmpleadoAsalariado extends Empleado {
    protected double salario;

    public EmpleadoAsalariado(String nombre,
                               String id,
                               double salario) {
        super(nombre, id);
        this.salario = salario;
    }

    @Override
    public double calcularPago() {
        return this.salario;
    }
}
```

La subclase aporta su atributo propio en `protected`, delega los datos comunes con `super` y cumple el contrato con `@Override`. Sin ese método la clase no compila, porque el contrato queda pendiente.

EmpleadoPorHora

```
public class EmpleadoPorHora extends Empleado {
    protected double tarifa;
    protected int horas;

    public EmpleadoPorHora(String nombre,
                            String id,
                            double tarifa,
                            int horas) {
        super(nombre, id);
        this.tarifa = tarifa;
        this.horas = horas;
    }

    @Override
    public double calcularPago() {
        if (this.horas <= 40) {
            return this.tarifa * this.horas;
        }
        return this.tarifa * 40
            + this.tarifa * 1.5 * (this.horas - 40);
    }
}
```

La fórmula de este esquema es la más complicada de las tres, y nada de esa complejidad se asoma fuera de la clase. Desde afuera la llamada sigue siendo `calcularPago`.

`mostrarRecibo` no se modificó y ahora imprime un pago con horas extra incluidas.

EmpleadoComisionista

```
public class EmpleadoComisionista extends Empleado {
    protected double ventas;
    protected double porcentaje;
```

```

    public EmpleadoComisionista(String nombre,
                                String id,
                                double ventas,
                                double porcentaje) {

        super(nombre, id);
        this.ventas = ventas;
        this.porcentaje = porcentaje;
    }

    @Override
    public double calcularPago() {
        return this.ventas * this.porcentaje;
    }
}

```

La tercera subclase repite la estructura con otra fórmula. La superclase no cambió ni una línea para que las tres funcionen.

Tres clases, tres fórmulas, una sola firma de método. Eso es lo que hace posible el polimorfismo.

La nómina en acción

```

import java.util.ArrayList;

public class Nomina {
    public static void main(String[] args) {
        ArrayList<Empleado> empleados = new ArrayList<>();

        empleados.add(new EmpleadoAsalariado(
            "Ana", "E01", 9000));
        empleados.add(new EmpleadoPorHora(
            "Luis", "E02", 45, 46));
        empleados.add(new EmpleadoComisionista(
            "Sofía", "E03", 80000, 0.05));
    }
}

```

```
        double total = 0;
        for (Empleado e : empleados) {
            e.mostrarRecibo();
            total += e.calcularPago();
        }
        System.out.println("Total: Q" + total);
    }
}
```

La lista se declara `ArrayList<Empleado>` y almacena objetos de cualquier subclase. Ese cambio de tipo es automático y se denomina upcasting.

El `for` recorre la nómina sin consultar el esquema de cada empleado. No aparece un solo `if` en el código.

`e.calcularPago()` es una sola línea que ejecuta tres fórmulas distintas, una por cada objeto de la lista.

Salida:

```
E01 Ana -> Q9000.0
E02 Luis -> Q2205.0
E03 Sofía -> Q4000.0
Total: Q15205.0
```

El pago de Luis incluye 40 horas normales a Q45 más 6 horas extra a Q67.5, que dan Q2205.

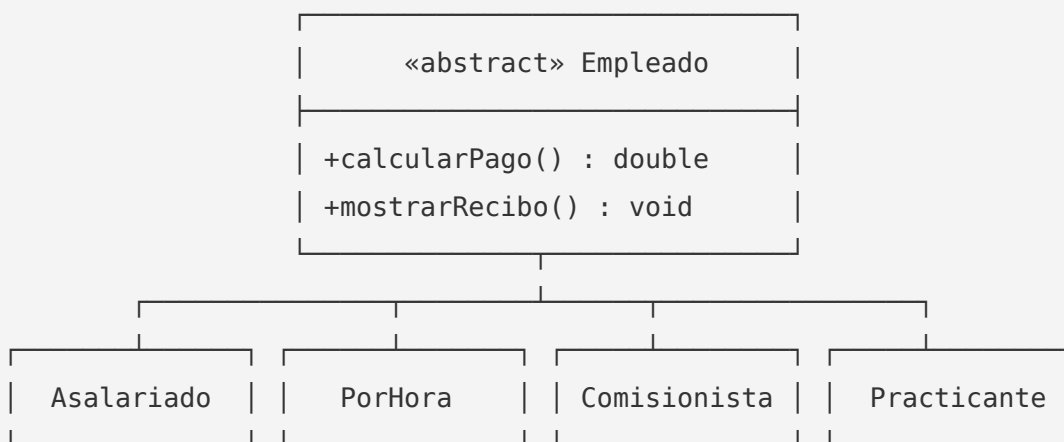
Antes y después

Aspecto	Con cadena de <code>if</code>	Con polimorfismo
Fórmula de pago	en un método central	en cada subclase

Aspecto	Con cadena de if	Con polimorfismo
Agregar un esquema	editar el método	crear una clase
Esquema mal escrito	compila y falla en ejecución	no compila
Campos por objeto	todos los del sistema	solo los propios
Fórmula obligatoria	nada la exige	el método abstracto la exige
Código del recorrido	crece con cada esquema	no cambia

Un requerimiento nuevo

La empresa empieza a contratar practicantes, que reciben una beca fija mensual. El sistema debe incluirlos en la nómina sin alterar el cálculo de los demás esquemas.



