

## **ПМ3 Разработка модулей ПО.**

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

# **Тема 1. Введение в ООП.**

## **Лекция 1. Основы ООП и инкапсуляция**

# **Цели занятия:**

- Ознакомиться с базовыми понятиями объектно-ориентированного программирования.**
- Рассмотреть сущность и назначение инкапсуляции в JavaScript.**

# **Учебные вопросы:**

- 1. Понятие объектно-ориентированного программирования (ООП).**
- 2. Основные принципы ООП (инкапсуляция, наследование, полиморфизм, абстракция).**
- 3. Инкапсуляция как основной принцип ООП.**

# **1. Понятие объектно-ориентированного программирования (ООП).**

Объектно-ориентированное программирование (ООП) — это парадигма программирования, в которой основными строительными элементами программы являются объекты, объединяющие в себе данные (свойства) и методы (функции для работы с этими данными).

# Парадигма программирования — что это?

Простыми словами: Парадигма программирования — это способ думать о том, как писать программы.

Парадигма отвечает на вопросы:

- Как организовать код?
- Как решать задачи?
- Как думать о программе?

# Основные парадигмы программирования

## 1. Объектно-ориентированное программирование (ООП)

Статус: Самая популярная парадигма

Что это: Программа состоит из объектов — «коробочек» с данными и действиями. Как в реальном мире: у машины есть цвет, скорость и она может ехать, тормозить.

Принцип: Всё — объекты, которые взаимодействуют друг с другом.

Популярные языки: Java, C#, Python, C++, TypeScript, JS

## 2. Функциональное программирование

Статус: Быстро растущая популярность

Что это: Программа — набор функций, которые берут данные и возвращают новые, ничего не изменяя. Как математические функции:  $f(x) = x + 2$ .

Принцип: Никаких изменений, только создание нового из старого.

Особенно востребовано в:

- Веб-разработке (React, Redux)
- Обработке больших данных
- Машинном обучении

Популярные языки: JavaScript (функциональный стиль), Python, Scala, F#



### 3. Императивное/Процедурное

Статус: Остается важным для системного программирования

Что это: Программа — список команд, выполняемых по порядку. Как инструкция: «сначала сделай это, потом то, затем вон то».

Принцип: Пошаговые инструкции для компьютера.

Популярные языки: C, Go, Rust

## 4. Событийно-ориентированное

Статус: Критически важно для современных интерфейсов

Что это: Программа ждет событий (клик мыши, нажатие клавиши) и реагирует на них. Как охранник, который реагирует на сигнал тревоги.

Принцип: «Случилось событие — выполни действие».

Доминирует в:

- Веб-разработке
- Мобильных приложениях
- Desktop-приложениях

Популярные языки: JavaScript, C#, Python

## 5. Реактивное программирование

Статус: Новый тренд

Что это: Программа автоматически реагирует на изменения данных. Как Excel — изменил одну ячейку, все связанные пересчитались сами.

Принцип: «Данные изменились — все зависящее от них обновилось автоматически».

Растущая популярность для:

- Асинхронных приложений
- Приложений реального времени
- Микросервисов

Популярные инструменты: RxJS, React, Spring WebFlux

# Главный тренд: Мультипарадигменность

Современные языки поддерживают различные парадигмы программирования, включая объектно-ориентированное, императивное, процедурное и функциональное программирование.

Самые востребованные мультипарадигменные языки:

- Python — ООП + функциональное + процедурное
- JavaScript/TypeScript — ООП + функциональное + событийное
- C# — ООП + функциональное + событийное
- Rust — процедурное + функциональное + ООП

## Ключевые идеи ООП:

- Программа моделирует предметную область с помощью объектов.
- Каждый объект отражает реальный или абстрактный элемент: «студент», «автомобиль», «банк».
- Взаимодействие объектов строится через обмен сообщениями (вызов методов).

## Основные преимущества ООП:

- Лучшая структура кода, его модульность.
- Возможность повторного использования кода (через классы и наследование).
- Упрощение сопровождения и масштабирования программ.
- Естественное моделирование реальных систем.

## Базовые сущности ООП:

- Класс — шаблон/описание объекта (например, «Автомобиль»).
- Объект (экземпляр класса) — конкретный представитель класса (например, «BMW X5»).
- Свойства — характеристики объекта (цвет, пробег).
- Методы — действия объекта (ехать, тормозить).  
Метод – функция, объявленная внутри класса.

