Contenido

[Temario: Aprendiendo POO en Java con Yu-Gi-Oh! 2](#_Toc142396216)

[Módulo 1: Introducción a la Programación Orientada a Objetos y Java 2](#_Toc142396217)

[Módulo 2: Objetos y Clases en el Mundo de Yu-Gi-Oh! 3](#_Toc142396218)

[Módulo 3: Encapsulamiento y Modificadores de Acceso en Duelos de Yu-Gi-Oh! 6](#_Toc142396219)

[Practica 1 8](#_Toc142396220)

[Módulo 4: Herencia y Duelos entre Clases 11](#_Toc142396221)

[Practica 2 20](#_Toc142396222)

[Módulo 5: Composición y Relaciones en el Campo de Batalla 20](#_Toc142396223)

[Módulo 6: Manejo de Excepciones y Trampas en el Código 20](#_Toc142396224)

[Módulo 7: Persistencia de Datos y Guardado de Partidas 21](#_Toc142396225)

[Módulo 8: Proyecto Final - Desarrollo de un Juego de Yu-Gi-Oh! Simplificado 21](#_Toc142396226)

¡Por supuesto! Aquí tienes un temario básico para aprender programación orientada a objetos (POO) en Java utilizando el tema de Yu-Gi-Oh! como contexto. La idea es combinar la enseñanza de los conceptos de POO con elementos del mundo de Yu-Gi-Oh! para hacer el aprendizaje más divertido e interesante.

# Temario: Aprendiendo POO en Java con Yu-Gi-Oh!

# Módulo 1: Introducción a la Programación Orientada a Objetos y Java

**1. ¿Qué es la Programación Orientada a Objetos? Conceptos básicos.**

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos", los cuales son unidades independientes y autónomas que contienen tanto datos (atributos) como comportamientos (métodos) relacionados. La POO busca modelar el mundo real de una manera más natural y estructurada, permitiendo una mayor organización, reutilización y comprensión del código.

Principios Fundamentales de la POO:

| **Principio** | **Descripción** |
| --- | --- |
| **Abstracción** | Representación de los aspectos esenciales de un objeto, ocultando detalles innecesarios. |
| **Encapsulamiento** | Ocultación de los detalles internos de un objeto, exponiendo solo la interfaz necesaria. |
| **Herencia** | Creación de nuevas clases basadas en clases existentes, heredando atributos y comportamientos. |
| **Polimorfismo** | Tratar objetos de diferentes clases de manera uniforme, permitiendo intercambiar tipos de manera coherente. |

**2. Introducción a Java: Historia y características principales.**

**3. Configuración del entorno de desarrollo: Instalación de JDK y configuración de IDE.**

# Módulo 2: Objetos y Clases en el Mundo de Yu-Gi-Oh!

**1. Clases y Objetos: Definición y diferencia.**

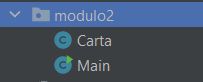
| Concepto | Descripción |
| --- | --- |
| Clases y Objetos | Una clase es una plantilla que define la estructura y el comportamiento de un tipo de objeto. Un objeto es una instancia concreta de una clase. |
| Atributos | Los atributos son variables que almacenan datos relacionados con un objeto. En Yu-Gi-Oh!, los atributos podrían representar características de las cartas. |
| Métodos | Los métodos son funciones que definen el comportamiento de un objeto. En Yu-Gi-Oh!, los métodos podrían permitir activar efectos o realizar acciones en el juego. |
| Constructores | Los constructores son métodos especiales utilizados para inicializar objetos al crear instancias de una clase. |
| Instancia | Una instancia es un objeto concreto creado a partir de una clase. |

