

Universidad de Las Américas
Facultad de Ingenierías y Ciencias Agropecuarias
Ingeniería de Software
Informe de laboratorio

1. DATOS DEL ALUMNO: Camila Cabrera

2. TEMA DE LA PRÁCTICA: Implementación principio Polimorfismo

3. OBJETIVO DE LA PRÁCTICA

- Comprender el principio de Polimorfismo en la programación orientada a objetos. Mediante un ejemplo de aplicación

4. OBJETIVOS ESPECÍFICOS

- Leer y comprender el principio de Polimorfismo
- Inicializar Visual Studio Code para proyectos de java
- Crear un proyecto de java en visual Studio code
- Implementar el principio de Polimorfismo en el proyecto

5. MATERIALES Y MÉTODOS

Materiales:

- Visual studio code
- JDK 17
- Documentación principio Polimorfismo

Métodos:

1. Validación del JDK 17.
2. Configuración de Visual Studio Code para proyectos de Java.
3. Creación del proyecto Java sin herramientas de construcción del proyecto.
4. Implementación del principio de Polimorfismo

5. Registro de hallazgos durante el proceso de implementación de la documentación.

6. DESARROLLO DE LA PRÁCTICA Y RESULTADOS

Marco Teórico

El polimorfismo es uno de los cuatro pilares fundamentales de la programación orientada a objetos (POO), junto con la Abstracción, Encapsulación y Herencia. Este principio se refiere a la capacidad de enviar un mensaje sintácticamente igual a objetos de distintos tipos, que actúan de manera distinta. Se relaciona estrechamente con la herencia, donde una clase padre implementa métodos con comportamientos "genéricos" que pueden ser especializados por las clases hijas.

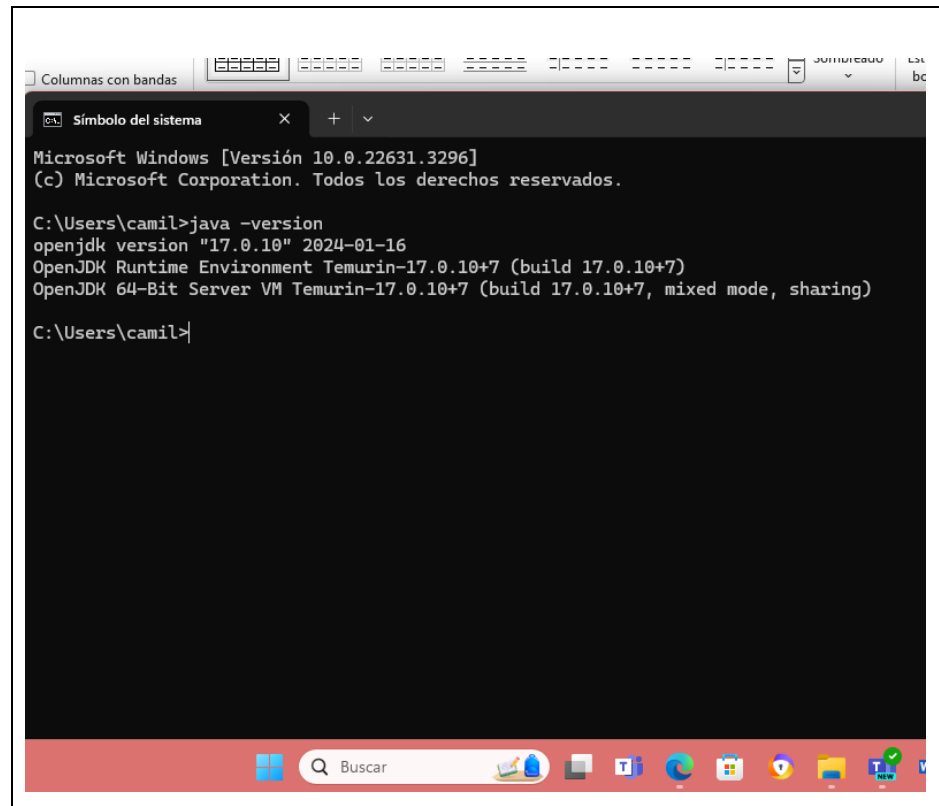
Para implementar el polimorfismo, se puede utilizar una clase padre abstracta que define métodos abstractos, los cuales deben ser implementados en las clases hijas. Esta clase abstracta no puede ser instanciada directamente. Además, se emplea la etiqueta "@Override" en el código para indicar que se está redefiniendo un método de la clase padre en una clase hija. También se puede partir desde un padre no abstracto que posea un método que las clases hijas lo puede sobrescribir. Se pueden sobrescribir o realizar sobrecarga de métodos, donde cual método se ejecutará viene dado por la cantidad de parámetros y tipo.

El polimorfismo es un concepto más avanzado que la herencia y resulta muy útil para establecer un patrón de comportamiento común entre una serie de objetos que heredan de la misma clase.

