Concepto Polimorfismo

🧩 Polimorfismo en Java

El **polimorfismo** es uno de los principios fundamentales de la **Programación Orientada a Objetos (POO)**.

Su nombre proviene del griego *poli* (muchos) y *morphos* (formas), lo que significa **“muchas formas”**.

En Java, el polimorfismo permite que un **mismo método o referencia** adopte **diferentes comportamientos**, dependiendo del **objeto real** que lo esté utilizando.

🔷 Concepto general

El polimorfismo en Java permite que una **referencia de tipo padre** pueda **apuntar a objetos de diferentes clases hijas**, y que al invocar un método, se ejecute el que corresponda al **tipo real del objeto**.

Esto significa que el **tipo de referencia** puede ser diferente del **tipo del objeto**.

La decisión de **qué método ejecutar** se toma **en tiempo de ejecución**, lo que hace al programa más **flexible y dinámico**.

⚙️ Ejemplo práctico

```
// Clase padre
class Animal {
    public void sonido() {
        System.out.println("El animal hace un sonido");
    }
}

// Clase hija
class Perro extends Animal {
    @Override
    public void sonido() {
        System.out.println("El perro ladra 🐶");
    }
}

// Otra clase hija
class Gato extends Animal {
    @Override
    public void sonido() {
        System.out.println("El gato maúlla 🐱");
    }
}

// Clase principal
public class Main {
    public static void main(String[] args) {
        // Polimorfismo: la referencia es de tipo Animal
        Animal firulais = new Perro();
        Animal michi = new Gato();

        // Se ejecuta el método según el tipo real del objeto
        firulais.sonido(); // El perro ladra 🐶
        michi.sonido();    // El gato maúlla 🐱
    }
```

```
}
```

🧠 Explicación paso a paso

1. **La clase** `Animal` define un método general llamado `sonido()`.
2. **Las clases** `Perro` **y** `Gato` heredan de `Animal` y **sobrescriben** el método `sonido()` con su propio comportamiento.
3. Cuando se declara `Animal firulais = new Perro();`,
   - El tipo de referencia (`Animal`) determina qué métodos **pueden llamarse**.
   - El tipo real (`Perro`) determina **qué implementación se ejecuta**.
4. Por eso, aunque `firulais` es de tipo `Animal`, al ejecutar `firulais.sonido()`,
   se ejecuta el método sobrescrito de `Perro`, no el de `Animal`.

🧬 Relación con la herencia

El polimorfismo **depende directamente de la herencia**.

Gracias a ella, las subclases (`Perro`, `Gato`, etc.) **heredan los métodos** del padre (`Animal`),

pero pueden **cambiar su comportamiento** según sus propias necesidades.

Esto permite escribir código que **trabaja con el tipo más general**, pero **se comporta según el tipo específico**.

Ejemplo:

```
Animal animal = new Perro(); // Polimórfico
animal.sonido(); // Ejecuta el método del Perro
```

Así, un mismo método puede producir **diferentes resultados** según el tipo de objeto que lo invoque.

🚫 Importante recordar

- El polimorfismo **solo aplica a métodos**, no a atributos.
- El **tipo de referencia** define qué métodos están disponibles.
- El **tipo del objeto real** define cuál implementación se ejecuta.
- Solo es posible aplicar polimorfismo **si existe una relación de herencia o una interfaz común**.

💡 En resumen

> El polimorfismo en Java permite que un objeto adopte diferentes comportamientos según el tipo real que lo representa.
>
> Es una forma de escribir código flexible y reutilizable, capaz de adaptarse sin necesidad de conocer las clases concretas que se están usando.

🧩 IMPORTANTE: **El tipo de referencia define qué métodos puedes usar**

Cuando escribes:

```
Animal firulais = new Perro();
```

tienes **dos tipos** en juego:

- **Tipo de referencia:** `Animal`
- **Tipo real del objeto:** `Perro`

👉 El **tipo de referencia (`Animal`)** determina **qué métodos son visibles y pueden ser invocados**.

👉 El **tipo real (`Perro`)** determina **qué implementación se ejecuta** de esos métodos (si fueron sobrescritos).

