

# Цели занятия

- узнать что такое парадигма и ООП;
- разобраться с базовым синтаксисом классов в JavaScript;
- рассмотреть некоторые возможности классов;

## Полезные материалы

- [Знакомство с ООП на примерах](#)
- [Базовый синтаксис классов](#)

## Что такое парадигма?

Парадигма программирования — это набор идей и понятий, которые определяют стиль написания программ и подход к программированию. Существует довольно много различных парадигм, таких как функциональное программирование, структурное, объектно-ориентированное и реактивное. У каждой из них есть определенные правила и ограничения, поэтому некоторые задачи может быть удобнее решать с помощью функционального программирования, а другие — с помощью объектно-ориентированного.

В рамках данного занятия мы поверхностно познакомимся с объектно-ориентированным программированием, но далее в курсе будем использовать функциональное и реактивное.

## Что такое ООП?

Объектно-ориентированное программирование (сокр. **ООП**) — подход к программированию, основанный на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

При объектно-ориентированной разработке мы описываем происходящее на уровне объектов, которые создаются, меняют свои свойства, взаимодействуют друг с другом и (в случае браузера) со страницей, в общем, живут.

Например, «пользователь», «меню», «форма»... При объектно-ориентированном подходе каждый объект должен представлять собой интуитивно понятную сущность, у которой есть методы и данные.

## ООП — не просто коллекции

В JavaScript объекты часто используются просто как коллекции.

Например, встроенный объект Math содержит функции (Math.sin, Math.pow, ...) и данные (константа Math.PI).

При таком использовании объектов мы не можем сказать, что применён объектно-ориентированный подход. В частности, никакую «единую сущность» Math из себя не представляет, это просто коллекция независимых функций с общим префиксом Math.

Но мы уже вскользь видели пример в ООП-стиле:

```
function User(name) {
  this.sayHi = function () {
    alert(`Привет, я ${name}!`);
  };
}

let vasya = new User("Вася"); // создали пользователя
let yury = new User("Юрий"); // И ещё одного

// Оба пользователя умеют здороваться
vasya.sayHi(); // Привет, я Вася!
yury.sayHi(); // Привет, я Юрий!
```

Здесь мы видим ярко выраженную сущность – User (посетитель), у которого есть метод sayHi и на основе которого мы можем создать несколько объектов. Используя терминологию ООП, такие конструкторы часто называют классами, то есть можно сказать "класс User".

## Классы в JavaScript

Код из примера выше был написан в функциональном стиле, но современный JavaScript имеет отдельный синтаксис для того, чтобы писать код в ООП-стиле.

Базовый синтаксис выглядит так:

```
class MyClass {  
    // методы класса  
    constructor() { ... }  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```

Затем используйте вызов `new MyClass()` для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод `constructor()`, в нём мы можем инициализировать объект.

Давайте перепишем пример с пользователем в ООП-стиле:

```
class User {  
    constructor(name) {  
        this.name = name;  
    }  
}
```

```
sayHi() {  
    alert(`Привет, я ${this.name}!`);  
}  
}  
  
// Использование:  
let vasya = new User("Вася");  
let yury = new User("Юрий");  
  
vasya.sayHi();  
yury.sayHi();
```

Поначалу может показаться, что код стал только сложнее, но в этом подходе таится много возможностей, часть из которых мы разберем ниже.

## Геттеры и сеттеры

В классах можно создавать вычисляемые свойства, а также писать геттеры и сеттеры, которые будут вызваны когда мы хотим получить значение какого-то свойства или записать новое значение.

На примере:

```
class User {  
    constructor(firstName, lastName) {  
        // вызывает сеттер  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    get firstName() {  
        return this._firstName;  
    }  
}
```

```
// Создаём новое свойство, используя существующие
get fullName() {
    return `${this.firstName} ${this.lastName}`;
}

// Выполняем проверку перед записью свойства
set firstName(value) {
    if (value.length < 4) {
        alert("Имя слишком короткое.");
        return;
    }

    this._firstName = value;
}
}

let user = new User("Иван", "Петров");

alert(user.name); // Иван
alert(user.fullName); // Иван Петров

user = new User(""); // Имя слишком короткое.
```

## Свойства классов

В приведённом выше примере у класса User были только методы. Давайте добавим свойство:

```
class User {
    name = "Аноним";
```

```
sayHi() {  
    alert(`Привет, ${this.name}!`);  
}  
}  
  
new User().sayHi(); // Привет, Аноним!
```

Свойство `name` создаётся оператором `new` перед запуском конструктора и это именно свойство объекта.

## Наследование классов

Наследование классов – это способ расширения одного класса другим классом. Таким образом, мы можем добавить новый функционал к уже существующему.

Давайте представим, что у нас может быть два типа пользователей — обычный и VIP. И у второго типа пользователя должны быть все методы от обычного пользователя, но вдобавок ещё и несколько своих. Вместо того чтобы дублировать код, мы можем использовать наследование:

```
class User {  
    constructor(name) {  
        this.name = name;  
    }  
  
    sayHi() {  
        alert(`Привет, я ${this.name}!`);  
    }  
}  
  
class VIPUser extends User {  
    inform() {
```

```
        alert(`Я ${this.name} и я VIP-пользователь!`);
    }
}

let vasya = new User("Вася");
let yury = new VIPUser("Юрий");

// У VIP-пользователя есть метод обычного
yury.sayHi(); // Привет, я Юрий!

// А также есть свой доп. метод
yury.inform(); // Я Юрий и я VIP-пользователь!
vasya.inform(); // Ошибка, метода нет
```

Это очень простой пример, но в реальных задачах наследование позволяет масштабировать код, создавать, к примеру, пользователей с разными возможностями и не дублировать код. Например, на учебной платформе могут быть пользователи с типами “преподаватель”, “слушатель”, “администратор” и у каждого из них будет свой набор возможностей.