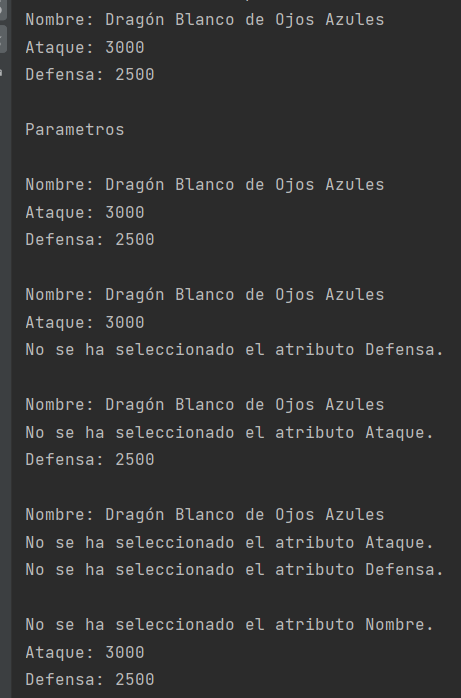
**2. Creación de clases y objetos basados en cartas de Yu-Gi-Oh!.**

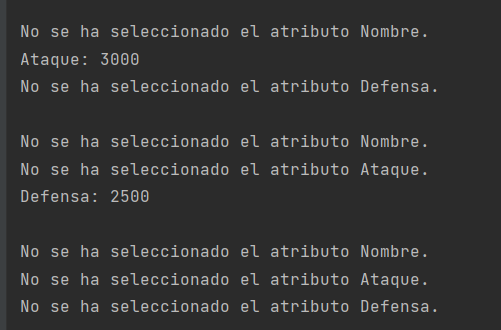
**3. Atributos y métodos: Asociación con características de las cartas.**

package modulo2;  
  
public class Carta {  
  
 //ATRBUTOS  
 String nombre;  
 int ataque;  
 int defensa;  
  
 //CONSTRUCTOR  
 public Carta(String nombre, int ataque, int defensa) {  
 this.nombre = nombre;  
 this.ataque = ataque;  
 this.defensa = defensa;  
 }  
  
 //METODOS  
 public void mostrarInformacion() {  
 System.*out*.println("Nombre: " + nombre);  
 System.*out*.println("Ataque: " + ataque);  
 System.*out*.println("Defensa: " + defensa);  
 }  
  
 public void mostrarInformacionParametros(boolean name, boolean atk, boolean def) {  
 System.*out*.println("");  
 if (name) {  
 System.*out*.println("Nombre: " + nombre);  
 } else {  
 System.*out*.println("No se ha seleccionado el atributo Nombre.");  
 }  
  
 if (atk) {  
 System.*out*.println("Ataque: " + ataque);  
 } else {  
 System.*out*.println("No se ha seleccionado el atributo Ataque.");  
 }  
  
 if (def) {  
 System.*out*.println("Defensa: " + defensa);  
 } else {  
 System.*out*.println("No se ha seleccionado el atributo Defensa.");  
 }  
  
 }  
  
}

package modulo2;  
  
public class Main {  
 public static void main(String[] args) {  
  
 //Creacion de una instancia (objeto) de la clase carta  
 Carta miCarta = new Carta("Dragón Blanco de Ojos Azules", 3000, 2500);  
  
 miCarta.mostrarInformacion();  
  
 System.*out*.println("\nParametros");  
 // Combinaciones de parámetros para mostrarInformacionParametros  
 miCarta.mostrarInformacionParametros(true, true, true);  
 miCarta.mostrarInformacionParametros(true, true, false);  
 miCarta.mostrarInformacionParametros(true, false, true);  
 miCarta.mostrarInformacionParametros(true, false, false);  
 miCarta.mostrarInformacionParametros(false, true, true);  
 miCarta.mostrarInformacionParametros(false, true, false);  
 miCarta.mostrarInformacionParametros(false, false, true);  
 miCarta.mostrarInformacionParametros(false, false, false);  
  
 }//MAIN  
}//CLASS







# Módulo 3: Encapsulamiento y Modificadores de Acceso en Duelos de Yu-Gi-Oh!

**1. Encapsulamiento: Definición y ventajas.**

**Concepto de Encapsulamiento:** El encapsulamiento es uno de los principios fundamentales de la Programación Orientada a Objetos (POO). En el contexto de Yu-Gi-Oh!, el encapsulamiento se refiere a la práctica de ocultar los detalles internos de una carta o jugador y proporcionar una interfaz controlada para interactuar con ellos. Esto se logra mediante la restricción del acceso directo a los atributos y métodos internos y utilizando métodos públicos para acceder y modificar estos valores.

