Universidad Autónoma de Querétaro

Facultad de Informática

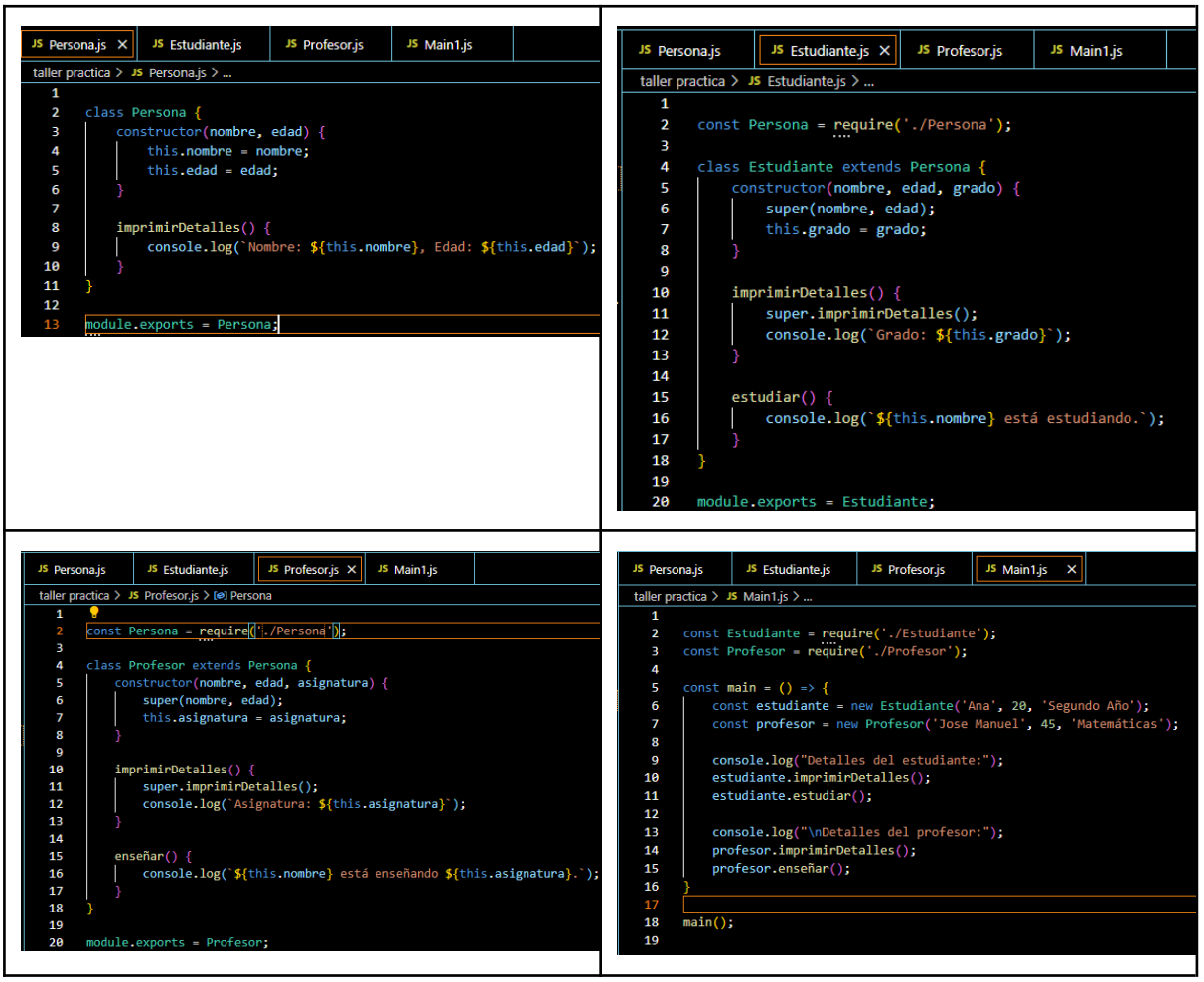
Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671
11/05/2024

Crear una clase 'Persona':

1. Define una clase Persona con propiedades como nombre, edad y método para imprimir los detalles de la persona.

Crear subclases:

2. Extiende la clase Persona para crear subclases como Estudiante, Profesor, etc. Cada subclase debe tener sus propias propiedades y métodos además de las heredadas de la clase Persona.