🐶 Ejemplo práctico

```
class Animal {
    public void sonido() {
        System.out.println("Sonido genérico de animal");
    }
}

class Perro extends Animal {
    @Override
    public void sonido() {
        System.out.println("El perro ladra 🐶");
    }

    public void ladrar() {
        System.out.println("Guau Guau!");
    }
}
```

Ahora mira esto 👇

```
Animal firulais = new Perro();

firulais.sonido(); // ✅ válido → método existe en Animal (y se ejecuta el de Perro)
firulais.ladrar(); // ❌ error → método no existe en Animal
```

💡 ¿Por qué da error `firulais.ladrar()`?

Porque **el compilador** solo ve el tipo de la referencia (`Animal`).

En la clase `Animal` **no existe** un método llamado `ladrar()`,

así que **no permite llamarlo**, incluso si el objeto real (`Perro`) sí lo tiene.

🧠 ¿Y si quiero acceder a `ladrar()`?

Debes **hacer un casting** (conversión explícita) al tipo correcto:

```
Animal firulais = new Perro();

// Conversión explícita a Perro
((Perro) firulais).ladrar(); // ✅ Ahora sí, porque lo tratamos como Perro
```

⚠️ Pero debes tener cuidado:

el **objeto real** debe ser efectivamente un `Perro`.

Si fuera otro tipo (por ejemplo `new Gato()`), el programa lanzará un `ClassCastException` en tiempo de ejecución.

📘 5. En resumen

Aspecto	Tipo de referencia	Tipo real
Define qué métodos **puedes llamar**	✅	❌
Define **qué versión** del método se ejecuta	❌	✅
Se usa en **tiempo de compilación**	✅	❌
Se usa en **tiempo de ejecución**	❌	✅

🧩 Polimorfismo con estructuras de datos y métodos específicos

El polimorfismo te permite **tratar diferentes objetos como si fueran del mismo tipo**, pero en ocasiones necesitas acceder a **comportamientos particulares** de cada uno.

Ahí entra en juego el **casting** y la comprobación con `instanceof`.

⚙️ Ejemplo completo

```
import java.util.ArrayList;
import java.util.List;

// Clase base abstracta
abstract class Animal {
    public abstract void sonido(); // comportamiento general
}

// Subclase Perro
class Perro extends Animal {
    @Override
    public void sonido() {
        System.out.println("El perro ladra 🐶");
    }

    public void ladrar() {
        System.out.println("Guau Guau! (método específico de Perro)");
    }
}

// Subclase Gato
class Gato extends Animal {
    @Override
    public void sonido() {
        System.out.println("El gato maúlla 🐱");
    }

    public void ronronear() {
        System.out.println("Prrrr... (método específico de Gato)");
    }
}

// Subclase Vaca
class Vaca extends Animal {
    @Override
    public void sonido() {
        System.out.println("La vaca muge 🐮");
    }

    public void darLeche() {
        System.out.println("La vaca está dando leche 🥛");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        // Lista polimórfica de tipo Animal
        List<Animal> animales = new ArrayList<>();

        animales.add(new Perro());
        animales.add(new Gato());
        animales.add(new Vaca());

        // Recorremos la lista con polimorfismo
        for (Animal a : animales) {
            a.sonido(); // Método común (polimórfico)

            // Ahora verificamos el tipo real del objeto
            if (a instanceof Perro) {
                ((Perro) a).ladrar(); // Acceso a método específico
            } else if (a instanceof Gato) {
                ((Gato) a).ronronear();
            } else if (a instanceof Vaca) {
                ((Vaca) a).darLeche();
            }

            System.out.println("------------------");
        }
    }
}
```

🧠 Explicación paso a paso

1. **La lista** `List<Animal>` almacena objetos de distintos tipos (`Perro`, `Gato`, `Vaca`), gracias al **polimorfismo**.
2. Con `a.sonido()`, se ejecuta el método **sobrescrito** en cada clase según su tipo real.
3. Sin embargo, los métodos **específicos** como `ladrar()`, `ronronear()` o `darLeche()` no existen en `Animal`, por lo tanto, no se pueden llamar directamente.
4. Para acceder a ellos:
   - Se usa `instanceof` para **verificar el tipo real del objeto**.
   - Luego se hace un **casting** al tipo correspondiente, para poder invocar sus métodos particulares.