Desarrollo de la práctica

- **Validación del JDK 17:**
 - Para verificar la versión del JDK se comprueba con el comando “java -version” en la terminal del computador.

Imagen #1: Validación versión JDK



```
Microsoft Windows [Versión 10.0.22631.3296]
(c) Microsoft Corporation. Todos los derechos reservados.

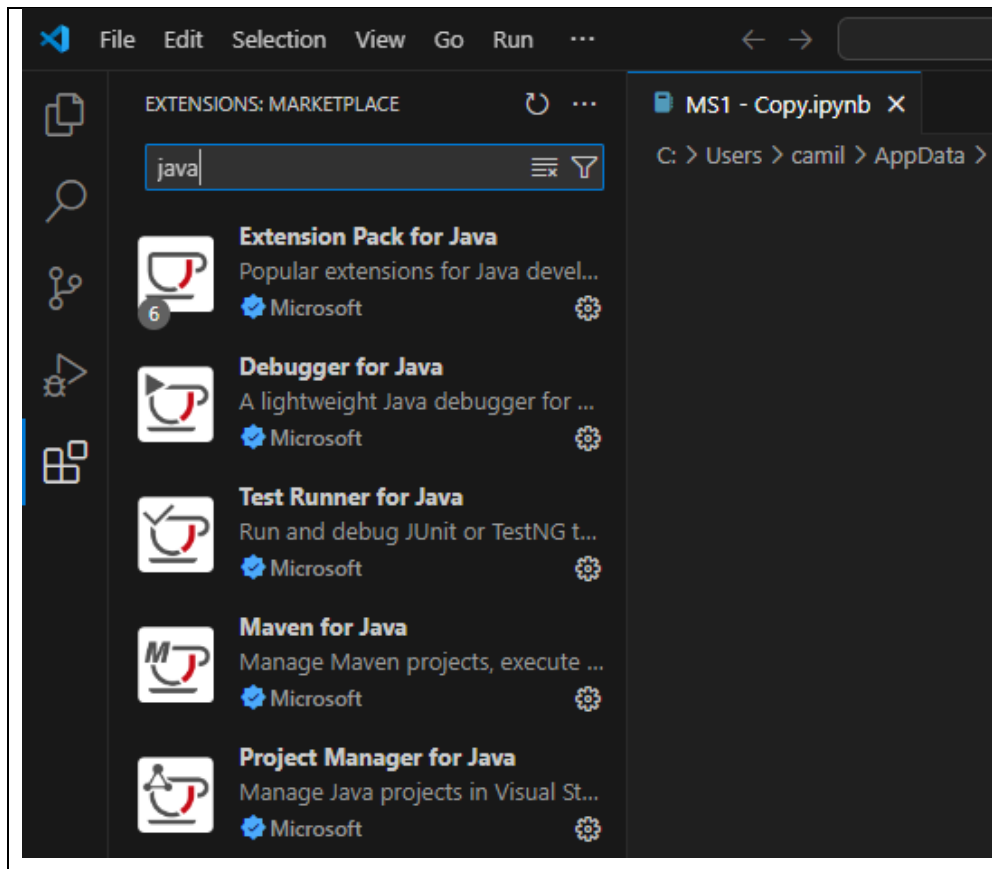
C:\Users\camil>java -version
openjdk version "17.0.10" 2024-01-16
OpenJDK Runtime Environment Temurin-17.0.10+7 (build 17.0.10+7)
OpenJDK 64-Bit Server VM Temurin-17.0.10+7 (build 17.0.10+7, mixed mode, sharing)

C:\Users\camil>
```

Fuente: Elaboración propia

- **2. Configuración de Visual Studio Code para proyectos de Java.**
 - En el IDE visual studio code se debe instalar o validar la dependencia para Java (Imagen 2) .

