OBJETOS

Supongamos que queremos modelar un objeto "Persona" que tenga atributos como nombre, edad y ocupación. Crearemos una clase llamada "Persona" con métodos para inicializar los atributos y mostrar la información de la persona.

Crea una instancia de "Persona" y muestra su información.

OBJETOS

Supongamos que estás diseñando un sistema de gestión de biblioteca y necesitas modelar el concepto de libros.

1. Define una clase llamada "Libro" que tenga los siguientes atributos:
    - Título
    - Autor
    - ISBN (International Standard Book Number)
    - Año de publicación
    - Número de páginas
2. La clase "Libro" debe tener métodos para:
    - Obtener y establecer el título del libro.
    - Obtener y establecer el autor del libro.
    - Obtener y establecer el ISBN del libro.
    - Obtener y establecer el año de publicación del libro.
    - Obtener y establecer el número de páginas del libro.
    - Imprimir toda la información del libro.
3. Crea al menos dos instancias de la clase "Libro" y muestra toda la información de cada libro.

Este ejercicio teórico te permite aplicar conceptos como la creación de clases, la definición de atributos y métodos, así como la creación de instancias de objetos y el acceso a sus atributos y métodos.

ENCAPSULACIÓN

Supongamos que estás diseñando una clase llamada "Estudiante" en un sistema de gestión escolar. Quieres encapsular el atributo de calificación para que no se pueda acceder directamente desde fuera de la clase, pero aún así permitir a los usuarios acceder a la calificación solo para lectura y actualizarla solo a través de métodos específicos.

Los usuarios pueden obtener la calificación llamando a un método y actualizarla llamando a otro método.

ENCAPSULACIÓN

Supongamos que queremos crear una clase llamada "CuentaBancaria" que tenga atributos como saldo y métodos para depositar y retirar dinero, pero queremos encapsular el saldo para que no se pueda acceder directamente desde fuera de la clase.

Utilizaremos métodos para acceder y modificar el saldo de manera controlada.

HERENCIA

Imagina que tienes una clase base llamada "Vehículo" que tiene atributos y métodos comunes para todos los vehículos, como "marca", "modelo", "año de fabricación", "número de ruedas", etc.

Ahora, quieres crear diferentes tipos de vehículos, como "Automóvil" y "Motocicleta", que comparten algunas características con la clase base "Vehículo", pero también tienen sus propias características únicas.

Usando herencia, puedes crear las clases "Automóvil" y "Motocicleta" como subclases de "Vehículo". Esto significa que heredarán todos los atributos y métodos de la clase base "Vehículo", y además, puedes agregar atributos y métodos específicos para cada tipo de vehículo.

Por ejemplo, un automóvil podría tener un atributo adicional como "número de puertas" y un método como "encender_motor()", mientras que una motocicleta podría tener un atributo adicional como "tipo de motor" y un método como "arrancar_motor()".

Este es un ejemplo de cómo la herencia puede ayudar a organizar y reutilizar código al modelar objetos del mundo real con estructuras de clases en la programación orientada a objetos.

HERENCIA

Imagina que tienes una clase base llamada "Publicación" que contiene los atributos y métodos comunes para cualquier tipo de publicación, como "título", "autor", "año de publicación", etc.

Ahora, quieres crear clases específicas para diferentes tipos de publicaciones, como "Libro" y "Revista". Ambos tipos de publicaciones comparten algunas características comunes, pero también tienen algunas características únicas.

Usando herencia, puedes crear las clases "Libro" y "Revista" como subclases de "Publicación". Esto significa que heredarán todos los atributos y métodos de la clase base

"Publicación", y además, puedes agregar atributos y métodos específicos para cada tipo de publicación.

Por ejemplo, la clase "Libro" podría tener atributos adicionales como "ISBN" y "número de páginas", mientras que la clase "Revista" podría tener atributos adicionales como "número de edición" y "tema".

Este es un ejemplo de cómo la herencia puede ayudar a organizar y reutilizar código al modelar objetos del mundo real con estructuras de clases en la programación orientada a objetos.

POLIMORFISMO

Imagina que tienes una clase base llamada "Vehículo" que contiene un método llamado "conducir()".

Ahora, quieres crear diferentes tipos de vehículos, como "Automóvil", "Motocicleta" y "Camión", cada uno con su propia implementación del método "conducir()".

- Para un automóvil, el método "conducir()" podría imprimir "El automóvil se está moviendo por la carretera".

- Para una motocicleta, el método "conducir()" podría imprimir "La motocicleta está recorriendo la autopista".

- Para un camión, el método "conducir()" podría imprimir "El camión avanza por la autopista".

Aunque cada clase tiene su propia implementación del método "conducir()", puedes llamar al mismo método "conducir()" en cada tipo de vehículo y obtener un comportamiento diferente en función del tipo de vehículo.

Esto es un ejemplo de polimorfismo, donde un método puede tener diferentes formas en diferentes clases, y la implementación específica del método se determina en tiempo de ejecución, dependiendo del tipo de objeto al que se está llamando.

En resumen, el polimorfismo te permite tratar diferentes tipos de objetos de manera uniforme a través de una interfaz común, lo que facilita la flexibilidad y la extensibilidad en tu código.

POLIMORFISMO

Imagina que tienes una clase base llamada "Publicación" que contiene un método llamado "mostrar_detalle()".

Ahora, quieres crear diferentes tipos de publicaciones, como "Libro" y "Revista", cada una con su propia implementación del método "mostrar_detalle()".

- Para un libro, el método "mostrar_detalle()" podría imprimir información específica del libro, como el título, el autor, el ISBN y el número de páginas.
- Para una revista, el método "mostrar_detalle()" podría imprimir información específica de la revista, como el título, el número de edición y el tema.

Aunque cada clase tiene su propia implementación del método "mostrar_detalle()", puedes llamar al mismo método "mostrar_detalle()" en cada tipo de publicación y obtener un detalle diferente en función del tipo de publicación.

Esto es un ejemplo de polimorfismo, donde un método puede tener diferentes formas en diferentes clases, y la implementación específica del método se determina en tiempo de ejecución, dependiendo del tipo de objeto al que se está llamando.

En resumen, el polimorfismo te permite tratar diferentes tipos de objetos de manera uniforme a través de una interfaz común, lo que facilita la flexibilidad y la extensibilidad en tu código.

INTERFACES DE CLASES

Imagina que estás diseñando un sistema de gestión de formas geométricas. Quieres crear una interfaz llamada "FormaGeometrica" que contenga métodos para calcular el área y el perímetro de diferentes formas geométricas, como círculos, rectángulos y triángulos.

1. Define una interfaz llamada "FormaGeometrica" que contenga los siguientes métodos abstractos:
    - calcular_area(): este método debe calcular y devolver el área de la forma geométrica.
    - calcular_perimetro(): este método debe calcular y devolver el perímetro de la forma geométrica.
2. Crea clases concretas que implementen la interfaz "FormaGeometrica" para diferentes formas geométricas, como "Circulo", "Rectangulo" y "Triangulo". Cada clase debe proporcionar una implementación para los métodos abstractos de la interfaz.
3. Crea instancias de las diferentes formas geométricas y llama a los métodos "calcular_area()" y "calcular_perimetro()" en cada una para verificar su correcto funcionamiento.

Este ejercicio te permitirá entender cómo usar interfaces para definir un conjunto común de métodos que deben ser implementados por clases específicas, proporcionando así una estructura consistente y coherente en tu sistema de gestión de formas geométricas.