educación IT
CODE YOUR FUTURE

Java Standard Web Programming

Módulo 3

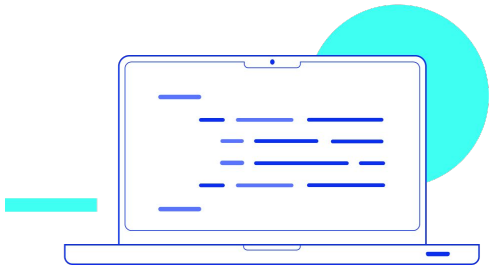
Polimorfismo

Introducción a Polimorfismo

El concepto científico de *polimorfismo* se refiere a aquello que puede adoptar múltiples formas. En POO (programación orientada a objetos), a la **posibilidad de definir clases diferentes en la misma jerarquía de herencia, que tienen métodos o atributos denominados de forma idéntica (sobreescritura de miembros), pero que se comportan de manera distinta**.

No podemos decir que haya un pilar *(Encapsulamiento, Herencia, Abstracción y Polimorfismo)* que sea más importante que otro, pero sí podemos decir que hay conceptos más simples que otros.

El polimorfismo es uno de los más importantes porque **simplifica la codificación** y resuelve muchas casuísticas a la hora de solucionar un determinado problema.



Polimorfismo por asignación

Un mismo objeto puede hacer referencia a más de un tipo de clase. El conjunto de las que pueden ser referenciadas está restringido por la herencia o la implementación (Interfaces).

```
Auto autoCarga = new Carga("Blanco", "Mercedez", new Patente("ARG-32165", true), 2, "321D65463DDD", "GRUA", 9.7F, 8);
```

Polimorfismo por sobrecarga

Recordemos este tema que ya vimos en la parte de métodos. Se pueden **codificar varios métodos bajo un mismo nombre con diferentes parámetros**. Eso indica que, al momento de invocar un método, el JDK entiende a qué método llamar. Pero esto parece tener poco sentido cuando vemos que los métodos tienen cantidad de parámetros distintos o de diferente tipo.

Ejemplo

Imaginemos dos métodos **`sumar`** con la misma cantidad de parámetros y de tipo numérico. En ambos casos son enteros pero a la hora de invocarlos el JVM interpreta correctamente a cuál llamar. Veamos el código del siguiente slide.

```java
public static void sumar(int a, int b) {
    System.out.println("El resultado de los enteros es: " + (a + b));
}

public static void sumar(long a, long b) {
    System.out.println("El resultado de los enteros largos es: " + (a + b));
}

public static void main(String[] args) {
    sumar(1, 2);
    sumar(8l, 28l);
}
```

Polimorfismo por Sobreescritura (Con Redefinición)

Esto ocurre cuando un objeto puede ser declarado de una clase padre, pero instanciado como una subclase o clase hija y se hace referencia a un ***método sobreescrito***.

```
Auto autoCarga;

autoCarga = new Carga("Blanco", "Mercedez", new Patente("ARG-32165", true), 2, "321D65463DDD", "GRUA", 9.7F, 8);

autoCarga.lavar(new Date(), MantenimientoPeriodico.LAVADO_PRESION);

autoCarga = new Familiar("Negro Mate", "FIAT", new Patente("ARG-86132", true), 4, "Compacto");

autoCarga.lavar(new Date(), MantenimientoPeriodico.LAVADO_TUNEL);
```

Debemos tomar en cuenta que un objeto instanciado como un hijo solo puede invocar los métodos que ambos posean. Aquí es donde más toma importancia la **abstracción** asegurando que **nuestros objetos tengan sentido con el polimorfismo**.

Polimorfismo sin Redefinición

Esto ocurre cuando **en diferentes clases se implementa el mismo método pero el comportamiento es totalmente distinto** y no está en la jerarquía de herencia.

Ejemplo

Podemos observar que en nuestra concesionaria se pueden vender **`Autos`** y **`Patentes`**. El método posee la misma firma pero son dos ventas de índole distinta.

```
public final class Patente {
```

```
public void vender() {
    System.out.println("Patente vendida (" + this + ")");
}
```

```
public class Familiar extends Auto {
```

```
@Override
public void vender() {
    // algoritmo para vender el auto
    System.out.println("Familiar vendido (" + this + ")");
}
```

Polimorfismo y una de sus principales ventajas

En muchas oportunidades, hemos utilizado polimorfismo sin darnos cuenta. Por ejemplo: al usar el método de impresión por consola de Java (cuando enviamos un objeto propio y este termina llamando al método `toString` del objeto enviado). ***¿Cómo es posible esto, si Java no tiene en sus librerías las clases que yo desarrollé?***

Ejemplo

Si se observa el método **`println`** de la clase **`PrintStream`** no tiene un parámetro **`Auto`**, **`Patente`**, ni otro personalizado.

Posee un parámetro que recibe un **`Object`** que al ser el padre de todo, puede convertirse en cualquiera de sus hijos incluyendo las clases propias de nuestro sistema.

```
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

Casteo de objetos

Como en los casteos vistos antes, se puede convertir un tipo de objeto en otro, siempre y cuando pertenezcan a la misma jerarquía de herencia. En este caso, vemos que declaramos un objeto de la Clase **`Auto`** pero lo instanciamos como **`Familiar`**.

De esta forma no podemos alcanzar los métodos ni atributos de la clase **`Familiar`** que no estén en la clase **`Auto`**, por lo que necesitaríamos convertirlo en un objeto **`Familia`**r. En este caso, **vemos el casteo de forma *explícita***.

```
Auto autoCarga;

autoCarga = new Familiar("Negro Mate", "FIAT", new Patente("ARG-86132", true), 4, "Compacto");

Familiar autoFamiliar3 = (Familiar) autoCarga;
```

¡Sigamos trabajando!