Лабораторная работа № 7

Тема: Исключения и продвинутые паттерны ООП.

Цель: Научиться обрабатывать исключения в JavaScript и применять паттерны для создания объектов.

🔷 Вариант 1.

Задание 1. Обработка исключений.
Создайте функцию calculateSquareRoot(number), которая вычисляет квадратный корень из числа.
Используйте блок try...catch для обработки ошибок.
Если переданное значение не является числом или является отрицательным, выбросьте (throw) новую ошибку с информативным сообщением.
В блоке catch выведите сообщение об ошибке в консоль.
В блоке finally выведите сообщение "Операция завершена".

Задание 2. Паттерн "Фабрика".
Создайте "Фабрику" ShapeFactory для создания разных геометрических фигур: Circle и Square.
Создайте классы Circle (с полем radius) и Square (с полем side).
В ShapeFactory создайте метод createShape(type, size).
Если type равен 'circle', метод должен возвращать новый экземпляр Circle.
Если type равен 'square', должен возвращаться экземпляр Square.
Если type не подходит, выбросьте ошибку.

Задание 3. Применение паттернов и исключений.
Используйте "Фабрику" из Задания 2:
Создайте новый круг с радиусом 5.
Создайте новый квадрат со стороной 10.
Попытайтесь создать фигуру с некорректным типом (например, 'triangle') и обработайте эту ошибку с помощью try...catch.
Выведите в консоль информацию о созданных фигурах.

Вариант 2

Задание 1. Обработка исключений.
Создайте функцию processOrder(quantity), которая обрабатывает заказ.
Используйте блок try...catch для обработки ошибок.
Если переданное значение quantity не является числом, является отрицательным или равно нулю, выбросьте новую ошибку с информативным сообщением.

В блоке catch выведите сообщение об ошибке в консоль.
В блоке finally выведите сообщение "Обработка заказа завершена".

Задание 2. Паттерн "Одиночка".
Создайте класс AppSettings по паттерну "Одиночка". Этот класс должен гарантировать, что существует только один его экземпляр.
Создайте приватное статическое свойство #instance для хранения единственного экземпляра.
Сделайте конструктор приватным, чтобы запретить прямой вызов через new.
Создайте статический метод getInstance(), который будет возвращать #instance или создавать его, если он ещё не существует.
Класс должен иметь публичное свойство theme со значением по умолчанию 'light'.

Задание 3. Применение паттернов и исключений.
Используйте класс "Одиночка" из Задания 2:
Создайте первый экземпляр AppSettings с помощью getInstance() и измените его тему на 'dark'.
Попытайтесь создать второй экземпляр, снова вызвав getInstance(). Убедитесь, что второй экземпляр имеет ту же тему ('dark'), что и первый, и что оба ссылаются на один и тот же объект.
Попытайтесь создать экземпляр класса напрямую, используя new AppSettings(), и обработайте возникающую ошибку с помощью try...catch, чтобы показать, что это невозможно.

⚡ **Отчет должен содержать (см. образец):**
- номер и тему лабораторной работы;
- фамилию, номер группы студента и вариант задания;
- скриншоты окна VSC с исходным кодом программ;
- скриншоты с результатами выполнения программ;
- пояснения, если необходимо;
- выводы.

Отчеты в формате **pdf** отправлять на email:
colledge20education23@gmail.com

Шпаргалка по обработке исключений и продвинутым паттернам ООП

1. Обработка исключений

Основные конструкции

try...catch: используется для обработки ошибок в блоке кода.

Пример простого try...catch;

```
function divide(a, b) {
    try {
        if (b === 0) {
            throw new Error("Деление на ноль!");
        }
        return a / b;
    } catch (error) {
        console.error(error.message);
        return null; // Возврат null в случае ошибки
    }
}

console.log(divide(10, 2)); // 5
console.log(divide(10, 0)); // "Деление на ноль!"
```

2. Паттерн "Одиночка" (Singleton)

Паттерн "Одиночка" гарантирует, что у класса будет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.

Пример реализации;

```
class Logger {
    static #instance;

    constructor() {
        if (Logger.#instance) {
            return Logger.#instance; // Возврат
существующего экземпляра
        }
        Logger.#instance = this; // Сохранение нового
экземпляра
    }

    log(message) {
        console.log(`[LOG]: ${message}`);
    }

    static getInstance() {
```

```
        if (!Logger.#instance) {
            Logger.#instance = new Logger();
        }
        return Logger.#instance;
    }
}

// Использование паттерна "Одиночка"
const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

logger1.log("Первая запись."); // [LOG]: Первая запись.
console.log(logger1 === logger2); // true, оба ссылаются на
один экземпляр
```

3. Паттерн "Фабрика" (Factory)
Паттерн "Фабрика" предоставляет интерфейс для создания объектов, позволяя подклассам изменять тип создаваемого объекта.

Пример реализации;

```
class Shape {
    area() {
        throw new Error("Метод area() должен быть
переопределен");
    }
}

class Circle extends Shape {
    constructor(radius) {
        super();
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius ** 2;
    }
}

class Rectangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
```

```
    }

    area() {
        return this.width * this.height;
    }
}

class ShapeFactory {
    static createShape(type, ...params) {
        switch (type) {
            case 'circle':
                return new Circle(...params);
            case 'rectangle':
                return new Rectangle(...params);
            default:
                throw new Error('Неизвестный тип фигуры');
        }
    }
}

// Использование фабрики
const circle = ShapeFactory.createShape('circle', 5);
console.log(circle.area()); // 78.53981633974483

const rectangle = ShapeFactory.createShape('rectangle', 4,
6);
console.log(rectangle.area()); // 24
```

4. Паттерн "Абстрактная фабрика"
Паттерн "Абстрактная фабрика" предоставляет интерфейс для создания семейств связанных объектов.

Пример реализации;

```
class Animal {
    speak() {
        throw new Error("Метод speak() должен быть
переопределен");
    }
}

class Dog extends Animal {
    speak() {
        return "Гав!";
```

```
    }
}

class Cat extends Animal {
    speak() {
        return "Мяу!";
    }
}

class AnimalFactory {
    static createAnimal(type) {
        switch (type) {
            case 'dog':
                return new Dog();
            case 'cat':
                return new Cat();
            default:
                throw new Error('Неизвестный тип
животного');
        }
    }
}

// Использование фабрики
const dog = AnimalFactory.createAnimal('dog');
console.log(dog.speak()); // Гав!

const cat = AnimalFactory.createAnimal('cat');
console.log(cat.speak()); // Мяу!
```