Orientación a Objetos II (Parte I)

Polimorfismo

Competencias

- 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”.

¿ 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:

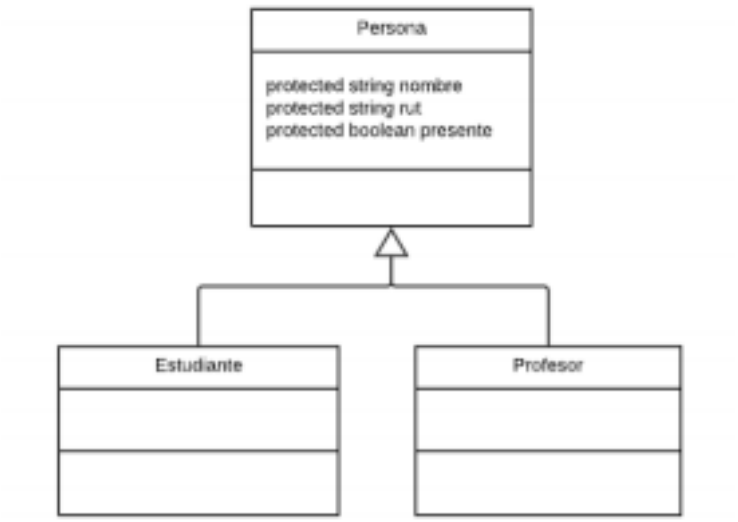

Imagen 2. Herencia de persona.
Fuente: Desafío Latam.

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

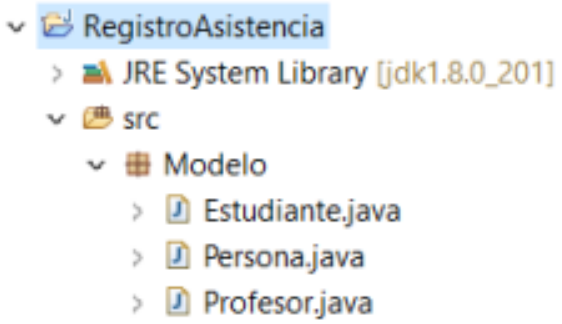

Imagen 3. Estructura del proyecto.
Fuente: Desafío Latam.

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 + "]";
    }
  }
```

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).

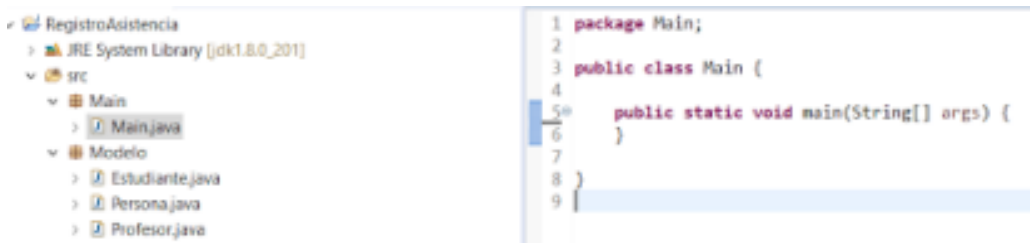

Imagen 4. Creando Main en el proyecto.
Fuente: Desafío Latam.

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 imprimimos 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.

Ejercicio propuesto (1)

En base a lo realizado en el ejercicio guiado “Academia” y el casteo de clases, haremos lo siguiente para finalizar:

- Agregar una variable del tipo Float que se llame Sueldo a la clase `Profesor`.
- Modificar el Main para recibir esta nueva variable.
- Agregar sueldo a 2 profesores.
- Imprimir en pantalla para cada profesor la nueva variable y su respectivo casteo.

Abstracción I

Competencias

- Comprender una interface para separar código.
- Implementar polimorfismo mediante interfaces para el desarrollo de herencias entre clases.

Introducción

En este capítulo continuaremos con polimorfismo en relación con la aplicación en interfaces, importantes a la hora de llevar a cabo el concepto de abstracción. Además, veremos la primera parte de lo que es abstracción en POO, ya que es un concepto que se utiliza bastante a la hora de hacer aplicaciones más dinámicas.

Las interfaces

Como vimos anteriormente, se puede utilizar el polimorfismo entre superclases y subclases, pero no es la única forma de implementar el polimorfismo, ya que también existe algo llamado interfaces.

Las interfaces se declaran utilizando la palabra reservada “interface” en lugar de “class” y proveen una lista de prototipos de métodos, lo que significa que solo se declara tipo de retorno, nombre y parámetros de entrada. Otras clases pueden implementar (heredar) a las interfaces, para lo que deben agregar la palabra “implements” después del nombre de la clase. Al haber implementado una interface, la clase podrá contener sus propias versiones de los métodos de la interface.