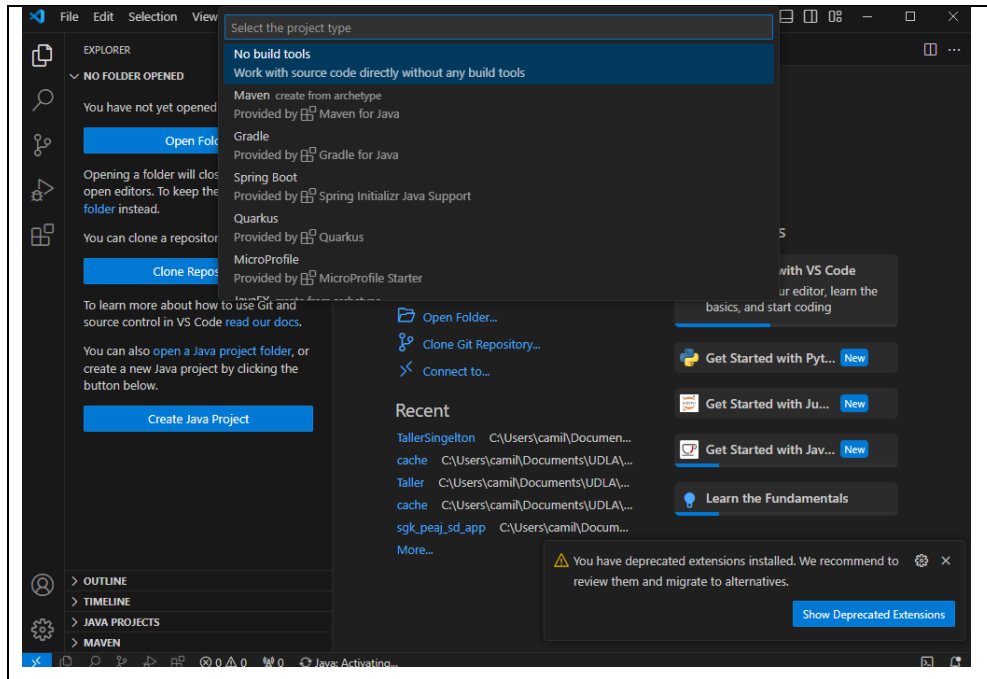
Imagen #2: Dependencia de Java



Fuente: Elaboración propia

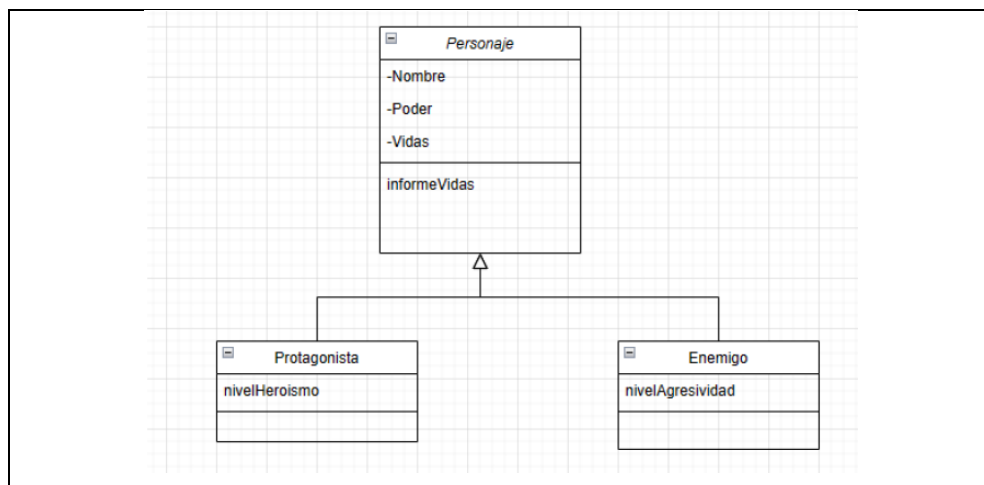
- **Creación del proyecto Java sin herramientas de construcción del proyecto.**
 - Se inicializo el proyecto de Java escogiendo la opción sin herramientas de construcción.

