

Polimorfismo

Polimorfismo	1
¿Qué aprenderás?	2
Introducción	2
¿Qué es el polimorfismo?	3
Ejercicio Guiado: Academia	4
Casteo de clases	9



¡Comencemos!

¿Qué aprenderás?

- Comprender conceptos de polimorfismo para utilizar en listas.
- Reconocer cuándo realizar un casteo para transformar variables o clases.

Introducción

En el mundo de los programadores/as existen diversos conceptos que son claves para comprender códigos. Algunos de estos requieren un poco más de abstracción para su aprendizaje y posterior comprensión. Sin embargo, a lo largo de este capítulo veremos un elemento fundamental al interior de la Programación Orientada a Objetos (POO) y es el llamado “Polimorfismo”.

¡Vamos con todo!



¿Qué es el polimorfismo?

El concepto de polimorfismo en POO obtiene su nombre gracias al significado morfológico de la palabra.



Imagen 1. Origen morfológico de la palabra.
Fuente: Desafío Latam

Esto quiere decir que una instancia puede ser tratada como si fuese una de sus superclases y viceversa. Este concepto está fuertemente ligado con la herencia y es mucho más simple de lo que suena, a pesar de que parezca algo complicado. Veremos un ejemplo para entenderlo mejor.

Ejercicio Guiado: Academia

Paso 1: Tenemos una aplicación llamada `RegistroAsistencia`, con la cual buscamos registrar la asistencia a clases en una academia. Las personas de la academia están compuestas por profesores o estudiantes, por ende, vamos a crear la siguiente herencia:

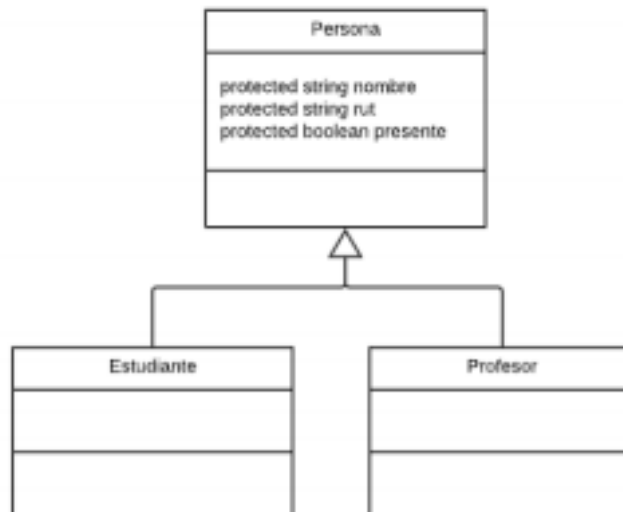


Imagen 2. Herencia de persona.
Fuente: Desafío Latam.

Paso 2: Creamos la clase `Persona`, `Estudiante` y `Profesor` dentro de un package llamado `Modelo`, tal como se muestra en la imagen a continuación:

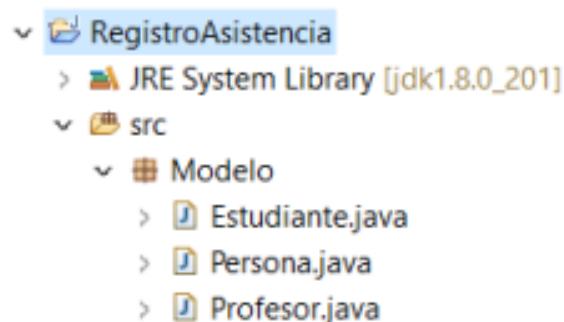


Imagen 3. Estructura del proyecto.
Fuente: Desafío Latam.

Paso 3: La clase `Persona` tendrá sus tres atributos encapsulados y el método `toString()`. Cabe destacar que la asistencia será: `presente` o `ausente`. Por lo tanto, será del tipo booleano.

```
package Modelo;

public class Persona {
    protected String rut;
    protected String nombre;
    protected boolean presente;
    public Persona(String rut, String nombre, boolean presente) {
super();
        this.rut = rut;
        this.nombre = nombre;
        this.presente = presente;
    }
    public String getRut() {
        return rut;
    }
    public void setRut(String rut) {
        this.rut = rut;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public boolean isPresente() {
        return presente;
    }
    public void setPresente(boolean presente) {
        this.presente = presente;
    }

    @Override
    public String toString() {
        return "Persona [rut=" + rut + ", nombre=" + nombre + ", presente="
+ presente + "]\n";
    }
}
```

Paso 4: Las otras dos clases deben extender a `Persona` para ser subclases de ella y, por consecuencia, deben sobrescribir el constructor de `Persona`. No realizaremos más que eso en `Estudiante` y `Profesor`.

```
//Extensión en la clase Profesor
public class Profesor extends Persona {

    public Profesor(String rut, String nombre, boolean presente) {
        super(rut, nombre, presente);
    }
}

//Extensión en la clase Estudiante
public class Estudiante extends Persona {
    public Estudiante(String rut, String nombre, boolean presente) {
        super(rut, nombre, presente);
    }
}
```

Paso 5: Ahora que tenemos la primera parte de la estructura, vamos a crear un `Main` como lo habíamos estado haciendo en la unidad anterior. Para ello vamos a crear una clase `Main`, dentro de un package llamado `Main` con un método llamado `main` (el uso de estos nombres es totalmente opcional, puedes usar el que quieras para el package y la clase).