```
public class Practicante extends Empleado {
    protected double beca;

    public Practicante(String nombre, String id,
                        double beca) {
        super(nombre, id);
        this.beca = beca;
    }

    @Override
    public double calcularPago() {
        return this.beca;
    }
}
```

```
empleados.add(new Practicante("Diego", "E04", 2500));
```

El recorrido de la nómina no cambia ni una línea, y tampoco cambian `Empleado` ni las otras tres subclases.

Se agregó una clase nueva en lugar de modificar código que ya funcionaba. Esa es la ganancia práctica del polimorfismo.

Métodos que reciben la superclase

El polimorfismo no aparece solo en las listas, también en los parámetros de los métodos.

```
public static double totalNomina(
    ArrayList<Empleado> empleados) {
    double total = 0;
    for (Empleado e : empleados) {
        total += e.calcularPago();
    }
}
```

```
        return total;
    }

    public static Empleado mejorPagado(
        ArrayList<Empleado> empleados) {
        Empleado mejor = empleados.get(0);
        for (Empleado e : empleados) {
            if (e.calcularPago() > mejor.calcularPago()) {
                mejor = e;
            }
        }
        return mejor;
    }
}
```

Los dos métodos operan con cualquier subclase, incluso con las que todavía no existen.

La regla práctica es declarar parámetros, variables y colecciones con el tipo más general que sirva para el trabajo.

Tipo declarado y tipo real

Una variable tiene dos tipos que conviene distinguir.

- Tipo declarado: el que aparece en el código, y decide qué métodos se pueden invocar.
- Tipo real: el objeto que está en memoria, y decide qué versión del método se ejecuta.

```
Empleado e = new EmpleadoPorHora(
    "Luis", "E02", 45, 46);

e.calcularPago();      // corre la de EmpleadoPorHora
e.registrarHoras(8);   // no compila
```

Suponiendo que `EmpleadoPorHora` incorpore un método propio:

```
public void registrarHoras(int extra) {  
    this.horas += extra;  
}
```

El compilador solo ve el tipo declarado, así que rechaza `registrarHoras` aunque el objeto sí lo tenga. El método existe, pero no es visible a través de una referencia de tipo `Empleado`.

Upcasting y downcasting

```
// upcasting implícito  
Empleado e = new EmpleadoPorHora(  
    "Luis", "E02", 45, 46);  
  
// downcasting explícito  
EmpleadoPorHora h = (EmpleadoPorHora) e;  
h.registrarHoras(8);
```

- Upcasting: subir al tipo de la superclase, siempre seguro y automático.
- Downcasting: bajar al tipo de la subclase, se escribe de forma explícita y puede fallar.

```
Empleado e = new EmpleadoAsalariado(  
    "Ana", "E01", 9000);  
  
EmpleadoPorHora h = (EmpleadoPorHora) e;    // falla
```

El compilador acepta la conversión porque ambas clases pertenecen a la misma jerarquía. En ejecución el objeto no es un `EmpleadoPorHora` y se lanza una `ClassCastException`.

La forma segura es verificar el tipo con `instanceof`, que desde Java 16 declara la variable en la misma línea.

```
for (Empleado e : empleados) {  
    if (e instanceof EmpleadoPorHora h) {  
        h.registrarHoras(8);  
    }  
}
```

Para un caso puntual es aceptable. Si aparece una cadena de `instanceof` para decidir comportamiento, el diseño regresó al punto de partida y ese comportamiento pertenece a un método sobrescrito.

toString y la clase Object

Toda clase hereda de `Object`, así que `toString()` ya es un método sobrescribible.