Imagen #3: Dependencia de Java



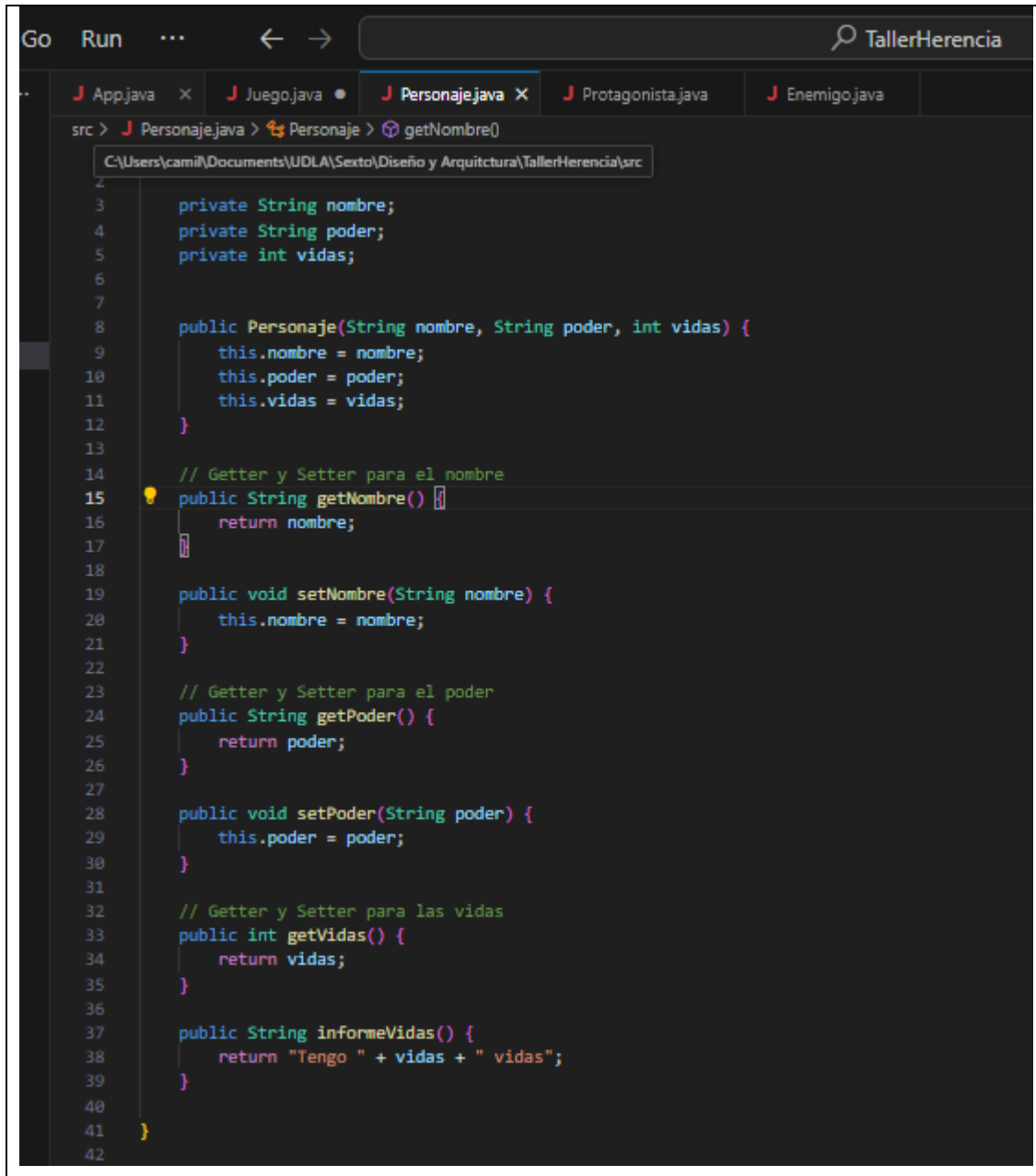
- **Implementación del principio Polimorfismo**
 - Se tomo como base del proyecto, el ejemplo de herencia realizado en clase, que consta de la clase padre Personaje y de las clases hija protagonista y enemigo.

Imagen #3: Diagrama UML



A continuación, se detallan las clases participantes. La clase Personaje es la clase padre la cual tiene atributo nombre, poder y vidas. Además implementa el método informe de vidas que retorna el número de vidas.

Imagen #4: Clase Personaje



```
Go Run ... < > TallerHerencia
J App.java x J Juego.java J Personaje.java x J Protagonista.java J Enemigo.java
src > J Personaje.java > Personaje > getNombre()
C:\Users\camil\Documents\UDLA\Sexto\Diseño y Arquitectura\TallerHerencia\src
2
3     private String nombre;
4     private String poder;
5     private int vidas;
6
7
8     public Personaje(String nombre, String poder, int vidas) {
9         this.nombre = nombre;
10        this.poder = poder;
11        this.vidas = vidas;
12    }
13
14    // Getter y Setter para el nombre
15    public String getNombre() {
16        return nombre;
17    }
18
19    public void setNombre(String nombre) {
20        this.nombre = nombre;
21    }
22
23    // Getter y Setter para el poder
24    public String getPoder() {
25        return poder;
26    }
27
28    public void setPoder(String poder) {
29        this.poder = poder;
30    }
31
32    // Getter y Setter para las vidas
33    public int getVidas() {
34        return vidas;
35    }
36
37    public String informeVidas() {
38        return "Tengo " + vidas + " vidas";
39    }
40
41 }
42
```

La clase Protagonista es la clase hija de personaje que implementa un constructor y tiene como atributo propio el nivel de Heroísmo.