## Пример:

```
1  class Car {
2      constructor(model, color) {
3          this.model = model;
4          this.color = color;
5      }
6      drive() {
7          console.log(`${this.model} едет...`);
8      }
9  }
10
11  const myCar = new Car("Toyota", "red");
12  myCar.drive(); // Toyota едет...
```



## Объявление класса.

- В JavaScript классы объявляются с помощью ключевого слова **class**:
- **constructor(...)** — специальный метод, который вызывается при создании нового объекта. В нём задаются начальные значения свойств.
- Методы класса объявляются без слова **function**.

## Пример:

```
class User {  
  role = "guest"; // свойство по умолчанию  
  
  constructor(name, age) {  
    this.name = name; // свойство объекта  
    this.age = age;   // свойство объекта  
  }  
  
  sayHello() {          // метод объекта  
    console.log(`Привет, меня зовут ${this.name}`);  
  }  
}
```

## Создание экземпляров класса.

Для создания объекта используется ключевое слово `new`:

```
// Создадим два экземпляра класса
const user1 = new User("Иван", 20);
const user2 = new User("Анна", 25);

// вызов метода у экземпляра
user1.sayHello(); // Привет, меня зовут Иван
user2.sayHello(); // Привет, меня зовут Анна
```

Каждый вызов `new User(...)` создаёт новый экземпляр класса `User` со своими уникальными данными (`name`, `age`).

```
// значение свойств можно изменять  
user1.role = "Admin"  
user2.age = 26  
  
console.log(user1.role); // Admin  
console.log(user2.role); // guest  
console.log(user2.age); // 26
```

- **Класс** — шаблон, описание объекта (например, «чертёж дома»).
- **Экземпляр класса** (объект) — конкретный объект, созданный по этому шаблону (например, «построенный дом»).
- **Свойства** — характеристики объекта (например, имя, возраст).
- **Методы** — действия объекта (например, sayHello).
- Свойства в конструкторе нужны, если их значения задаются при создании объекта (например, имя, возраст).
- Свойства вне конструктора — это значения по умолчанию, которые будут одинаковыми для всех новых объектов, если их не изменить.

## **Вывод:**

ООП — это подход к программированию, в котором программа строится как система взаимодействующих объектов.

Такой подход облегчает понимание, разработку и поддержку сложных программных систем.

## 2. Основные принципы ООП.

Объектно-ориентированное программирование строится на четырёх фундаментальных принципах:

- Инкапсуляция
- Наследование
- Полиморфизм
- Абстракция\*

# Инкапсуляция

Определение: сокрытие деталей реализации объекта и предоставление доступа к данным только через определённый интерфейс.

- Данные объекта защищены от прямого изменения.
- Управление доступом идёт через методы (геттеры/сеттеры).
- Повышает безопасность и предсказуемость работы кода.



## Простыми словами:

Инкапсуляция = **спрятать внутренности**, показать только нужное.

Как в машине: вы нажимаете на педаль газа, но не знаете, что происходит с двигателем внутри. И это правильно — вам не нужно знать все детали!

## Зачем это нужно?

- ✓ Защита от поломок — никто случайно не испортит важные данные
- ✓ Простота использования — не нужно знать сложные детали
- ✓ Безопасность — важная информация скрыта от посторонних

пример — Кошелек:

```
1  class Wallet {
2      // СКРЫТО - никто не может трогать (# означает private)
3      #money = 100;
4
5      // ОТКРЫТО - можно использовать
6      addMoney(amount) {
7          this.#money = this.#money + amount;
8      }
9
10     getMoney() {
11         return this.#money;
12     }
13 }
```

## Как работает:

```
const myWallet = new Wallet();
```

```
//  МОЖНО
```

```
myWallet.addMoney(50);           // добавить деньги
```

```
const howMuch = myWallet.getMoney(); // узнать сколько денег
```

```
console.log(howMuch);           // выведет: 150
```

```
//  НЕЛЬЗЯ
```

```
// myWallet.#money = 999;        // Ошибка! Переменная скрыта
```

# Наследование

Определение: механизм, позволяющий создавать новые классы на основе уже существующих.

Новый класс (потомок) перенимает свойства и методы базового класса (родителя).

Обеспечивает повторное использование кода и расширяемость.

## Простыми словами:

Наследование = создать новую вещь на основе старой + добавить что-то свое.

Как в семье: дети получают черты родителей, но у них есть и свои особенности!

