



CICLO 2

[FORMACIÓN POR CICLOS]

POLIMORFISMO



Ingeni@
Soluciones TIC



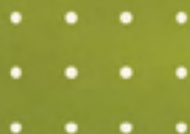
UNIVERSIDAD
DE ANTIOQUIA
Facultad de Ingeniería



Principios Básicos de la Programación Orientada a Objetos

Más allá de los elementos básicos de la programación orientada a objetos, existen cuatro principios básicos que constituyen los pilares de la programación orientada a objetos en Java, estos son: abstracción, encapsulamiento, herencia y polimorfismo. Algunos de ellos los hemos abordado en mayor o menor medida anteriormente:

- La **abstracción** consiste en aislar un elemento de su contexto (lo que lo rodea), así como simplificarlo. Esto, es lo que hacemos cuando definimos qué atributos, métodos y clases asociadas tendrá una clase que estamos diseñando, y qué otras cosas, definitivamente, no tendrá.
- El **encapsulamiento** consiste en el ocultamiento del estado de los objetos. Este principio propende por que los atributos de las clases sean privados y sus métodos sean públicos. Así, los atributos de las clases se deberían de poder acceder sólo mediante *getters* o *setters*, u otros métodos públicos ideados para accederlos.
- La **herencia**, como hemos visto, consiste en la posibilidad de que una clase sea creada a partir de otra ya existente, obteniendo mediante herencia métodos y atributos de dicha clase (su clase padre).
- Y finalmente, el cuarto principio se denomina polimorfismo.



Polimorfismo en Java

En la programación orientada a objetos, el **polimorfismo** permite que un objeto pueda responder de diferentes maneras a un mismo mensaje. Es decir, permite que, al llamar un método de un objeto dado, este funcione de manera diferente dependiendo de unas condiciones que veremos posteriormente.

En Java, el polimorfismo se da de dos formas, mediante ligadura estática y mediante ligadura dinámica. En ambos casos, se trata de que el programa “decida” qué instrucciones (o método) se ejecuten en un momento dado:

- En la **ligadura estática**, el código a ejecutar al llamar un método se especifica en tiempo de compilación, es decir, en el momento en que el código se compila antes de ejecutarse. La ligadura estática se da, por ejemplo, al presentarse sobrecarga de métodos: si una clase tiene varios métodos del mismo nombre y se llama uno de ellos, es en tiempo de compilación que el programa determina exactamente cuál método es el que se está llamando y, por lo tanto, que instrucciones se deben ejecutar.
- En la **ligadura dinámica**, el código a ejecutar al llamar un método se especifica en tiempo de ejecución. Es decir, una vez el programa ya ha sido compilado y está corriendo, como se espera. Normalmente, este tipo de polimorfismo se da al “variar” entre implementaciones de un mismo método en una jerarquía de clases, tal como veremos más abajo.

A continuación, vamos a explorar el concepto de polimorfismo por ligadura dinámica en Java mediante un ejemplo. En primer lugar, contamos con la clase `Bicicleta`, muy similar a la vista en lecturas previas¹.

¹El ejemplo completo se puede encontrar acá: https://github.com/leonjaramillo/udea_ruta2_ciclo2/tree/main/main/java/co/edu/udea/udea_ruta2_ciclo2/polimorfismo

```
package co.edu.udea.udea_ruta2_ciclo2.
polimorfismo;

public class Bicicleta {

    private String marca;
    private String color;
    private double velocidad;
    private String pedales;

    public Bicicleta() {
        this.marca = "GW";
        this.color = "gris";
        this.velocidad = 0;
        this.pedales = "Shimano";
    }

    public Bicicleta(String marca, String
color, double velocidad, String pedales) {
        this.marca = marca;
        this.color = color;
        this.velocidad = velocidad;
        this.pedales = pedales;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public double getVelocidad() {
        return velocidad;
    }

    public String getPedales() {
        return pedales;
    }
}
```

```

public void setPedales(String pedales) {
    this.pedales = pedales;
}

public void pedalear(double aceleracion) {
    this.velocidad = this.velocidad +
    aceleracion;
}

public void frenar() {
    if (this.velocidad > 0) {
        this.velocidad--;
    }
}

public String getDescripcion() {
    return "Soy una bicicleta de marca " +
    marca + ", con color " + color
        + ", a velocidad " + velocidad
    + " y con pedales " + pedales;
}

}