Imagen #5: Clase Protagonista

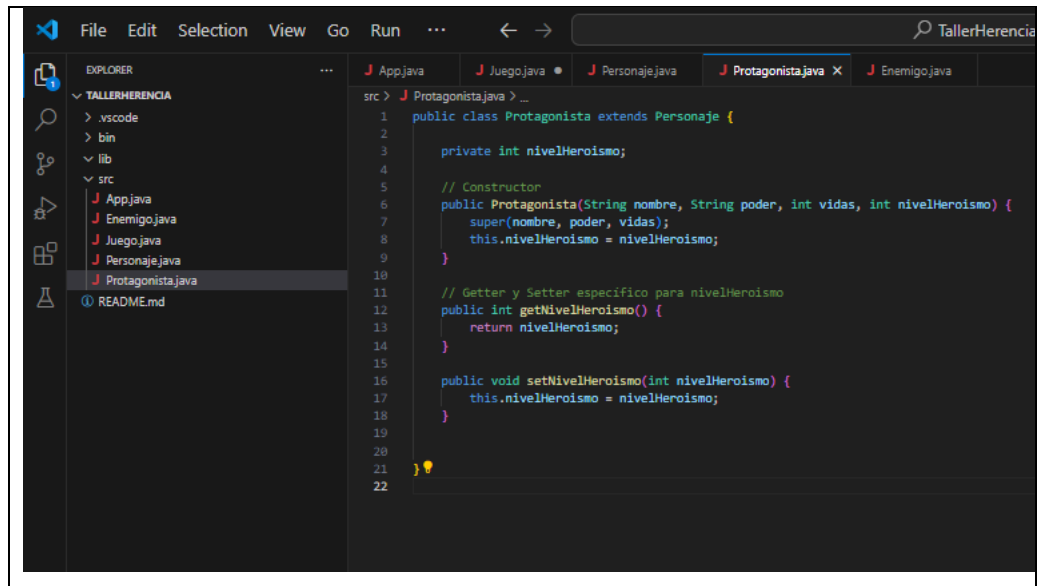
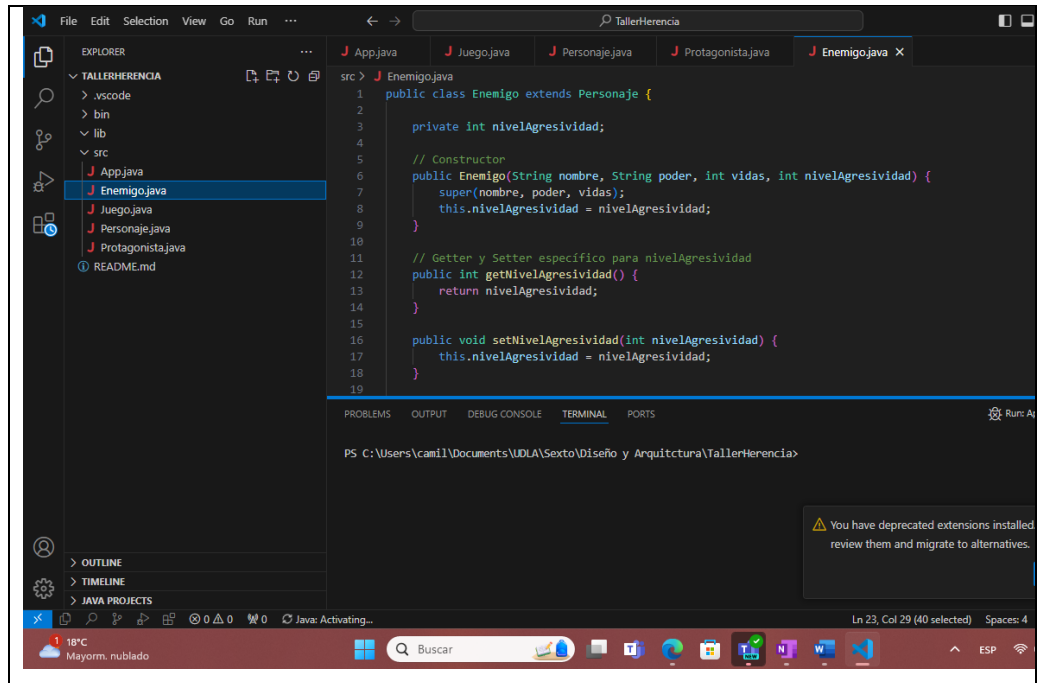