Зачем это нужно?

- ✓ Не повторяться — используем уже готовое
- ✓ Добавлять новое — расширяем возможности
- ✓ Экономить время — не писать один и тот же код заново

## Пример:

```
// РОДИТЕЛЬ - общие черты всех животных
class Animal {
    constructor(name) {
        this.name = name;
    }



    eat() {
        console.log(this.name + " кушает");
    }

    sleep() {
        console.log(this.name + " спит");
    }
}
```

```
// РЕБЕНОК - наследует все от Animal + добавляет свое
class Dog extends Animal {
    bark() {
        console.log(this.name + " лает: Гав-гав!");
    }
}
```

```
// РЕБЕНОК - наследует все от Animal + добавляет свое
class Cat extends Animal {
    meow() {
        console.log(this.name + " мяукает: Мяу!");
    }
}
```

## Как работает:

```
const myDog = new Dog("Бобик");  
const myCat = new Cat("Мурка");  
  
//  У собаки есть ВСЕ методы от Animal  
myDog.eat();    // "Бобик кушает"  
myDog.sleep();  // "Бобик спит"  
myDog.bark();   // "Бобик лает: Гав-гав!" (своя способность)  
  
//  У кошки тоже есть ВСЕ методы от Animal  
myCat.eat();    // "Мурка кушает"  
myCat.sleep();  // "Мурка спит"  
myCat.meow();   // "Мурка мяукает: Мяу!" (своя способность)
```



# В чем смысл наследования?

БЕЗ наследования — пришлось бы повторять код:

```
class Dog {  
    eat() { console.log("кушает"); }      // повторяем  
    sleep() { console.log("спит"); }      // повторяем  
    bark() { console.log("гав"); }  
}  
  
class Cat {  
    eat() { console.log("кушает"); }      // повторяем снова!  
    sleep() { console.log("спит"); }      // повторяем снова!  
    meow() { console.log("мяу"); }  
}
```

С наследованием — пишем общее один раз, добавляем только новое!

# Полиморфизм

Определение: способность объектов с одинаковым интерфейсом вести себя по-разному в зависимости от реализации.

Один и тот же метод может выполнять разные действия у разных классов.

Упрощает работу с группой объектов, обеспечивая гибкость.

## Простыми словами:

Полиморфизм = одна команда — разные действия.

Как кнопка "Play": на телефоне играет музыку, на телевизоре — фильм, в игре — начинает игру!

Зачем это нужно?

- ✓ Одинаковый интерфейс — не нужно запоминать разные команды
- ✓ Гибкость — можем работать с разными объектами одинаково
- ✓ Простота — один метод для всех

## Пример:

```
// РОДИТЕЛЬ
class Animal {
  |   constructor(name) {
  |   |   this.name = name;
  |   }
  |   makeSound() {console.log(this.name + " издает звук");}
}

// ДЕТИ - переопределяют makeSound() по-своему
class Dog extends Animal {
  |   makeSound() {console.log(this.name + " лает: Гав-гав!");}
}

class Cat extends Animal {
  |   makeSound() {console.log(this.name + " мяукает: Мяу!");}
}
```

```
// Создаем разных животных
const bobik = new Dog("Бобик")
const murka = new Cat("Мурка")

// ✅ МАГИЯ! Одна команда - разные звуки
bobik.makeSound(); // одинаковый вызов для всех!
murka.makeSound(); // одинаковый вызов для всех!

// Результат:
// "Бобик лает: Гав-гав!"
// "Мурка мяукает: Мяу!"
```

## **Абстракция.**

Определение: выделение только значимых характеристик объекта и скрывание второстепенных деталей.

Помогает работать с системой на более высоком уровне, не погружаясь в сложность реализации.

В JS абстракция чаще реализуется через классы и интерфейсные соглашения (условные контракты).

## **В литературе встречаются два подхода:**

Одни авторы говорят о трёх принципах (инкапсуляция, наследование, полиморфизм).

Другие выделяют ещё и абстракцию как четвёртый принцип.

На практике абстракция тесно связана с инкапсуляцией и обычно рассматривается в её контексте, как более концептуальный подход (какие свойства и методы у объекта вообще должны быть).

## Коротко:



Инкапсуляция = Прятать важное.

Как в сейфе: деньги внутри (private), доступ через окошко кассира (public методы).



Наследование = Получить от родителей + добавить свое.

Как дети в семье: глаза от мамы, нос от папы + свой характер.