💬 Salida esperada en consola

```
El perro ladra 🐶
Guau Guau! (método específico de Perro)
------------------
El gato maúlla 🐱
Prrrr... (método específico de Gato)
------------------
La vaca muge 🐮
La vaca está dando leche 🥛
------------------
```

⚠️ Importante recordar

- El polimorfismo te permite **tratar diferentes tipos de objetos de forma general**.
- El **casting** y `instanceof` se usan **solo cuando necesitas comportamientos específicos** del tipo concreto.

- Sin `instanceof`, intentar hacer un casting incorrecto (por ejemplo `((Perro) a)` cuando `a` es un `Gato`) provocará un `ClassCastException` en tiempo de ejecución.

💡 En resumen

> Con polimorfismo, una misma lista puede almacenar distintos tipos de objetos que comparten una clase base.
>
> Usando `instanceof` y casting, puedes acceder a métodos exclusivos de cada clase cuando lo necesites,
>
> sin perder la flexibilidad que ofrece tratar todo como un solo tipo (`Animal`).

🧩 Polimorfismo como ventaja al enviar información entre métodos

El **polimorfismo** permite que un método reciba como parámetro un **tipo genérico (padre o interfaz)** y acepte objetos de **cualquiera de sus subclases**.

Esto elimina la necesidad de **sobrecargar** el mismo método para cada tipo concreto.

🔷 Sin polimorfismo (forma limitada y repetitiva)

Imagina que queremos un método que haga que los animales emitan su sonido.

Sin polimorfismo, tendríamos que crear un método por cada tipo:

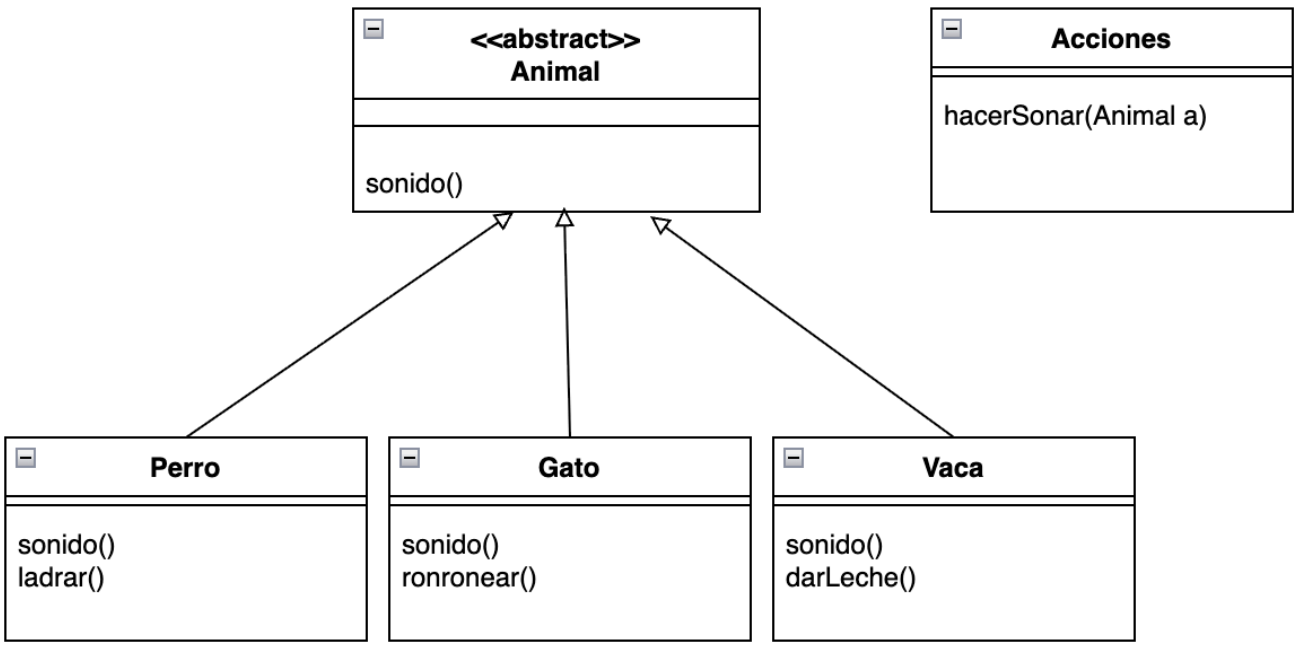
```
class Acciones {
    public void hacerSonar(Perro perro) {
        perro.sonido();
    }

    public void hacerSonar(Gato gato) {
        gato.sonido();
    }

    public void hacerSonar(Vaca vaca) {
        vaca.sonido();
    }
}
```