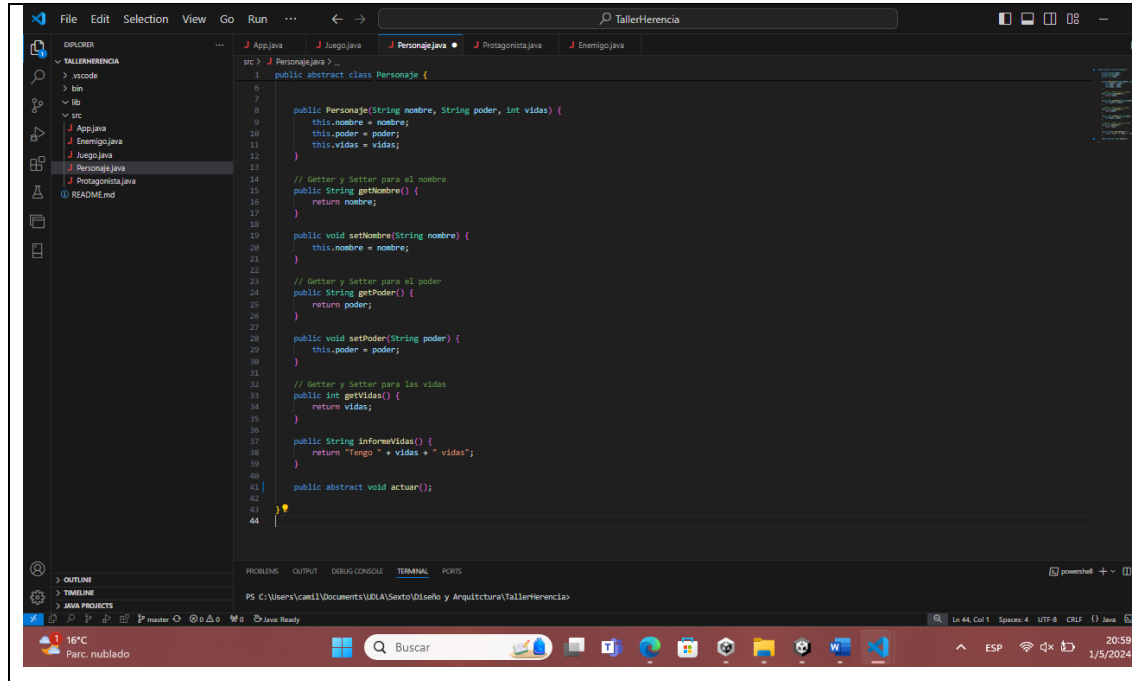
Imagen #6: Clase Enemigo

La clase Enemigo es la clase hija de personaje que implementa un constructor y tiene como atributo propio el nivel de agresividad.



En base a las clases ya establecidas se decidió implementar el principio de polimorfismo, aplicando la sobreescripción de métodos a partir de una clase padre no abstracta que implementa el método.

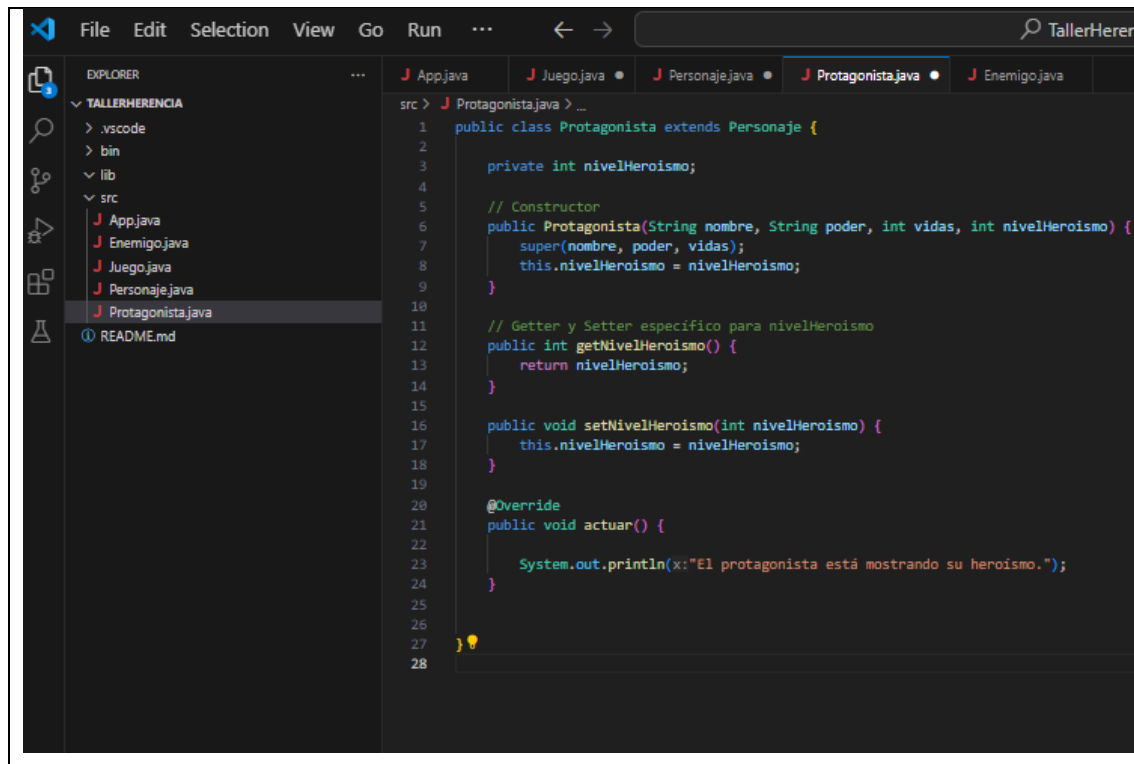
Imagen #7: Clase Padre



```
src > J App.java J Juego.java J Personaje.java J Protagonista.java J Enemigo.java
src > J Personaje.java
1 public abstract class Personaje {
2
3
4
5
6
7
8
9
10 public Personaje(String nombre, String poder, int vidas) {
11     this.nombre = nombre;
12     this.poder = poder;
13     this.vidas = vidas;
14 }
15
16 // Getter y Setter para el nombre
17 public String getNombre() {
18     return nombre;
19 }
20
21 public void setNombre(String nombre) {
22     this.nombre = nombre;
23 }
24
25 // Getter y Setter para el poder
26 public String getPoder() {
27     return poder;
28 }
29
30 public void setPoder(String poder) {
31     this.poder = poder;
32 }
33
34 // Getter y Setter para las vidas
35 public int getVidas() {
36     return vidas;
37 }
38
39 public String informeVidas() {
40     return "Tengo " + vidas + " vidas";
41 }
42
43 public abstract void actuar();
44 }
```