Полиморфизм = Одна команда — разные действия.

Как кнопка "Play": на телефоне — музыка, на ТВ — фильм, в игре — старт.

## Еще короче:

ООП = как конструктор LEGO 

- Инкапсуляция: детали защищены от поломки.
- Наследование: делаем новые детали на основе старых.
- Полиморфизм: одинаковые разъемы — разные возможности.



# 3. Инкапсуляция как основной принцип ООП.

Инкапсуляция — это механизм ООП, который объединяет данные и методы для работы с ними в одном объекте и скрывает внутреннюю реализацию от внешнего мира.

Объект предоставляет только публичный интерфейс для взаимодействия, а детали остаются недоступными напрямую.

## Ключевые термины:

- Интерфейс объекта — набор методов и свойств, доступных пользователю объекта.
- Публичные свойства/методы (**public**) — открытая часть объекта, к которой можно обращаться извне.
- Приватные свойства/методы (**private**) — скрытая часть объекта, недоступная напрямую. В JavaScript приватные поля обозначаются символом #.
- Геттер (**getter**) — метод для получения значения приватного свойства. В JS создаётся через ключевое слово `get`.
- Сеттер (**setter**) — метод для изменения значения приватного свойства с проверкой или дополнительной логикой. В JS создаётся через ключевое слово `set`.

# Зачем нужна инкапсуляция?

- Безопасность данных — защита от неконтролируемого изменения состояния.
- Удобство — пользователю важно «что делает объект», а не «как он это делает».
- Гибкость — внутреннюю реализацию можно изменить без изменения интерфейса.
- Поддерживаемость — программы становятся более структурированными и читаемыми.

# Реализация инкапсуляции в JavaScript

## 1. Приватные поля в классах (ES2020+):

```
1  class User {
2      #password; // приватное свойство
3
4      constructor(name, password) {
5          this.name = name;
6          this.#password = password;
7      }
8
9      checkPassword(pwd) {
10         return this.#password === pwd;
11     }
12 }
13
14 const u = new User("Ivan", "qwerty");
15 console.log(u.checkPassword("qwerty")); // true
16 // console.log(u.#password); // Ошибка! Поле приватное
```

## 2. Геттеры и сеттеры (контролируемый доступ):

```
1  class Car {
2      #mileage = 0;
3
4      get mileage() {          // геттер
5          return this.#mileage;
6      }
7
8      set mileage(value) { // сеттер
9          if (value >= this.#mileage) {
10             this.#mileage = value;
11          }
12      }
13  }
14
15  const c = new Car();
16  c.mileage = 100;          // вызов сеттера
17  console.log(c.mileage); // вызов геттера -> 100
```

Имя геттера и сеттера обычно совпадает с названием свойства, к которому они относятся.

## Итог:

 В JavaScript современным способом реализации инкапсуляции являются приватные поля (**#**) и геттеры/сеттеры.

 Геттер — метод, который возвращает значение приватного свойства.

 Сеттер — метод, который изменяет значение приватного свойства.

 Геттеры и сеттеры объявляются через **get** и **set**.

 Имя геттера и сеттера обычно совпадает с названием свойства, к которому они относятся.

 Геттер и сеттер должны называться одинаково

 У геттера нет параметров, у сеттера — один.

# Контрольные вопросы:

Что такое парадигма программирования?

Назовите основные парадигмы программирования.

В чём суть объектно-ориентированного программирования?

Что такое класс в JavaScript?

Что такое экземпляр класса (объект)?

Для чего используется метод constructor?

Как создаётся новый объект по классу?

Чем отличаются свойства, объявленные в конструкторе, от свойств, объявленных вне конструктора?

Какой ключевой объект внутри класса указывает на текущий экземпляр?

Как объявляется метод класса в JavaScript?

Назовите четыре основных принципа ООП.

Дайте определение инкапсуляции.

Как реализовать приватные данные в JavaScript (назовите не менее двух способов)?

Для чего нужны геттеры и сеттеры?

# Домашнее задание:

1. <https://ru.hexlet.io/courses/js-introduction-to-oop>

1 О курсе

Знакомимся с курсом, его структурой, целями и задачами

2 Инкапсуляция

Знакомимся с объединением данных и функций в одной структуре

2. Повторить материал лекции.



# **Материалы лекций:**

<https://github.com/ShViktor72/Education2025>

# **Обратная связь:**

colledge20education23@gmail.com