```

Luego, definimos dos clases hijas o subclases de Bicicleta.
Una es BicicletaMontania


```
package co.edu.udea.udea_ruta2_ciclo2.
polimorfismo;

public class BicicletaMontania extends
Bicicleta {

    private int cambio;

    public BicicletaMontania() {
        super();
        this.cambio = 6;
    }

    public BicicletaMontania(int cambio,
String marca, String color, double
velocidadInicial, String pedales) {
        super(marca, color, velocidadInicial,
pedales);
        if (cambio > 0 && cambio <= 12) {
            this.cambio = cambio;
        } else {
            this.cambio = 6;
        }
    }

    public void subirCambio() {
        if (this.cambio < 12) {
            cambio++;
        }
    }

    public void bajarCambio() {
        if (this.cambio > 1) {
            cambio--;
        }
    }

    public int getCambio() {
        return this.cambio;
    }
}
```

```

@Override
public void pedalear(double aceleracion) {
    double aceleracionConCambios =
aceleracion * (this.cambio / 6.0);
    super.pedalear(aceleracionConCambios);
}

@Override
public String getDescripcion() {
    return super.getDescripcion() + ", y
en el cambio " + cambio;
}
}

```

En la clase anterior, se le ponen cambios a la bicicleta y, evidentemente, cambia su descripción. La otra subclase es BicicletaRuta.

```

package co.edu.udea.udea_ruta2_ciclo2.
polimorfismo;

public class BicicletaRuta extends Bicicleta {

    private double anchoNeumatico;

    public BicicletaRuta() {
        super();
        this.anchoNeumatico = 28;
    }

    public BicicletaRuta(double
anchoNeumatico, String marca, String color,
double velocidadInicial, String pedales) {
        super(marca, color, velocidadInicial,
pedales);
        if (anchoNeumatico > 17 &&
anchoNeumatico <= 62) {
            this.anchoNeumatico =
anchoNeumatico;
        } else {
            this.anchoNeumatico = 28;
        }
    }

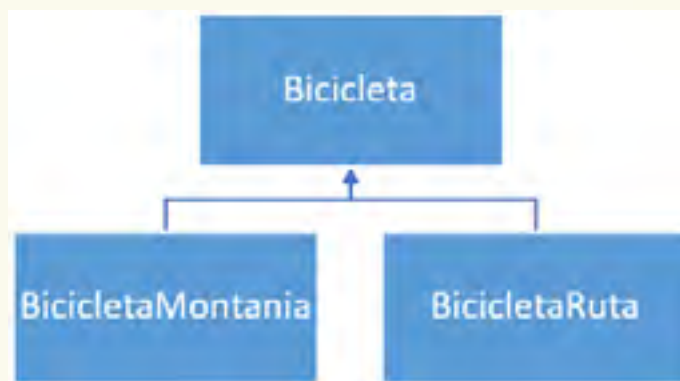
    public double getAnchoNeumatico() {
        return anchoNeumatico;
    }
}

```

```
public void setAnchoNeumatico(double
anchoNeumatico) {
    this.anchoNeumatico = anchoNeumatico;
}

@Override
public String getDescripcion() {
    return super.getDescripcion() + ", y
con ancho de neumático " + anchoNeumatico;
}
}
```

En la clase anterior, se está agregando a la clase Bicicleta el ancho del neumático, y también cambia la descripción. En ambos casos, se sobrescribe el método `getDescripcion`. Así pues, la jerarquía de clases en este caso, quedaría como podemos observar en el siguiente esquema.



Ahora, observemos el código a continuación.


```
package co.edu.udea.udea_ruta2_ciclo2.
polimorfismo;

public class PruebaBicicletas {
    public static void main(String[] args){
        Bicicleta cicla1, cicla2, cicla3;

        cicla1 = new Bicicleta();
        cicla2 = new BicicletaMontania(5,
"Trek", "Rojo", 0, "SRAM");
        cicla3 = new BicicletaRuta(32,
"Specialized", "Azul", 0, "Shimano");

        System.out.println(cicla1.
getDescription());
        System.out.println(cicla2.
getDescription());
        System.out.println(cicla3.
getDescription());
    }
}
```

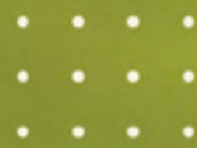
Notemos que, en la primera línea del método main, se declaran tres objetos de tipo `Bicicleta`. Luego, en las siguientes líneas, se instancian dichos objetos, pero no todos con la clase `Bicicleta`, como uno esperaría en un principio, sino con las clases `Bicicleta`, `BicicletaMontania` y `BicicletaRuta`, respectivamente. Estamos, entonces, instanciando objetos de la clase `Bicicleta` y de subclases suyas, y se las estamos asignando a objetos declarados como `Bicicleta`. Esto, dado que a un objeto declarado como de una clase le podemos asignar una instancia de su misma clase o de una de sus subclases. Finalmente, llamamos el método `getDescription` de cada uno de los objetos, debiendo obtener el resultado que se aprecia a continuación.



```
Soy una bicicleta de marca GW, con color gris,  
a velocidad 0.0 y con pedales Shimano  
Soy una bicicleta de marca Trek, con color  
Rojo, a velocidad 0.0 y con pedales SRAM, y en  
el cambio 5  
Soy una bicicleta de marca Specialized, con  
color Azul, a velocidad 0.0 y con pedales  
Shimano, y con ancho de neumático 32.0
```

Aquí podremos notar algo: En las tres líneas finales se llamó el mismo método (`getDescripcion`), pero se ejecutó siempre aquel correspondiente a la instancia llamada. Es decir, en el primer caso se ejecutó el método de la clase `Bicicleta`, en el segundo el de la clase `BicicletaMontania`, y en el tercer caso se ejecutó el método de la clase `BicicletaRuta`. Esta es una de las formas en las que se manifiesta el polimorfismo en Java.