```
   //Creación de interface
   public interface nombreInterface{
     void imprimirHola();
   }
   //Implementación de interface:
   public class nombreClase implements nombreInterface[,
nombreOtraInterface]{  @Override
     public void imprimirHola(){
        System.out.println("hola");
    }
   }
```

Además de estos prototipos de método en las interfaces se pueden crear constantes que, como su nombre lo indica, sirven para almacenar valores (como las variables) que no cambiarán.

Las interfaces permiten conocer la lista de métodos que tendrán las clases que las implementen sin conocer el comportamiento específico de cada una, ya que cada implementación puede ser diferente. Los métodos que se declaran en la interface deben existir en todas las clases que la implementen, por ende, ayudan a establecer la forma de las clases, lo que define protocolos de comunicación fácilmente, además de hacer la aplicación de fácil entendimiento para quienes tengan que modificarla.

Por ejemplo, al utilizar un control remoto de televisor, presionamos los botones sabiendo lo que hacen pero no nos interesa saber cómo lo hacen, solo queremos que haga lo que le pedimos que haga.

Imagen 5. Ejemplo de abstracción.
Fuente: Desafío Latam.

Las interfaces no ayudan mucho con la reutilización de código, pero lo mantienen ordenado.

Ejercicio Guiado: El juego

Veamos un ejemplo en código del uso de las interfaces y cómo podríamos utilizarlas aplicando polimorfismo con ellas. Crearemos una porción de la estructura de un juego donde solo se puede avanzar hacia adelante y saltar (algo asi como escapar de un enemigo que rompe todo a su paso).

Paso 1: Crear un proyecto en Eclipse y crear la siguiente interface dentro de un package llamado `Interfaces`:

```
public interface Personaje {
 void mover(int x);
}
```

Paso 2: Crear un package `Personajes` y una implementación de `Personaje` llamada `Protagonista`.

```
public class Protagonista implements Personaje{
}
```

Paso 3: Importar la interface `Personaje` dentro de la clase `Protagonista`, y Eclipse arrojará un mensaje diciendo que debemos implementar los métodos de dicha interface, dándonos la opción de hacerlo automáticamente, tal como se muestra:

```
package Personajes;

import Interfaces.Personaje;

public class Protagonista implements Personaje {
}
```

Esto nos pide que agreguemos implementaciones de los métodos que no están en `Protagonista` y que existen en la interface, es decir, nos obliga a que `Protagonista` tenga la forma de su interface `Personaje`.

Paso 4: Dar clic a la primera opción para que Eclipse genere las implementaciones por nosotros; la clase quedaría así:

```
public class Protagonista implements Personaje{
  @Override
  public void mover(int x) {
    // TODO Auto-generated method stub
  }
}
```

La implementación del método no es más que una sobre-escritura del mismo.

Paso 5: Agregar una variable llamada `xActual` para indicar la posición del personaje y que `mover()` modifique esa variable.

```
public class Protagonista implements Personaje{

   private int xActual;

   @Override
   public void mover(int x){
   xActual = xActual + x;
  }
}
```

Paso 6: Crear otra clase llamada `Enemigo` en el package `Personajes` que implementa la interface `Personaje`. Esto con la idea de crear comportamientos específicos para `Enemigo` y `Protagonista`, es decir, el `Enemigo` avanzará desde el punto A al punto B de una forma progresiva y el `Protagonista` se moverá instantáneamente de un punto a otro.

```
public class Enemigo implements Personaje{
 private int xActual;

 @Override
 public void mover(int x){
 while(xActual < x){
 xActual++;
 }
 }
}
```

Paso 7: Crear una Interface de `Jugador` que le permita a las clases implementar un comportamiento de jugador.

```
package Interfaces;
   public interface Jugador {
   void saltar();
   void ejecutarAccion(String accion);
}
```

Paso 8: Implementar la interface `Jugador` en la clase `Protagonista`.