Crear una clase 'Libro':

1. Crea una clase Libro que tenga propiedades como título, autor, y método para imprimir los detalles del libro.

Agregar métodos:

2. Agrega métodos a la clase Libro para prestar y devolver el

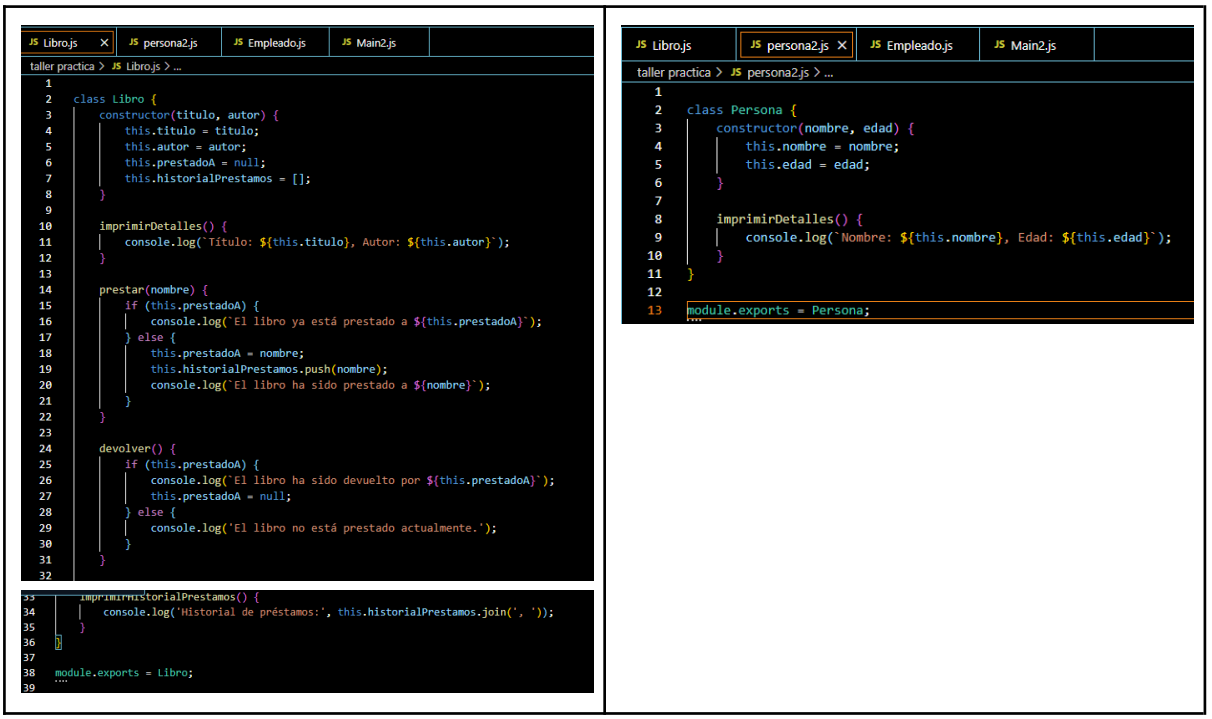
libro, manteniendo un registro de quién tiene prestado el libro en un array.

Implementar herencia:

3. Crea una clase Empleado que herede de la clase Persona, y añade propiedades específicas de un empleado, como salario y cargo.

Utilizar getters y setters:

4. Modifica alguna de las propiedades de las clases para que utilicen getters y setters, por ejemplo, para validar datos de entrada.



```
Libro.js | persona2.js | Empleado.js | Main2.js
taller practica > Libro.js > ...

class Libro {
    constructor(titulo, autor) {
        this.titulo = titulo;
        this.autor = autor;
        this.prestadoA = null;
        this.historialPrestamos = [];
    }

    imprimirDetalles() {
        console.log(`Título: ${this.titulo}, Autor: ${this.autor}`);
    }

    prestar(nombre) {
        if (this.prestadoA) {
            console.log(`El libro ya está prestado a ${this.prestadoA}`);
        } else {
            this.prestadoA = nombre;
            this.historialPrestamos.push(nombre);
            console.log(`El libro ha sido prestado a ${nombre}`);
        }
    }

    devolver() {
        if (this.prestadoA) {
            console.log(`El libro ha sido devuelto por ${this.prestadoA}`);
            this.prestadoA = null;
        } else {
            console.log('El libro no está prestado actualmente.');
        }
    }

    imprimirHistorialPrestamos() {
        console.log('Historial de préstamos:', this.historialPrestamos.join(', '));
    }
}

module.exports = Libro;
```

```
Libro.js | persona2.js | Empleado.js | Main2.js
taller practica > persona2.js > ...

class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    imprimirDetalles() {
        console.log(`Nombre: ${this.nombre}, Edad: ${this.edad}`);
    }
}

module.exports = Persona;
```