A continuación, vamos a observar una nueva clase, la clase `Mecanico`. Por ahora, el `mecánico` sólo sabe poner pedales. Para esto, tendrá un conjunto de pedales (en un `ArrayList`), y si debe ponerle los pedales a una bicicleta, recibe dicha bicicleta, toma unos pedales de su lista y se los pone, avisando en pantalla.



```

package co.edu.udea.udea_ruta2_ciclo2.
polimorfismo;

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

public class Mecanico {

    private List<String> misPedales;

    public Mecanico() {
        misPedales = new ArrayList<>();
    }

    public List<String> getMisPedales() {
        return misPedales;
    }

    public void setMisPedales(List<String>
misPedales) {
        this.misPedales = misPedales;
    }

    public void ponerPedales(Bicicleta
unaCicla) {
        String pedalesAPoner = this.
misPedales.get(0);
        unaCicla.setPedales(pedalesAPoner);
        this.misPedales.remove(0);
        System.out.println("Pongo pedales de
marca " + pedalesAPoner +
                        " a una " + unaCicla.
getClass().getName());
    }
}

```

Es importante notar que el método `ponerPedales` de la clase anterior, recibe como parámetro un objeto de la clase `Bicicleta`. Esto, permite que el mecánico trabaje no solo con bicicletas, sino también con objetos de sus subclases, es decir, con bicicletas de montaña y bicicletas de ruta. Esta propiedad es una de las cuales nos permite sacar provecho del polimorfismo. A continuación, pondremos a trabajar el mecánico.

```
package co.edu.udea.udea_ruta2_ciclo2.
polimorfismo;

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

public class PruebaMecanico {

    public static void main(String[] args) {
        Bicicleta cicla01, cicla02, cicla03,
        cicla04;

        cicla01 = new Bicicleta();
        cicla02 = new BicicletaMontania(5,
"Trek", "Rojo", 0, "SRAM");
        cicla03 = new BicicletaRuta(32,
"Specialized", "Azul", 0, "Shimano");
        cicla04 = new BicicletaMontania(6,
"Scott", "Blanco", 0, "SRAM");

        List<Bicicleta> bicicletas = new
ArrayList<>();
        bicicletas.add(cicla01);
        bicicletas.add(cicla02);
        bicicletas.add(cicla03);
        bicicletas.add(cicla04);

        Mecanico juancho = new Mecanico();
        juancho.getMisPedales().
add("Pinarello");
        juancho.getMisPedales().add("Scott");
        juancho.getMisPedales().
add("Cannondale");
        juancho.getMisPedales().add("Giant");

        for (Bicicleta miCicla : bicicletas) {
            juancho.ponerPedales(miCicla);
        }

    }

}
```

El resultado del código anterior debería de ser el siguiente.

```
Pongo pedales de marca Pinarello a una
Bicicleta
Pongo pedales de marca Scott a una
BicicletaMontania
Pongo pedales de marca Cannondale a una
BicicletaRuta
Pongo pedales de marca Giant a una
BicicletaMontania
```

Hagamos un recuento de lo que sucedió:

1. Se declararon cuatro objetos de la clase `Bicicleta`.
2. Se instanciaron dichos objetos, pero no solo con la clase `Bicicleta`, sino también con sus subclases.
3. Se creó un `ArrayList` de bicicletas y se le agregaron los objetos creados en los dos pasos anteriores. Notemos que dicho `ArrayList` recibe bicicletas normales, de montaña y de ruta (otra manifestación del polimorfismo).
4. Luego, se creó un objeto de la clase `Mecanico`, y se le "dieron" algunos pedales.
5. Finalmente, el mecánico les puso pedales a todas las bicicletas. Para esto, iteramos sobre el `ArrayList` creado previamente, y para cada bicicleta se llamó el método `ponerPedales` del mecánico.

Si nos fijamos, el objeto `juancho` de la clase `Mecanico` le puso pedales no solo al objeto de la clase `Bicicleta` (que es explícitamente la clase de los objetos que recibe dicho método), sino también a objetos de sus clases hijas (`BicicletaMontania` y `BicicletaRuta`). Lo anterior, es un caso de uso que da cuenta de la flexibilidad que ofrece el polimorfismo en el lenguaje Java a la interacción entre diferentes objetos.