👉 Si mañana agregas una nueva clase (`Caballo`, `Pajaro`, etc.), tendrías que **crear otro método más**, lo cual es poco escalable.

🔷 Con polimorfismo (forma flexible y profesional)

Gracias al polimorfismo, podemos escribir **un solo método** que reciba el tipo más general (`Animal`)

y funcione para todos los tipos concretos:




```
// Clase base
abstract class Animal {
    public abstract void sonido();
}

// Subclases
class Perro extends Animal {
    @Override
    public void sonido() {
        System.out.println("El perro ladra 🐶");
    }

    public void ladrar() {
        System.out.println("Guau Guau!");
    }
}

class Gato extends Animal {
    @Override
    public void sonido() {
        System.out.println("El gato maúlla 🐱");
    }

    public void ronronear() {
        System.out.println("Prrrrr...");
    }
}

class Vaca extends Animal {
    @Override
    public void sonido() {
        System.out.println("La vaca muge 🐮");
    }

    public void darLeche() {
        System.out.println("La vaca está dando leche 🥛");
    }
}

// Clase con método polimórfico mejorado
class Acciones {
```

```
    public void hacerSonar(Animal animal) {
        // Se ejecuta el método polimórfico
        animal.sonido();

        // Comprobación de tipo real del objeto
        if (animal instanceof Perro) {
            ((Perro) animal).ladrar();
        } else if (animal instanceof Gato) {
            ((Gato) animal).ronronear();
        } else if (animal instanceof Vaca) {
            ((Vaca) animal).darLeche();
        }

        System.out.println("----------------------");
    }
}

// Clase principal
public class Main {
    public static void main(String[] args) {
        Acciones acciones = new Acciones();

        Animal perro = new Perro();
        Animal gato = new Gato();
        Animal vaca = new Vaca();

        acciones.hacerSonar(perro);
        acciones.hacerSonar(gato);
        acciones.hacerSonar(vaca);
    }
}
```

🧠 Explicación

- El método `hacerSonar(Animal animal)` recibe el tipo **general** `Animal`.
- Gracias al polimorfismo, puede recibir **cualquier objeto que herede de** `Animal` (`Perro`, `Gato`, `Vaca`, etc.).
- Cuando el método llama a `animal.sonido()`, Java ejecuta **la versión correspondiente a la subclase real**, no la del padre.
- El **casting** (`(Perro) animal`) habilita el acceso a los **métodos específicos** de cada clase.

💬 Salida esperada

```
El perro ladra 🐶
Guau Guau!
----------------------
El gato maúlla 🐱
Prrrrr...
----------------------
La vaca muge 🐮
La vaca está dando leche 🥛
----------------------
```

⚙️ Ventajas clave

Sin polimorfismo	Con polimorfismo
Repetición de métodos para cada tipo	Un solo método genérico
Código rígido y poco mantenible	Código flexible y escalable
Requiere cambios cada vez que se añade una clase nueva	El método sigue funcionando sin cambios
No aprovecha herencia ni abstracción	Total compatibilidad con herencia

💡 En resumen

> El polimorfismo permite enviar diferentes tipos de objetos a un mismo método, siempre que compartan una superclase o interfaz.
>
> Así, un único método puede manejar múltiples comportamientos sin necesidad de sobrecargarlo,
>
> haciendo el código **más limpio, flexible y extensible**.

Instructor: Cristian David Henao Hoyos