**2. Modificadores de acceso: public, private, protected y default.**

**Modificadores de Acceso:** Los modificadores de acceso son palabras clave utilizadas en la declaración de atributos y métodos para controlar el nivel de acceso que otros componentes del programa tienen a ellos. En Yu-Gi-Oh!, estos modificadores ayudan a determinar qué información y acciones están disponibles para los jugadores y cómo interactúan con las cartas.

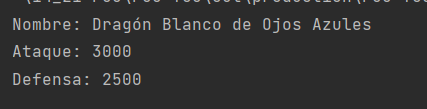
Tipos de Modificadores de Acceso:

* **public**: El atributo o método es accesible desde cualquier parte del programa, incluyendo otras clases.
* **private**: El atributo o método solo es accesible desde dentro de la misma clase. Otros componentes no pueden acceder directamente a ellos.
* **protected**: El atributo o método es accesible desde la misma clase y también desde subclases (herederos).
* **default** (sin modificador): El atributo o método es accesible solo dentro del mismo paquete.

**3. Aplicación de encapsulamiento y modificadores en la representación de cartas y jugadores**.

package modulo3;  
  
public class Carta {  
  
 //ATRBUTOS  
 private String nombre;  
 private int ataque;  
 private int defensa;  
  
 //CONSTRUCTOR  
  
 //METODOS  
 public void mostrarInformacion() {  
 System.*out*.println("Nombre: " + nombre);  
 System.*out*.println("Ataque: " + ataque);  
 System.*out*.println("Defensa: " + defensa);  
 }  
  
 //GET AND SET  
 public String getNombre() {  
 return nombre;  
 }  
  
 public void setNombre(String nombre) {  
 this.nombre = nombre;  
 }  
  
 public int getAtaque() {  
 return ataque;  
 }  
  
 public void setAtaque(int ataque) {  
 this.ataque = ataque;  
 }  
  
 public int getDefensa() {  
 return defensa;  
 }  
  
 public void setDefensa(int defensa) {  
 this.defensa = defensa;  
 }  
  
}

package modulo3;  
  
public class Main {  
 public static void main(String[] args) {  
  
 Carta miCarta = new Carta();  
  
 miCarta.setNombre("Dragón Blanco de Ojos Azules");  
 miCarta.setAtaque(3000);  
 miCarta.setDefensa(2500);  
  
 miCarta.mostrarInformacion();  
  
  
 }//MAIN  
}//CLASS



# Practica 1

---  
  
\*\*Práctica de Programación: Cartas de Yu-Gi-Oh!\*\*  
  
Crea una clase llamada `Monstruo` que represente a un monstruo del juego de cartas Yu-Gi-Oh!.  
La clase debe tener los siguientes atributos privados:  
  
- `nombre` (String): El nombre del monstruo.  
- `nivel` (int): El nivel del monstruo.  
- `ataque` (int): Los puntos de ataque del monstruo.  
- `defensa` (int): Los puntos de defensa del monstruo.  
  
Implementa los siguientes métodos públicos en la clase `Monstruo`:  
  
1. Un constructor que acepte parámetros para inicializar los atributos `nombre`, `nivel`, `ataque` y `defensa`.  
2. Métodos `get` y `set` para cada uno de los atributos.  
3. Un método `calcularPoderTotal()` que calcule y retorne el poder total del monstruo sumando sus puntos de ataque y defensa.  
4. Un método `subirDeNivel()` que aumente el nivel del monstruo en 1 y ajuste sus puntos de ataque y defensa en +300 cada uno.  
5. Un método `mostrarInformacionDetallada()` que muestre en la consola la información del monstruo, incluyendo su nombre, nivel, ataque, defensa y poder total.  
  
Luego, en la clase `Main`, crea una instancia de `Monstruo` utilizando el constructor, establece valores para sus atributos utilizando los métodos `set`,  
y utiliza los nuevos métodos para calcular el poder total, subir de nivel y mostrar información detallada del monstruo.  
  
