

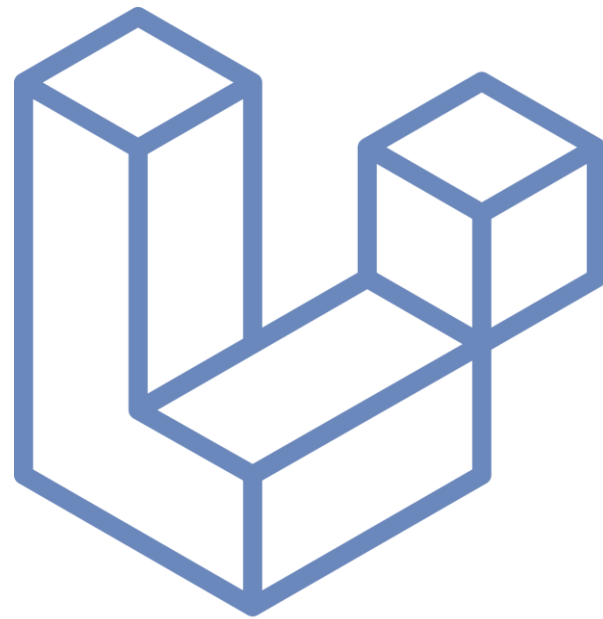
Introducción

El polimorfismo desde el concepto científico se refiere a aquello que puede adoptar múltiples formas.

En POO se refiere a la posibilidad de definir clases diferentes en la misma jerarquía de herencia que tienen métodos o atributos denominados de forma idéntica (sobreescripción de miembros), pero que se comportan de manera distinta.

No podemos decir que hay un pilar (Encapsulamiento, Herencia, Abstracción y Polimorfismo) que sea mas importante que otro, pero si podemos decir que hay conceptos mas simples que otros.

El polimorfismo es uno de los mas importantes porque simplifica la codificación y resuelve muchas casuísticas a la hora de resolver un problema.



Polimorfismo por Asignación

En java, 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).

Declaración del Objeto de tipo Auto

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

Se instancia como de tipo Carga


Polimorfismo por Sobrecarga

Este tema ya lo vimos en la parte de métodos pero recordemos un poco.

Podemos codificar varios métodos bajo un mismo nombre con diferentes parámetros eso indica que al momento de invocar el métodos el JDK entiende a que método llamar, pero esto pareciera de poco sentido cuando vemos que los métodos tienen cantidad de parámetros distintos o de diferente tipo.

Ahora imaginemos que necesitamos un método sumar con la misma cantidad de parámetros y ambos de tipo numérico, en ambos casos son enteros pero a la hora de invocarlos el JVM interpreta correctamente a cual de sus métodos llamar.

```
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(81, 281);  
}
```



Polimorfismo por Sobreescritura (Con Redefinición)

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

```
Auto autoCarga;
```

```
autoCarga = new Carga("Blanco", "Mercedes", 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 mas 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 esta en la jerarquía de herencia.

Acá 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

Hemos utilizado polimorfismo sin darnos cuenta cuando utilizamos el método de impresión por consola de Java cuando mandamos un objeto propio y este termina llamando al método **toString** del objeto enviado.

¿y cómo es posible esto, si Java no tiene en sus librerías las clases que yo desarrolle?

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

Si vemos el método **println** de la clase **PrintStream** no posee un parámetro Auto, Patente, ni otro personalizado; posee un parámetro que recibe un **Object** y esté al ser el padre de todo en Java puede convertirse en cualquiera de sus hijos incluyendo las clases propias de nuestro sistema.

Casteo de Objetos

Como en los casteos vistos anteriormente nosotros podemos convertir un tipo de objeto en otro siempre y cuando pertenezcan a la misma jerarquía de herencia.

```
Auto autoCarga;  
  
autoCarga = new Familiar("Negro Mate", "FIAT", new Patente("ARG-86132", true), 4, "Compacto");  
  
Familiar autoFamiliar3 = (Familiar) autoCarga;
```

En este caso vemos que declaramos un objeto de la Clase **Auto** pero lo instanciamos como **Familiar**, ahora bien, 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 Familiar.

En este caso vemos el casteo de forma explícita.