Universidad Autónoma de Querétaro

Facultad de Informática

Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671

11/05/2024

Libro.js | persona2.js | Empleado.js | Main2.js

taller practica > Empleado.js > ...

```
const Persona = require('./persona');

class Empleado extends Persona {
    constructor(nombre, edad, salario, cargo) {
        super(nombre, edad);
        this._salario = salario;
        this.cargo = cargo;
    }

    get salario() {
        return this._salario;
    }

    set salario(nuevoSalario) {
        if (nuevoSalario > 0) {
            this._salario = nuevoSalario;
        } else {
            console.log('El salario debe ser un valor positivo.');
        }
    }

    imprimirDetalles() {
        super.imprimirDetalles();
        console.log(`Salario: ${this.salario}, Cargo: ${this.cargo}`);
    }
}

module.exports = Empleado;
```

Libro.js | persona2.js | Empleado.js | Main2.js

taller practica > Main2.js > ...

```
const Libro = require('./Libro');
const Empleado = require('./Empleado');

const main = () => {
    // Crear y probar la clase Libro
    const libro = new Libro('Cien Años de Soledad', 'Gabriel García Márquez');
    libro.imprimirDetalles();
    libro.prestar('Juan');
    libro.imprimirHistorialPrestamos();
    libro.devolver();
    libro.imprimirHistorialPrestamos();

    // Crear y probar la clase Empleado
    const empleado = new Empleado('Ana', 30, 50000, 'Ingeniera');
    empleado.imprimirDetalles();
    empleado.salario = 60000; // Usar setter para actualizar el salario
    empleado.imprimirDetalles();
    empleado.salario = -5000; // Intentar establecer un salario negativo
}

main();
```

Crear una clase 'Rectángulo':

1. Define una clase Rectángulo con propiedades como ancho y alto, y métodos para calcular el área y el perímetro del rectángulo.

Crear una clase 'Círculo':

2. Crea una clase Círculo con propiedades como radio y métodos para calcular el área y la circunferencia del círculo.

Crear una clase 'FiguraGeometrica':

3. Define una clase abstracta FiguraGeometrica con métodos abstractos como calcularArea y calcularPerimetro, y luego implementa subclases como Rectángulo y Círculo que hereden de esta clase y proporcionan implementaciones concretas de esos métodos.

Aplicar polimorfismo:

4. Crea una función que tome un objeto de tipo FiguraGeometrica como parámetro y utilice sus métodos calcularArea y calcularPerimetro sin importar si el objeto es un rectángulo o un círculo.
5.

Universidad Autónoma de Querétaro

Facultad de Informática

Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671
11/05/2024

FigurasGeometricas.js

```
class FiguraGeometrica {
    constructor() {
        if (this.constructor === FiguraGeometrica) {
            throw new Error("No se puede instanciar una clase abstracta.");
        }
    }

    calcularArea() {
        throw new Error("Método abstracto 'calcularArea' no implementado.");
    }

    calcularPerimetro() {
        throw new Error("Método abstracto 'calcularPerimetro' no implementado.");
    }
}

module.exports = FiguraGeometrica;
```

Rectangulo.js

```
module.exports = Rectangulo;

const FiguraGeometrica = require('./figuraGeometrica');

class Rectangulo extends FiguraGeometrica {
    constructor(ancho, alto) {
        super();
        this.ancho = ancho;
        this.alto = alto;
    }

    calcularArea() {
        return this.ancho * this.alto;
    }

    calcularPerimetro() {
        return 2 * (this.ancho + this.alto);
    }
}

module.exports = Rectangulo;
```

Circulo.js

```
const FiguraGeometrica = require('./figuraGeometrica');

class Circulo extends FiguraGeometrica {
    constructor(radio) {
        super();
        this.radio = radio;
    }

    calcularArea() {
        return Math.PI * this.radio * this.radio;
    }

    calcularPerimetro() {
        return 2 * Math.PI * this.radio;
    }
}

module.exports = Circulo;
```