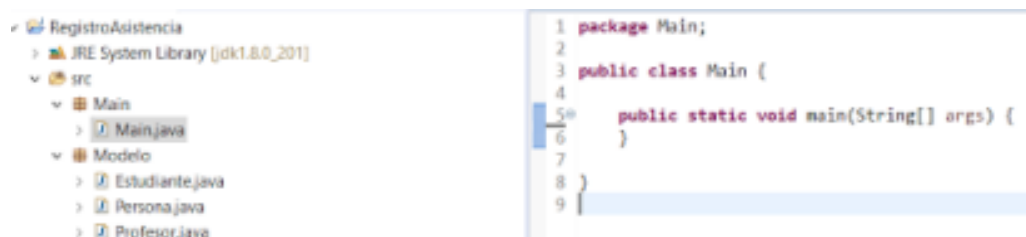


Imagen 4. Creando Main en el proyecto.
Fuente: Desafío Latam.

Paso 6: En el método main, vamos a hacer dos listas una de `Estudiante` y otra de `Profesor` para conocer quiénes están presentes durante una reunión de estudios.

```
package Main;
import java.util.ArrayList;

public class Main {

    public static void main(String[] args){

        ArrayList<Estudiante> listaEstudiantes = new ArrayList<>();
        ArrayList<Profesor> listaProfesores = new ArrayList<>();

        //Vamos a agregar individuos a las listas:
        listaEstudiantes.add(new Estudiante("1", "Juan", true));
        listaEstudiantes.add(new Estudiante("2", "Andrés", true));
        listaEstudiantes.add(new Estudiante("3", "Juan", false));
        listaProfesores.add(new Profesor("10", "Jose", true));

        for(Profesor profesor : listaProfesores) {
            System.out.println(profesor.toString());
        }
        for(Estudiante estudiante : listaEstudiantes) {
            System.out.println(estudiante.toString());
        }
    }
}
```

Paso 7: Le daremos `run` a nuestra aplicación y vamos a ver el siguiente resultado:

```
Persona [rut=10, nombre=Jose, presente=true]
Persona [rut=1, nombre=Juan, presente=true]
Persona [rut=2, nombre=Andres, presente=true]
Persona [rut=3, nombre=Juan, presente=false]
```

Esto está utilizando el método `toString()` heredado de la clase `Persona` y nos está mostrando los atributos que le dimos a los individuos. Vamos a ver cómo podemos utilizar el polimorfismo para crear una sola lista con todos estos individuos, haciendo el código más rápido en su ejecución y sin la necesidad de recorrer dos listas para imprimir todo.

Paso 8: Vamos a quitar las dos listas previas y crearemos una sola lista.

```
ArrayList<Persona> lista = new ArrayList<>();
```

Paso 9: Ahora, vamos a agregar a los individuos en esta lista.

```
lista.add(new Estudiante("1", "Juan", true));  
lista.add(new Estudiante("2", "Andrés", true));  
lista.add(new Estudiante("3", "Juan", false));  
lista.add(new Profesor("10", "Jose", true));
```

Paso 10: Recorrer la lista con un `for-each`, iterando con los objetos del tipo `Persona` e imprimiéndolos en pantalla.

```
for(Persona individuo : lista) {  
    System.out.println(individuo.toString());  
}
```

Paso 11: Clickeamos play y seguiremos teniendo el mismo resultado del código de antes.

```
Persona [rut=1, nombre=Juan, presente=true]  
Persona [rut=2, nombre=Andres, presente=true]  
Persona [rut=3, nombre=Juan, presente=false]  
Persona [rut=10, nombre=Jose, presente=true]
```

Esto es posible gracias al polimorfismo que dice que la instancia de una subclase (`Estudiante` y `Profesor`) puede ser tratada como si fuese una instancia de su superclase (`Persona`) durante la ejecución del programa. De esta forma es posible utilizar métodos que se compartan entre la superclase y la subclase, como lo es el método `toString()`. Pero si se utiliza un método que la instancia no tiene, arrojará una excepción de método no encontrado.

Para trabajar con este tipo de casos se puede utilizar un método para obtener la clase de una instancia, este método se hereda desde la clase `Object` y se llama `getClass()`.

Casteo de clases

El casteo es un procedimiento para transformar una variable primitiva de un tipo a otro, o para transformar objetos de una clase a otra, siempre y cuando haya una relación de herencia entre ambas.

Basado en el ejercicio anterior, vamos a imprimir el nombre de las clases de cada instancia dentro de la lista utilizando `getClass()`, esto nos retornará un tipo de dato llamado `class` que sirve para comparar clases, entre otras cosas. Para esto, le agregaremos el método `getSimpleName()` el cual convierte el nombre de la clase a un `String` y lo usaremos para imprimir en pantalla.

```
for(Persona individuo : lista) {  
    System.out.println(individuo.getClass().getSimpleName());  
}  
-----  
Impresión en pantalla:  
  
//Nos va a devolver la clase a la que pertenece cada elemento  
Estudiante  
Estudiante  
Estudiante  
Profesor
```

Otra forma en que podemos conocer la clase de una instancia es aplicar polimorfismo, ya que las subclases podrían tener métodos o atributos distintos.