---  
  
Espero que este enunciado te brinde una práctica más completa y desafiante para aplicar los conceptos aprendidos en los Módulos 2 y 3. ¡Diviértete programando!

package practices.practice1;  
  
public class Monstruo {  
 //ATRIBUTOS  
 private String nombre;  
 private int ataque;  
 private int defensa;  
 private int nivel;  
  
 //CONSTRUCTOR  
  
  
 public Monstruo() {  
 }  
  
 public Monstruo(String nombre, int ataque, int defensa, int nivel) {  
 this.nombre = nombre;  
 this.ataque = ataque;  
 this.defensa = defensa;  
 this.nivel = nivel;  
 }  
  
 //METODOS  
 public int calcularPoderTotal() {  
 int total = ataque + defensa;  
 return total;  
 }  
  
 public String mostrarInformacionDetallada() {  
 StringBuilder sb = new StringBuilder();  
 sb.append("\nNombre: " + getNombre());  
 sb.append("\nAtaque: " + getAtaque());  
 sb.append("\nDefensa: " + getDefensa());  
 sb.append("\nNivel: " + getNivel());  
 sb.append("\nPoder Total:" + calcularPoderTotal());  
// sb.append("\nPoder Total:" + subirDeNivel());  
 return sb.toString();  
 }  
  
 public void subirDeNivel() {  
 nivel++;  
 ataque += 300;  
 defensa += 300;  
 }  
  
  
 //GET AND SET  
 public String getNombre() {  
 return nombre;  
 }  
  
 public void setNombre(String nombre) {  
 this.nombre = nombre;  
 }  
  
 public int getAtaque() {  
 return ataque;  
 }  
  
 public void setAtaque(int ataque) {  
 this.ataque = ataque;  
 }  
  
 public int getDefensa() {  
 return defensa;  
 }  
  
 public void setDefensa(int defensa) {  
 this.defensa = defensa;  
 }  
  
 public int getNivel() {  
 return nivel;  
 }  
  
 public void setNivel(int nivel) {  
 this.nivel = nivel;  
 }  
  
}

package practices.practice1;  
  
public class Main {  
 public static void main(String[] args) {  
  
 Monstruo cartaMonstruo = new Monstruo();  
  
 cartaMonstruo.setNombre("Dark Magician");  
 cartaMonstruo.setAtaque(2500);  
 cartaMonstruo.setDefensa(2100);  
 cartaMonstruo.setNivel(7);  
  
  
 for(int i =1; i<4; i++){  
 System.*out*.println("\nsalida "+i);  
 System.*out*.print(cartaMonstruo.mostrarInformacionDetallada());  
 cartaMonstruo.subirDeNivel();  
 }  
  
 }  
}



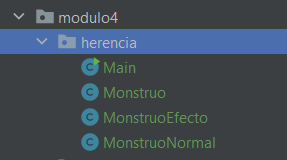
# Módulo 4: Herencia y Duelos entre Clases

**1. Herencia: Concepto y jerarquías de clases.**

La herencia permite crear una relación entre clases donde una clase más específica (subclase) puede heredar características de una clase más general (superclase). Esto significa que la subclase puede acceder a los atributos y métodos públicos o protegidos de la superclase, además de poder agregar sus propios atributos y métodos específicos.

Las jerarquías de clases se forman a medida que creamos subclases que extienden (heredan de) una superclase. Las subclases pueden, a su vez, ser superclases de otras clases, formando una cadena de herencia.