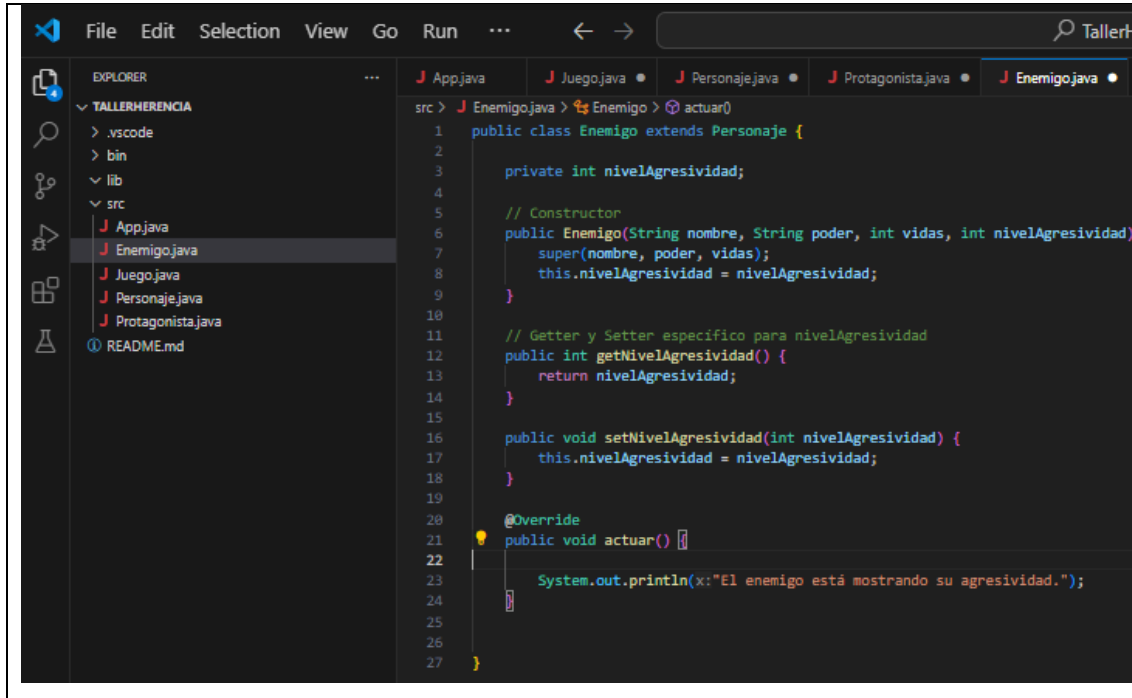
En la clase Personaje se la detallo de tipo abstracta y se aumentó un método actuar que será el que implementen las clases que extiendan de esta.

Imagen #8: Clase Hija Protagonista



En la clase Protagonista se coloca el método actuar y se lo sobrescribe (@override) cambiando el retorno del método a: "El protagonista está mostrando su heroísmo".

Imagen #9: Clase Hija Enemigo

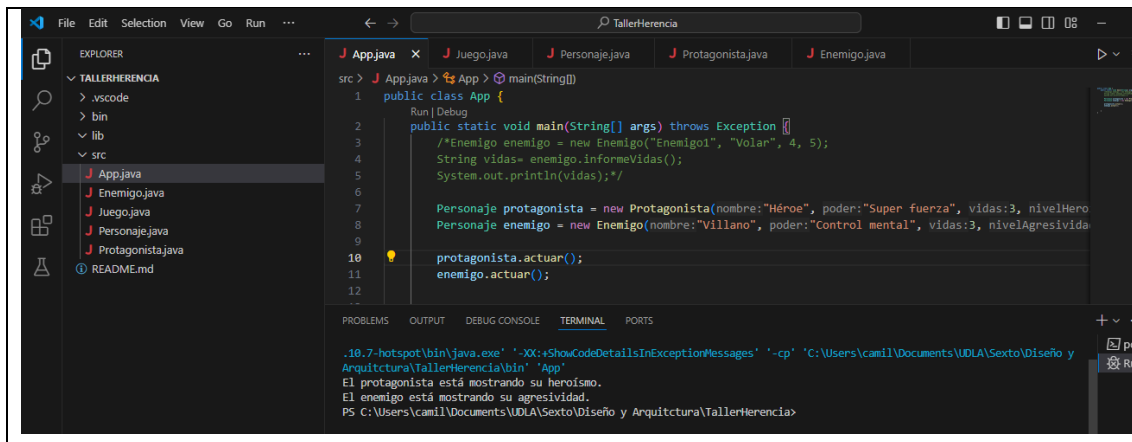


```

src > J Enemigo.java > Enemigo > actuar()
1  public class Enemigo extends Personaje {
2
3      private int nivelAgresividad;
4
5      // Constructor
6      public Enemigo(String nombre, String poder, int vidas, int nivelAgresividad) {
7          super(nombre, poder, vidas);
8          this.nivelAgresividad = nivelAgresividad;
9      }
10
11     // Getter y Setter específico para nivelAgresividad
12     public int getNivelAgresividad() {
13         return nivelAgresividad;
14     }
15
16     public void setNivelAgresividad(int nivelAgresividad) {
17         this.nivelAgresividad = nivelAgresividad;
18     }
19
20     @Override
21     public void actuar() {
22
23         System.out.println("El enemigo está mostrando su agresividad.");
24
25     }
26
27 }
  
```

En la clase enemigo se coloca el método actuar y se lo sobrescribe (@override) cambiando el retorno del método a: “El enemigo está mostrando su agresividad”.