Main3.js

```
const Rectangulo = require('./Rectangulo');
const Circulo = require('./Circulo');

const mostrarDetalles = (figura) => {
    console.log(`Área: ${figura.calcularArea()}`);
    console.log(`Perímetro: ${figura.calcularPerimetro()}`);
};

const main3 = () => {
    const rectangulo = new Rectangulo(10, 5);
    const circulo = new Circulo(7);

    console.log('Detalles del rectángulo:');
    mostrarDetalles(rectangulo);

    console.log('\nDetalles del círculo:');
    mostrarDetalles(circulo);
};

main3();
```

Universidad Autónoma de Querétaro
Facultad de Informática
Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671
11/05/2024

Analiza el siguiente código en JavaScript y determina qué patrón de diseño se está utilizando y describirlos

```
class Database {
  constructor() {
    if (!Database.instance) {
      Database.instance = this;
    }
    return Database.instance;
  }

  query(sql) {
    console.log("Ejecutando consulta:", sql);
  }

const db1 = new Database();
const db2 = new Database();

console.log(db1 === db2); // Output: true
db1.query("SELECT * FROM users");
```

Utiliza el **Patrón de Diseño Singleton**, este patrón se asegura de que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.
Constructor: En el constructor de la clase Database, se verifica si Database.instance ya existe, si no existe, se asigna la instancia actual (this) a Database.instance, luego, el constructor retorna Database.instance.
Verificación de la instancia: Al crear dos instancias db1 y db2 de Database, ambas variables referencian la misma instancia debido a la lógica en el constructor. Esto se verifica con console.log(db1 === db2); que imprimirá true.
Método query: El método query es un método típico de la clase Database que simplemente imprime la consulta SQL que se le pasa como argumento.

```
class Logger {
  constructor() {
    this.logs = [];
  }

  log(message) {
    this.logs.push(message);
    console.log("Log registrado:", message);
  }

  static getInstance() {
    if (!Logger.instance) {
      Logger.instance = new Logger();
    }
    return Logger.instance;
  }
}

const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

console.log(logger1 === logger2); // Output: true
logger1.log("Error: No se puede conectar al servidor");
```

Utiliza el **Patrón de Diseño Singleton** en este caso la clase Logger tiene un método estático getInstance() que devuelve siempre la misma instancia de la clase Logger.
Constructor: El constructor de la clase Logger inicializa una propiedad logs como un array vacío.
Método log: Este método toma un mensaje como argumento, lo agrega al array logs y luego lo imprime en la consola.
Método estático getInstance(): Si no existe una instancia previa de

Universidad Autónoma de Querétaro
Facultad de Informática
Licenciatura en informática
Mireles Nieves Giovana Paola
Expediente 325671
11/05/2024

	Logger, crea una nueva instancia y la asigna a Logger.instance. Si ya existe una instancia, simplemente devuelve la instancia existente por lo cual esto garantiza que solo haya una instancia de Logger en toda la aplicación. **Uso del Singleton:** Al llamar a Logger.getInstance(), se obtiene la misma instancia de Logger en todos los lugares donde se llama, como se puede ver en la comparación logger1 === logger2, que devuelve true. Esto confirma que logger1 y logger2 se refieren a la misma instancia de Logger. **Llamada al método log:** Se llama al método log en logger1, agregando un mensaje al array logs y escribiendo el mensaje en la consola.
	Utiliza el **Patrón de Diseño Factory Method** que define una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán. **Clase User:** Tiene un constructor que toma un name como argumento y lo asigna a la propiedad name. **Tiene un método greet()** que imprime un saludo en la consola usando el nombre del usuario. **Clase UserFactory:**Tiene un método createUser(name) que toma un name como argumento y devuelve una nueva instancia de la clase User. **Uso del Factory Method:** Se crea una instancia de UserFactory llamada factory, se utilizan los métodos factory.createUser("Juan") y factory.createUser("Maria") para

	crear dos instancias de User, user1 y user2, respectivamente. **Invocación del método greet():** Se llama al método greet() en user1, que imprime "Hola, soy Juan". Se llama al método greet() en user2, que imprime "Hola, soy Maria".