**2. Clases base y clases derivadas: Relación en el contexto de Yu-Gi-Oh!.**

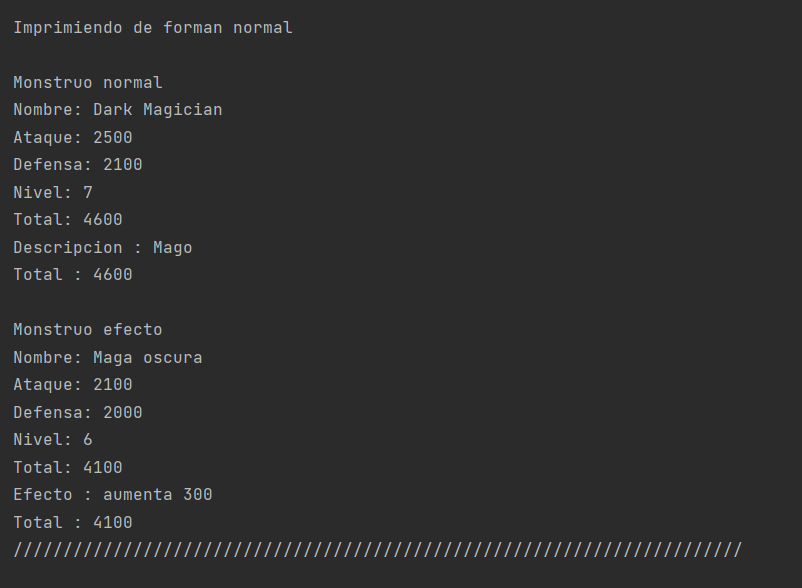
****

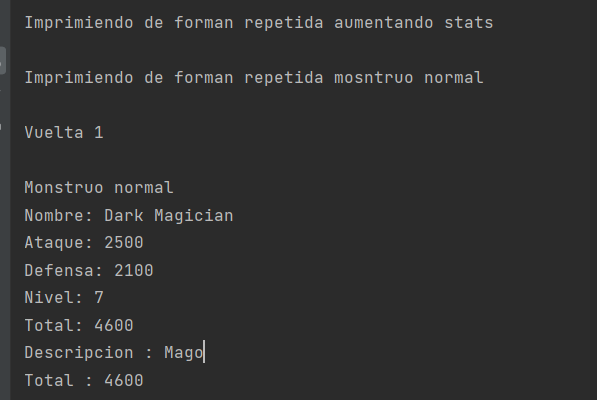
package modulo4.herencia;  
  
public class Monstruo {  
  
 //ATRIBUTOS  
 private String nombre;  
 private int nivel;  
 private int ataque;  
 private int defensa;  
  
 //CONSTRUCTOR  
 public Monstruo() {  
 }  
  
 public Monstruo(String nombre, int nivel, int ataque, int defensa) {  
 this.nombre = nombre;  
 this.nivel = nivel;  
 this.ataque = ataque;  
 this.defensa = defensa;  
 }  
  
 //METODOS  
 public int calcularPoderTotal(){  
 int total = ataque + defensa;  
 return total;  
 }  
  
 public void subirDeNivel() {  
 nivel++;  
 ataque += 300;  
 defensa += 300;  
 }  
  
  
 public String mostrarInformacion(){  
 StringBuilder sb = new StringBuilder();  
 sb.append("\nNombre: "+nombre);  
 sb.append("\nAtaque: "+ataque);  
 sb.append("\nDefensa: "+defensa);  
 sb.append("\nNivel: "+nivel);  
 sb.append("\nTotal: "+calcularPoderTotal());  
 return sb.toString();  
 }  
  
  
 //GET AND SET  
 public String getNombre() {  
 return nombre;  
 }  
  
 public void setNombre(String nombre) {  
 this.nombre = nombre;  
 }  
  
 public int getNivel() {  
 return nivel;  
 }  
  
 public void setNivel(int nivel) {  
 this.nivel = nivel;  
 }  
  
 public int getAtaque() {  
 return ataque;  
 }  
  
 public void setAtaque(int ataque) {  
 this.ataque = ataque;  
 }  
  
 public int getDefensa() {  
 return defensa;  
 }  
  
 public void setDefensa(int defensa) {  
 this.defensa = defensa;  
 }  
  
  
}

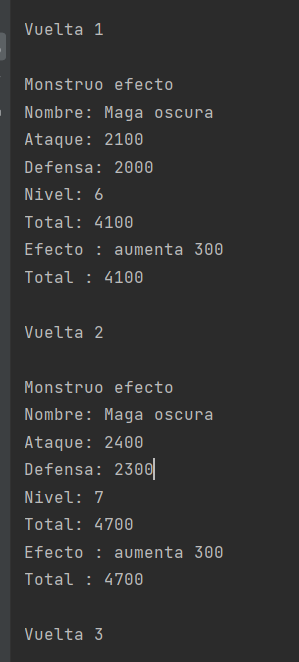
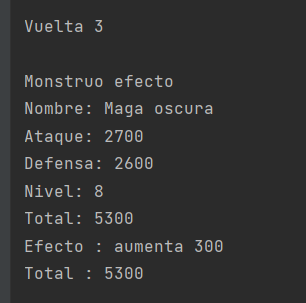
package modulo4.herencia;  
  
public class MonstruoNormal extends Monstruo{  
  
 //ATRIBUTOS  
 String descripcion;  
  
