ПМ3 Разработка модулей ПО.

РО 3.1 Понимать и применять принципы объектно-ориентированного и асинхронного программирования.

Тема 1. Введение в ООП.

Лекция 7. Исключения и продвинутое применение ООП.

Цель занятия:

Познакомиться с концепциями обработки исключений в JavaScript и с продвинутыми паттернами объектно-ориентированного программирования.

Учебные вопросы:

1. Обработка исключений.
2. Продвинутые паттерны ООП.

1. Обработка исключений.

Обработка исключений — это механизм, который позволяет вашему коду gracefully handle errors.

Вместо того, чтобы программа полностью завершалась при возникновении проблемы, вы можете "поймать" эту ошибку и предпринять определённые действия.

Понятие исключения (Exception) и ошибки (Error)

В JavaScript ошибка (Error) — это объект, который возникает, когда что-то идёт не так. Исключение (Exception) — это синоним ошибки. Исключение — это событие, которое прерывает нормальный ход выполнения программы.

JavaScript имеет встроенные типы ошибок: SyntaxError, TypeError, ReferenceError и другие. Вы можете также создавать свои собственные типы ошибок.

Синтаксис try...catch...finally

- **try**: Блок кода, который вы хотите проверить на ошибки. Если в этом блоке возникает исключение, выполнение немедленно переходит в блок catch.
- **catch**: Блок, который "ловит" ошибку из блока try. Он принимает один аргумент (обычно называемый error или e), который содержит объект ошибки. В этом блоке вы можете обработать ошибку.
- **finally**: Блок, который выполняется всегда, независимо от того, была ли ошибка или нет. Он полезен для выполнения кода, который должен быть выполнен в любом случае, например, для закрытия файлов или сетевых соединений.

Пример:

```
try {
  // Код, который может выбросить ошибку
  let result = 10 / a;
  console.log(result);
} catch (error) {
  // Код, который выполняется, если произошла ошибка
  console.error("Произошла ошибка:", error.message);
} finally {
  // Код, который выполняется всегда
  console.log("Операция завершена.");
}
```

Генерация собственных ошибок с помощью throw.

Вы можете "выбросить" (throw) свою собственную ошибку, чтобы сигнализировать о некорректных данных или условиях. Это делает ваш код более предсказуемым.

Оператор throw принимает любое значение, но обычно используют объекты Error для стандартизации.

new Error() создаёт новый объект ошибки с сообщением.

Пример:

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Деление на ноль невозможно!"); // Выброс собственной ошибки
  }
  return a / b;
}

try {
  const result = divide(10, 0);
  console.log(result);
} catch (error) {
  console.error("Ошибка:", error.message); // Ошибка: Деление на ноль невозможно!
}
```

Зачем это нужно:

- Предотвращение аварийного завершения программы.
- Контроль и логирование ошибок.
- Возможность корректного восстановления работы после ошибки.

2. Продвинутые паттерны ООП.

Что такое паттерны проектирования?

Паттерны проектирования — это обобщенные решения часто встречающихся проблем в разработке программного обеспечения.

Они представляют собой проверенные временем подходы, которые помогают разработчикам создавать гибкие и поддерживаемые системы.

Паттерны не являются готовым кодом, а служат шаблонами, которые можно адаптировать под конкретные задачи.

Зачем нужны паттерны проектирования?

- Упрощение разработки: Паттерны помогают разработчикам избежать "изобретения велосипеда" и предлагают готовые решения.
- Повышение читаемости и поддержки кода: Использование паттернов делает код более понятным для других разработчиков.
- Гибкость и расширяемость: Паттерны способствуют созданию кода, который проще модифицировать и расширять.
- Снижение связности: Паттерны помогают уменьшить зависимость между компонентами системы, что облегчает тестирование и модификацию.

Паттерн "Фабрика" — это создающий паттерн, который предоставляет интерфейс для создания объектов, но позволяет подклассам изменять тип создаваемого объекта.

Вместо того чтобы использовать оператор **new**, разработчик делегирует создание объектов фабрике.

Зачем использовать паттерн "Фабрика"?

- Инкапсуляция логики создания объектов: Фабрика отделяет создание объекта от его использования.
- Поддержка различных типов объектов: Можно легко добавлять новые типы объектов, не изменяя клиентский код.
- Упрощение тестирования: Замена реальных объектов на заглушки или моки становится проще.

Виды фабрик.

- Фабричный метод (Factory Method): Определяет интерфейс для создания объектов, но позволяет подклассам изменять тип создаваемого объекта.
- Абстрактная фабрика (Abstract Factory): Предоставляет интерфейс для создания семейств связанных или зависимых объектов.

Пример, где мы реализуется фабрика для создания различных видов автомобилей.