```
package Personajes;

import Interfaces.Jugador;
import Interfaces.Personaje;

public class Protagonista implements Personaje , Jugador {
...
}
```

Paso 9: Agregar los métodos que no hemos implementado gracias a la opción de la imagen y la clase queda así:

```
 private int xActual;

 @Override
 public void mover(int x){
   xActual = xActual + x;
  }
 @Override
 public void saltar() {
  // TODO Auto-generated method stub
 }
 @Override
 public void ejecutarAccion(String accion) {
  // TODO Auto-generated method stub
 }
```

Paso 10: Crear un comportamiento específico para el protagonista. Primero, modificar la implementación de `saltar()`, agregar una variable representando el eje "y" del plano (`yActual`), que actualmente tiene solo una dimensión y, posteriormente, haremos que el personaje suba y baje paulatinamente.

```
private int yActual = 1;
@Override
public void saltar() {
 //Aumentamos hasta 5
 while(yActual < 5){
   yActual++;
 }
 //Cuando sea 5, disminuimos a 1 nuevamente
 while(yActual > 1){
    yActual--;
  }
}
```

Ahora que tenemos la acción de saltar y mover, vamos a crear un método que permita llamar a estos comportamientos.

Paso 11: Crear la implementación del método `ejecutarAccion` (String accion).

```
@Override
public void ejecutarAccion(String accion) {
 if(accion.equals("saltar") && yActual == 1){
   saltar();
 } else if(accion.equals("avanzar")){
   mover(1);
 }
}
```

Y listo, hemos utilizado dos interfaces en el `Protagonista`, diciendo que tiene la forma de un `Personaje` controlado por el jugador gracias a la forma de la interface Jugador (polimorfismo).

Ejercicio propuesto (2)

- Crear una Interface llamada `Resultado` y un método `ganar` (int pasos).
- Posteriormente, implementamos la interface en `Protagonista` y le decimos que mientras los pasos sean mayores o iguales a 100 el protagonista gana.
- En el método `ejecutarAccion` realizamos otro else if donde la acción “ganar” sea 100.

Palabras de cierre

Hemos visto en este capítulo que en el mundo de la programación existe un concepto abstracto llamado “Polimorfismo”, el cual está directamente relacionado con la herencia entre clases. Es fundamental para cualquier desarrollador/a resolver problemas del día a día mediante este concepto, ya que cubre diversas necesidades en la creación de aplicaciones. Si a todo lo anterior le agregamos la incorporación de interfaces, podemos crear aplicaciones más robustas y compactas.

Soluciones ejercicios propuestos

Solución ejercicio propuesto (1)

Paso 1: Extensión clase `Profesor` en `Persona`.

```
public class Profesor extends Persona{
    private Float sueldo;
public Profesor (Float sueldo, String rut, String nombre, boolean
presente) {
       super(rut, nombre, presente);
       this.sueldo = sueldo;
    }
public Profesor (String rut, String nombre, boolean presente) {
    super(rut, nombre, presente);
}
public Float getSueldo() {
      return sueldo;
}
public void setSueldo(Float sueldo) {
      this.sueldo = sueldo;
  }
}
```

Paso 2: Agregar a los estudiantes y profesores.

```
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(400.5f, "10", "Jose", true));
     lista.add(new Profesor(500.5f, "11", "María", true));
    for(Persona p : lista) {
                  System.out.println(p.getClass().getSimpleName());
                  if(p.getClass() == Profesor.class) {
                  Profesor profe = (Profesor) p;
                  System.out.println("Sueldo: "+ profe.getSueldo());
                }
              }

     ---------------------------
```

```
     Impresión en pantalla:
     Estudiante
     Estudiante
     Estudiante
     Profesor
     Sueldo: 400.5
     Profesor
     Sueldo: 500.5
```

Solución ejercicio propuesto (2)

Paso 1: Crear Interface `Resultado`.

```
package Interfaces;

public interface Resultado {
    void ganar(int pasos);
}
```

Paso 2: Implementación de `Resultado` en `Protagonista`.

```
package personajes;

import Interfaces.Resultado;
import Interfaces.Jugador;
import Interfaces.Personaje;

public class Protagonista implements Personaje, Jugador, Resultado{

    private int xActual;
    private int yActual;
    private int zActual;
    @Override
    public void mover(int x) {
       xActual = xActual + x;

    }
    @Override
    public void saltar() {
       while(yActual<5) {
             yActual++;
```

```
        }
        while (yActual>1) {
              yActual --;
        }

    }
    @Override
    public void ejecutarAccion(String accion) {
        if(accion.equals("saltar") && yActual == 1) {
              saltar();
        } else if(accion.equals("avanzar")) {
              mover(1);
        } else if(accion.equals("ganar")) {
              ganar(100);
        }

    }
    @Override
    public void ganar(int pasos) {
        while(pasos >= 100) {
              System.out.println("Protagonista gana");
        }
    }

}
```