 //CONSTRUCTOR  
 public MonstruoNormal(){  
  
 }  
  
 public MonstruoNormal(String nombre, int nivel, int ataque, int defensa, String descripcion) {  
 super(nombre, nivel, ataque, defensa);  
 this.descripcion = descripcion;  
 }  
  
 //METODOS  
 @Override  
 public String mostrarInformacion(){  
 StringBuilder sb = new StringBuilder();  
 sb.append("\nMonstruo normal");  
 sb.append(super.mostrarInformacion());  
 sb.append("\nDescripcion : "+descripcion);  
 sb.append("\nTotal : "+calcularPoderTotal());  
 return sb.toString();  
 }  
  
  
 //GET AND SET  
 public String getDescripcion() {  
 return descripcion;  
 }  
  
 public void setDescripcion(String descripcion) {  
 this.descripcion = descripcion;  
 }  
  
}

package modulo4.herencia;  
  
public class MonstruoEfecto extends Monstruo {  
  
 //ATRIBUTOS  
 String efecto;  
  
 //CONSTRUCTOR  
 public MonstruoEfecto() {  
 }  
  
 public MonstruoEfecto(String nombre, int nivel, int ataque, int defensa, String efecto) {  
 super(nombre, nivel, ataque, defensa);  
 this.efecto = efecto;  
 }  
  
 //METODOS  
 @Override  
 public String mostrarInformacion(){  
 StringBuilder sb = new StringBuilder();  
 sb.append("\nMonstruo efecto");  
 sb.append(super.mostrarInformacion());  
 sb.append("\nEfecto : "+efecto);  
 sb.append("\nTotal : "+calcularPoderTotal());  
 return sb.toString();  
 }  
  
  
 //GET AND SET  
 public String getEfecto() {  
 return efecto;  
 }  
  
 public void setEfecto(String efecto) {  
 this.efecto = efecto;  
 }  
  
}

package modulo4.herencia;  
  
public class Main {  
 public static void main(String[] args) {  
  
 MonstruoNormal cartaMonstruoNormal = new MonstruoNormal("Dark Magician", 7, 2500, 2100, "Mago");  
 MonstruoEfecto cartaMonstruoEfecto = new MonstruoEfecto("Maga oscura", 6, 2100, 2000, "aumenta 300");  
  
 System.*out*.println("\nImprimiendo de forman normal");  
  
 System.*out*.println(cartaMonstruoNormal.mostrarInformacion());  
 System.*out*.println(cartaMonstruoEfecto.mostrarInformacion());  
  
  
 System.*out*.println("/////////////////////////////////////////////////////////////////////////");  
  
 System.*out*.println("\nImprimiendo de forman repetida aumentando stats");  
 System.*out*.println("\nImprimiendo de forman repetida mosntruo normal");  
 for (int i = 1; i < 4; i++) {  
 System.*out*.println("\nVuelta " + i);  
 System.*out*.println(cartaMonstruoNormal.mostrarInformacion());  
 cartaMonstruoNormal.subirDeNivel();  
 }  
  
 System.*out*.println("/////////////////////////////////////////////////////////////////////////");  
 System.*out*.println("\nImprimiendo de forman repetida mosntruo cartaMonstruoEfecto");  
 for (int i = 1; i < 4; i++) {  
 System.*out*.println("\nVuelta " + i);  
 System.*out*.println(cartaMonstruoEfecto.mostrarInformacion());  
 cartaMonstruoEfecto.subirDeNivel();  
 }  
  