```
// Основной класс автомобиля
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
    }

    getInfo() {
        return `${this.make} ${this.model}`;
    }
}
```

```
// Подклассы автомобилей
class Sedan extends Car {
    constructor(make, model) {
        super(make, model);
    }
}

class SUV extends Car {
    constructor(make, model) {
        super(make, model);
    }
}
```

```
// Фабрика автомобилей
class CarFactory {
    static createCar(type, make, model) {
        switch (type) {
            case 'sedan':
                return new Sedan(make, model);
            case 'suv':
                return new SUV(make, model);
            default:
                throw new Error('Unknown car type');
        }
    }
}
```

```
// Использование фабрики
const sedan = CarFactory.createCar('sedan', 'Toyota', 'Camry');
const suv = CarFactory.createCar('suv', 'Ford', 'Explorer');

console.log(sedan.getInfo()); // Toyota Camry
console.log(suv.getInfo());   // Ford Explorer
```

Заключение.

Паттерн "Фабрика" упрощает процесс создания объектов, обеспечивая гибкость и расширяемость системы.

Он позволяет разработчикам организовать код таким образом, чтобы логика создания объектов была отделена от их использования, что делает систему более поддерживаемой и понятной.

Паттерн "Одиночка" (Singleton) — это порождающий паттерн проектирования, который гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру.

Этот паттерн часто используется для управления ресурсами, такими как соединения с базами данных или конфигурационные параметры.

Зачем использовать паттерн "Одиночка"?

- Управление ресурсами: Когда создание нескольких экземпляров класса может привести к конфликтам или утечкам ресурсов.
- Глобальный доступ: Обеспечение единой точки доступа для данных или состояния, который может быть использован в различных частях приложения.
- Упрощение управления состоянием: Легче отслеживать состояние приложения, когда оно хранится в одном месте.

Принципы работы.

- Закрытый конструктор: Конструктор класса должен быть закрыт (или приватен), чтобы предотвратить создание экземпляров класса извне.
- Статический метод: Необходим статический метод, который будет отвечать за создание и возврат единственного экземпляра класса.
- Ленивая инициализация: Экземпляр создается только тогда, когда он действительно нужен.

Пример, где создается класс Database, который будет использоваться для управления подключением к базе данных.

```
class Database {
    // Приватное статическое поле для хранения экземпляра
    static #instance;
    // Приватный конструктор
    constructor() {
        if (Database.#instance) {
            return Database.#instance; // Возврат единственного экземпляра
        }
        this.connectionString = 'localhost:5432/mydb';
        Database.#instance = this; // Сохранение экземпляра
    }
```

```
    // Статический метод для получения экземпляра
    static getInstance() {
        if (!Database.#instance) {
            Database.#instance = new Database(); // Создание нового экземпляра
        }
        return Database.#instance;
    }
    // Метод для выполнения запроса
    query(sql) {
        console.log(`Executing query: ${sql} on ${this.connectionString}`);
    }
}
```

```
// Использование паттерна "Одиночка"
const db1 = Database.getInstance();
const db2 = Database.getInstance();

db1.query('SELECT * FROM users');

console.log(db1 === db2); // true, оба переменные ссылаются на один и тот же экземпляр
```

Пояснения к примеру:

- Приватный конструктор: Конструктор класса Database закрыт, чтобы предотвратить создание новых экземпляров извне.
- Статическое поле: Поле #instance используется для хранения единственного экземпляра класса.
- Статический метод getInstance(): Этот метод проверяет, существует ли уже экземпляр. Если нет, он создает его и возвращает.

Применение паттерна "Одиночка":

- Конфигурационные файлы: Для хранения и управления настройками приложения.
- Логирование: Для создания единой точки доступа к логированию, где все логи будут записываться в одно место.
- Управление ресурсами: Для управления соединениями с базами данных или внешними сервисами, где необходимо контролировать количество открытых соединений.

Заключение.

Паттерн "Одиночка" позволяет создать единственный экземпляр класса и предоставляет к нему глобальный доступ.

Это упрощает управление состоянием и ресурсами в приложении, однако его использование должно быть обоснованным, поскольку чрезмерное применение может привести к затруднениям в тестировании и усложнению кода.

Контрольные вопросы:

- Какое ключевое отличие между try и catch?
- Объясните, что делает блок finally и когда он выполняется.
- Зачем нужен паттерн "Фабрика"? В каком случае его лучше использовать, чем прямой вызов конструктора?
- Почему паттерн "Одиночка" может быть полезен в веб-приложении?
- Как можно создать свою собственную ошибку и выбросить её? Приведите пример.

Домашнее задание:

https://ru.hexlet.io/courses/

Материалы лекций:

https://github.com/ShViktor72/Education2025

Обратная связь:

colledge20education23@gmail.com