```
@Override  
public String toString() {  
    return this.nombre + " (" + this.id + ")";  
}
```

```
System.out.println(empleados.get(0));    // Ana (E01)
```

`println` recibe un `Object` y desconoce el dominio de la nómina, y aun así termina ejecutando el `toString()` de la subclase. Es el mismo mecanismo de la lección, aplicado por la biblioteca estándar.

Lo que no es polimorfismo dinámico

Solo los métodos de instancia se resuelven según el objeto real. Hay tres casos que se parecen y no lo son.

- Sobrecarga: varios métodos con el mismo nombre y distintos parámetros.
- Métodos `static` : pertenecen a la clase y no al objeto.
- Atributos: el de la subclase oculta al de la superclase.

Los tres se deciden con el tipo declarado de la variable.

La sobrecarga se resuelve en compilación. Estos son dos métodos distintos de la misma clase, sin herencia ni sobrescritura de por medio, y lo único que los diferencia es el tipo del parámetro:

```
class Reporte {  
    public void imprimir(Empleado e) {  
        System.out.println("Recibo de empleado");  
    }  
  
    public void imprimir(EmpleadoPorHora e) {  
        System.out.println("Recibo con horas");  
    }  
}
```

```
Empleado e = new EmpleadoPorHora(  
    "Luis", "E02", 45, 46);  
  
Reporte r = new Reporte();  
r.imprimir(e);    // imprime "Recibo de empleado"
```

El objeto es un `EmpleadoPorHora` , pero la variable se declaró como `Empleado` . El compilador elige la versión con ese tipo declarado y el objeto real no participa en la decisión.

Aspecto	Sobrecarga	Sobrescritura
---------	------------	---------------

Dónde se escriben	en la misma clase	en la subclase
Firma del método	distintos parámetros	la misma firma
Quién decide	el compilador	la máquina virtual
Con qué tipo decide	el declarado	el real
Cuándo se decide	al compilar	al ejecutar

Los métodos `static` no se sobrescriben, se ocultan, y responden al tipo declarado:

```
class Base {
    public static String quien() { return "Base"; }
}

class Hija extends Base {
    public static String quien() { return "Hija"; }
}

Base b = new Hija();
System.out.println(b.quien());    // imprime "Base"
```

Los atributos tampoco son polimórficos, también se ocultan. El objeto es uno solo y los dos atributos existen a la vez, y el que se lee depende del tipo con que se llega a él:

```
class Base {
    protected String etiqueta = "base";
}

class Hija extends Base {
    protected String etiqueta = "hija";
}
```

```
public void mostrar() {  
    Base comoBase = this;  
  
    System.out.println(this.etiqueta);    // hija  
    System.out.println(comoBase.etiqueta); // base  
}  
}
```

La regla corta es que el polimorfismo dinámico aplica a métodos de instancia sobrescritos, y a nada más.

Errores comunes

Modificar la firma sin advertirlo. `calcularPago(double bono)` no sobrescribe a `calcularPago()`, es un método nuevo. La anotación `@Override` convierte ese error silencioso en error de compilación.

Invocar un método de la subclase con una referencia de la superclase. El objeto lo tiene, pero el compilador no lo ve.

Aplicar downcasting sin verificar con `instanceof`, y encontrar la `ClassCastException` en producción.

Conservar una cadena de `instanceof` dentro del recorrido. Ese comportamiento pertenece a un método sobrescrito en cada subclase.

Intentar sobrescribir métodos `private`, `static` o `final`. El primero no se hereda, el segundo se oculta y el tercero está cerrado de forma deliberada.

Invocar un método sobrescribible desde el constructor de la superclase. La versión de la subclase se ejecuta antes de inicializar sus atributos, de modo que encuentra ceros y `null`.

Buenas prácticas

Declarar variables, parámetros y colecciones con el tipo más general que sirva.

Ubicar en la superclase lo que todos comparten y en cada subclase solo lo propio.

Usar `abstract` cuando la superclase no tenga una implementación razonable por defecto.

Escribir siempre `@Override`, porque no tiene costo y detecta errores de firma.

Sospechar de un `if` que pregunta por el tipo, casi siempre indica un método sobrescrito que falta.

Práctica

Sobre el sistema de nómina construido en clase, se solicita resolver los siguientes puntos sin modificar el recorrido de la nómina.

1. Implementar `EmpleadoPorProyecto`, que cobra un monto fijo por cada proyecto entregado.
2. Sobrescribir `mostrarRecibo()` en `EmpleadoComisionista` para que también imprima las ventas del mes, reutilizando `super.mostrarRecibo()`.
3. Escribir un método que reciba la lista y devuelva el costo de la nómina si todo el personal recibe un bono del 10 por ciento.
4. Predecir la salida de los tres ejemplos de sobrecarga, static y atributos, ejecutarlos y explicar la diferencia.
5. Dibujar el diagrama de clases resultante, con los atributos y los métodos de cada esquema de pago.