  
 }//MAIN  
}//CLASS

****

** **

** **

**3. Polimorfismo: Utilización de interfaces y clases abstractas para representar diferentes tipos de cartas.**

| Concepto | Descripción |
| --- | --- |
| Polimorfismo | Permite tratar objetos de diferentes clases de manera uniforme a través de una interfaz común. |
| Clase Abstracta | Una clase que no se puede instanciar directamente y puede contener métodos abstractos y concretos. |
| Interface | Una especificación de métodos que una clase debe implementar, permitiendo la herencia múltiple en Java. |

**Polimorfismo**

En la clase MonstruoEfecto agregue el método

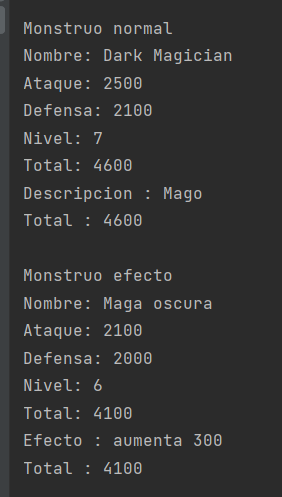
public void activarEfecto(){  
 System.*out*.println("Mi Monstruo "+getNombre()+" a activado su efecto");  
}

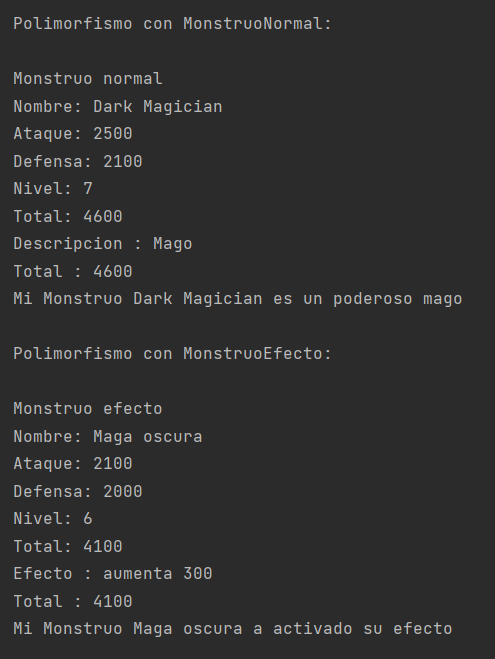
Y en la clase Monstruo Normal agregue el método

public void leerDescripcion(){  
 System.*out*.println("Mi Monstruo "+getNombre()+" es un poderoso mago");  
}

y el main

package modulo4.polimorfismo;  
  
public class Main {  
 public static void main(String[] args) {  
  
 MonstruoNormal cartaMonstruoNormal = new MonstruoNormal("Dark Magician", 7, 2500, 2100, "Mago");  
 MonstruoEfecto cartaMonstruoEfecto = new MonstruoEfecto("Maga oscura", 6, 2100, 2000, "aumenta 300");  
   
 System.*out*.println(cartaMonstruoNormal.mostrarInformacion());  
 System.*out*.println(cartaMonstruoEfecto.mostrarInformacion());  
  
 Monstruo polimorfismoNormal = cartaMonstruoNormal;  
 Monstruo polimorfismoEfecto = cartaMonstruoEfecto;  
  
 System.*out*.println("\nPolimorfismo con MonstruoNormal:");  
 System.*out*.println(polimorfismoNormal.mostrarInformacion());  
 ((MonstruoNormal)polimorfismoNormal).leerDescripcion();  
  
 System.*out*.println("\nPolimorfismo con MonstruoEfecto:");  
 System.*out*.println(polimorfismoEfecto.mostrarInformacion());  
 ( (MonstruoEfecto) polimorfismoEfecto).activarEfecto();  
  