Por ejemplo, el estudiante podría tener una deuda a pagar por estudios. Para ello agregamos ese atributo al interior de la clase como booleano.

```
public class Estudiante extends Persona {
    private double deuda;
    public Estudiante(double deuda, String rut, String nombre, boolean
presente) {
        super(rut, nombre, presente);
        this.deuda = deuda;
    }
    public Estudiante(String rut, String nombre, boolean presente) {
super(rut, nombre, presente);
    }
    public double getDeuda() {
        return deuda;
    }
    public void setDeuda(double deuda) {
        this.deuda = deuda;
    }
}
```

Entonces, modificamos un poco el método `main` y agregamos varios estudiantes con distintas deudas, además algunos profesores. Tal como se muestra a continuación:

```
public static void main(String[] args) {
    ArrayList<Persona> lista = new ArrayList<>();
    lista.add(new Estudiante(1500, "1", "Juan", true));
    lista.add(new Estudiante(2000, "2", "Andrés", true));
    lista.add(new Estudiante(3500, "3", "Juan", false));
    lista.add(new Profesor("10", "Jose", true));

    for(Persona individuo : lista) {
        System.out.println(individuo.getClass().getSimpleName());
    }
}
```

Vamos a imprimir la deuda de cada estudiante y el nombre de la clase con un `forEach`, pero para ello necesitaremos hacer algo previamente llamado "parseo de la instancia" o casteo. Todo esto con la finalidad de transformar la subclase `Estudiante` que es la que tiene el atributo `deuda`.

```
Estudiante estudiante = (Estudiante) instanciaPersona;
```

Lo que hacemos es intentar transformar la instancia a una de `Estudiante`, pero como es una instancia de la misma, el casteo funciona sin problemas ya que está tratada como si fuese una de `Persona`.

Si intentamos castear una clase "A" a una clase "B" y estas no tienen relación de herencia, el casteo no funcionaría. Sería como intentar castear una letra a `Integer` lo cual nos arrojaría un error como el siguiente.

```
Integer.parseInt("a");

//Exception in thread "main" java.lang.NumberFormatException: For input
string: "a" at java.lang.NumberFormatException.forInputString(Unknown
Source) at java.lang.Integer.parseInt(Unknown Source) at
java.lang.Integer.parseInt(Unknown Source)
```

Entonces, vayamos a castear la instancia `Estudiante` y luego imprimimos la deuda:

```
for(Persona individuo : lista) {
    Estudiante estudiante = (Estudiante) individuo;
    System.out.println(individuo.getClass().getSimpleName());
    System.out.println(estudiante.getDeuda());
}
```

Si ahora ejecutamos el código, arrojará la excepción mencionada más atrás y esto es debido a que existe una instancia de `Profesor` que no puede castear la instancia `Estudiante` a `Profesor`, ya que no existe una relación directa de herencia.

```
//Estudiante 1500.0 Estudiante 2000.0 Estudiante 3500.0 Exception in
thread "main" java.lang.ClassCastException: Modelo.Profesor cannot be
cast to Modelo.Estudiante at Main.Main.main(Main.java:18)
```

Esta tabla nos indica en qué casos es posible castear una clase:

Desde	Hacia	¿Es posible?
Superclase	Subclase	Sí
Subclase	Superclase	Sí
Subclase 1	Subclase 2	No

Tabla 1. Casteando clases.
Fuente: Desafío Latam.

Por ende, para continuar debemos verificar si es posible castear la clase de la siguiente forma:

```
for(Persona p : lista) {  
    System.out.println(p.getClass().getSimpleName());  
    if(p.getClass() == Estudiante.class) {  
        Estudiante est = (Estudiante) p;  
        System.out.println("Deuda: " + est.getDeuda());  
    }  
}
```

El output será:

```
//Estudiante Deuda: 1500.0  
Estudiante Deuda: 2000.0  
Estudiante Deuda: 3500.0  
Profesor
```

Estamos confirmando que es posible castear la instancia antes de hacerlo, verificando que es una instancia de la clase `Estudiante` con la línea `if(p.getClass() == Estudiante.class)`. De esta forma comparamos la clase de la instancia con la clase de `Estudiante` gracias a `Estudiante.class`, lo que nos entrega una variable de tipo Clase con el valor de clase `Estudiante`.

Con esto hemos logrado comprender el concepto de polimorfismo mediante la práctica y su teoría. Este concepto es fundamental y muy útil en el desarrollo de software que contenga la Orientación a Objetos.

El polimorfismo nos permite crear porciones de código más genéricas, lo que nos ayuda a no repetir una porción de código por cada una de las formas que pueda tener una clase. Apunta no a la misma clase, sino que a un punto más alto dentro de su jerarquía (superclases). Gracias al polimorfismo podemos hacer nuestro código reutilizable.