







Desafío 6


Desafío 6


 **Evaluador de Entregas de Programación**

 **Estudiante:** MILAGROS TERESITA POZZO FASINI


 **Centro:** CeRP del Suroeste

 **Curso:** PROGRAMACIÓN II: 02INPGR22025215525 Anual Semipresencial


 **Año:** 2025

 **Desafío 6**

Crea una clase abstracta `Personaje` con atributos `nombre` y `nivel`. Define un método abstracto `accionEspecial()`. Implementa las clases `Mago` y `Guerrero`, cada una con su propia versión de `accionEspecial()`.

 **Tu código de solución:**

Escribe tu código aquí...

 **Fundamentación de tu solución:**

Explica tu estrategia, qué pensaste para resolver el problema, etc.

Para resolver el Desafío 6 pensé primero en separar lo común de lo específico: todos los personajes comparten datos básicos (un nombre y un nivel), pero cada tipo de personaje hace su propia “acción especial”. Por eso definí una clase abstracta `Personaje` que funciona como plantilla: no se crea directamente, pero concentra lo compartido (atributos y un método para mostrar la información) y, además, declara el método abstracto `accionEspecial()`. Hacerlo abstracto obliga a que cada subclase implemente su versión, evitando olvidos y garantizando que todo personaje tenga su habilidad. Luego creé las clases concretas `Mago` y `Guerrero`, que heredan de `Personaje` para reutilizar el código común y solo se enfocan en definir su acción particular (el mago lanza un hechizo y el guerrero ataca con espada). En los constructores de estas subclases llamo a `super(nombre, nivel)` para inicializar correctamente los atributos definidos en la clase base. Opté por un método `mostrarInfo()` en la clase padre para no repetir la misma impresión de nombre y nivel en cada subclase. Al probar el programa, instancio un mago y un guerrero, los refiero mediante el tipo `Personaje` y llamo al mismo método `accionEspecial()` en ambos: esto muestra el polimorfismo de forma simple, porque la misma llamada produce comportamientos distintos según el tipo real del objeto. Esta estrategia mantiene el código claro para nivel inicial, evita duplicaciones, hace que sea fácil agregar nuevos tipos (por ejemplo, un `Arquero`) sin tocar lo ya hecho y asegura que todos los personajes cumplan con la regla principal del diseño: tener su propia acción especial.



Enlace a Desafío 6 en GitHub [https://github.com/MilagrosPozzo/Programacion-2/blob/main/U3 Lenguaje de Programacion Java/Unidad 3 Practico4 Introduccion a JAVA/src/Desaf%C3%ADo6.java](https://github.com/MilagrosPozzo/Programacion-2/blob/main/U3%20Lenguaje%20de%20Programacion%20Java/Unidad%203%20Practico4%20Introduccion%20a%20JAVA/src/Desaf%C3%ADo6.java)