 }//MAIN  
}//CLASS





# Practica 2

Esta en git hub

**Clase abstracta**

1. Clase Abstracta: Una clase abstracta es una clase que no se puede instanciar directamente, es decir, no se pueden crear objetos de esa clase. Sirve como una plantilla o base para otras clases. Puede contener tanto métodos abstractos como métodos concretos (con implementación).
2. Método Abstracto: Un método abstracto es un método declarado en una clase abstracta pero sin proporcionar una implementación en esa clase. En lugar de eso, solo se declara su firma (nombre, parámetros y tipo de retorno). Los métodos abstractos se utilizan para definir un contrato que las clases derivadas deben cumplir. Las clases derivadas deben proporcionar una implementación concreta para los métodos abstractos definidos en la clase abstracta.
3. Implementación Obligatoria: Cuando una clase hereda de una clase abstracta que tiene métodos abstractos, la clase derivada debe proporcionar una implementación concreta para todos los métodos abstractos heredados. Esto es una obligación impuesta por la herencia y la relación entre la clase base (abstracta) y la clase derivada.
4. Polimorfismo y Herencia: Cuando utilizamos clases abstractas y herencia, podemos aprovechar el polimorfismo para tratar objetos de clases derivadas como objetos de la clase base (abstracta). Esto nos permite crear código que funcione de manera general para múltiples tipos de objetos derivados.

Una clase abstracta es una clase en la programación orientada a objetos que no puede ser instanciada directamente, es decir, no se pueden crear objetos de esta clase. En lugar de eso, una clase abstracta se utiliza como una base para otras clases, proporcionando una estructura común y estableciendo métodos que las clases derivadas deben implementar. Las clases abstractas pueden contener métodos abstractos (sin implementación) y métodos concretos (con implementación). Las clases que heredan de una clase abstracta deben implementar todos los métodos abstractos definidos en la clase base.

**Revisar el código en git hub**

**Si el método `obtener` en la clase `Monstruo` se modifica el modificador de acceso, tendría diferentes efectos en la visibilidad y la capacidad de acceso desde otras clases:**

**1. `private`: Si cambias `private String obtener()` en la clase `Monstruo`, solo podría ser accedido dentro de la propia clase `Monstruo`. Las clases derivadas, como `MonstruoNormal` y `MonstruoEfecto`, no podrían acceder a este método y resultaría en un error de compilación si intentan hacerlo.**

**2. `public`: Si cambias `private String obtener()` a `public String obtener()`, el método sería accesible desde cualquier parte del programa, incluyendo desde clases externas. Esto puede tener ventajas, como permitir un acceso más directo a la información, pero también puede exponer innecesariamente la implementación interna de la clase.**

**3. `protected`: Si cambias `private String obtener()` a `protected String obtener()`, el método sería accesible dentro de la clase `Monstruo` y sus clases derivadas (subclases) como `MonstruoNormal` y `MonstruoEfecto`. Sin embargo, no sería accesible desde clases fuera de la jerarquía de herencia.**

**En resumen, el modificador de acceso de un método determina quién puede acceder a él desde otras clases. La elección del modificador de acceso depende de la encapsulación deseada y de cómo deseas que las clases interactúen entre sí.**

**Interface**

# Módulo 5: Composición y Relaciones en el Campo de Batalla

1. Composición: Creación de objetos complejos a partir de otros objetos.

2. Relaciones entre clases: Asociación, agregación y composición en los duelos de Yu-Gi-Oh!.

3. Implementación de relaciones entre cartas, mazos y jugadores.

# Módulo 6: Manejo de Excepciones y Trampas en el Código

1. Excepciones en Java: Tipos y manejo.

2. Implementación de manejo de excepciones en el juego de Yu-Gi-Oh!.

3. Creación de trampas (clases de manejo de excepciones personalizadas) para situaciones específicas en el juego.

# Módulo 7: Persistencia de Datos y Guardado de Partidas

1. Serialización: Guardado y carga de objetos en archivos.

2. Implementación de la persistencia de datos para guardar y cargar partidas de Yu-Gi-Oh!.

3. Uso de archivos para mantener un registro de las cartas, mazos y resultados de los duelos.

# Módulo 8: Proyecto Final - Desarrollo de un Juego de Yu-Gi-Oh! Simplificado

1. Aplicación de todos los conceptos aprendidos en la creación de un juego básico de Yu-Gi-Oh! en consola.

2. Implementación de duelos, cartas, jugadores, reglas básicas y persistencia de datos.

3. Pruebas y depuración del juego finalizado.

Recuerda que este temario es una guía general y puedes adaptarlo según tus necesidades y nivel de conocimiento. ¡Diviértete aprendiendo POO en Java con el emocionante mundo de Yu-Gi-Oh! como contexto!