Imagen #10: Clase main



```

src > J App.java > App > main(String[])
1  public class App {
2
3      public static void main(String[] args) throws Exception {
4          /*Enemigo enemigo = new Enemigo("Enemigo1", "Volar", 4, 5);
5          String vidas= enemigo.informeVidas();
6          System.out.println(vidas);*/
7
8          Personaje protagonista = new Protagonista(nombre:"Héroe", poder:"Super fuerza", vidas:3, nivelHero
9          Personaje enemigo = new Enemigo(nombre:"Villano", poder:"Control mental", vidas:3, nivelAgresivida
10
11         protagonista.actuar();
12         enemigo.actuar();
13     }
14 }
  
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

..10.7-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\camil\Documents\UDLA\Sexto\Diseño y
Arquitectura\TallerHerencia\bin' 'App'
El protagonista está mostrando su heroísmo.
El enemigo está mostrando su agresividad.
PS C:\Users\camil\Documents\UDLA\Sexto\Diseño y Arquitectura\TallerHerencia>
  
```

Se instanciaron dos objetos Personaje, el primero es un Protagonista y el segundo es un Enemigo. Luego se llama al método actuar y se mira en consola el mensaje correspondiente a cada subclase. Observando cómo se aplica el principio de polimorfismo donde sintácticamente se llama al mismo método pero cada clase lo implementa de acuerdo a sus especificaciones.

- **Pruebas y Evaluación de los Resultados:**

- Como resultado se pudo aplicar el principio de Polimorfismo y comprender cómo se puede interactuar con la sobreescritura de métodos y la función con diferentes tipos de objetos, lo que aumenta la flexibilidad y extensibilidad del código. Esto facilita la adición de nuevas clases y comportamientos sin necesidad de modificar el código existente.

7. OPINIÓN PERSONAL

- Considero que la programación orientada objetos ha revolucionado completamente el panorama de la programación ya que se enfoca en modelar objetos del mundo real. Esto facilita la comprensión del sistema, ya que refleja de cerca la forma en que interactúan los elementos en el entorno social. Este tipo de programación es modular y permite que se pueda desarrollar en equipos de trabajo grandes donde cada uno se enfoca en un apartado específico del sistema. Desde mi experiencia personal trabajar con POO facilita la integración del equipo y permite abstraer los módulos del mundo real que se pretende automatizar.
- A lo largo de mi experiencia he notado que la POO promueve la reutilización de código a través de conceptos como la herencia y polimorfismo. Las clases y objetos pueden extender o utilizar funcionalidades de otras clases existentes, lo que reduce la duplicación de código y acelera el desarrollo. Por ejemplo, en el caso expuesto la clase padre encapsula los atributos comunes de cada clase y declara métodos genéricos como la consulta de vidas, que

pueden ser extendidos tal cual a sus clases hijas o modificados dependiendo de la necesidad de estas.

- Una ventaja muy significativa referente a la programación orientada a objetos y aplicar el principio de polimorfismo es que permite a los objetos instanciados actuar de diferente forma en función de la situación en la que se los aplica, por ejemplo, si se hace sobrecarga de métodos estos sabrán cuando deben interactuar dependiendo de los parámetros que reciben en tiempo de ejecución.
- Considero que a pesar de las fortalezas que brinda el polimorfismo también se debe tener en cuenta que su aplicación debe ser coherente y bien pensada, dado que el uso excesivo o mal implementado de este principio puede terminar en un código más complejo dificultando la comprensión. Inclusive si la cantidad de clases aumenta y la relación entre ellas puede volver difícil al proceso de rastrear y entender el comportamiento de los objetos.

8. ANEXOS

Link Github del proyecto: [CamilaACT/TallerPolimorfismo \(github.com\)](https://github.com/CamilaACT/TallerPolimorfismo)

9. BIBLIOGRAFÍA

- Polimorfismo en Java: Programación orientada a objetos. (2023, 25 agosto). IfgeekthenNTTdata. <https://ifgeekthen.nttdata.com/es/polimorfismo-en-java-programaci%C3%B3n-orientada-objetos>
- Jarroba, R. [. (2017, 2 junio). Polimorfismo en Java (Parte I), con ejemplos - Jarroba. Jarroba. https://jarroba.com/polimorfismo-en-java-parte-i-con-ejemplos/#google_vignette

