

ПМЗ Разработка модулей ПО.

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

Тема 1. Введение в ООП.

Лекция 3. Функции-конструкторы.

Цель занятия:

**Показать эволюцию способов
создания объектов в JS, объяснить
роль функций-конструкторов и
оператора new.**

Учебные вопросы:

- 1. Фабричные функции: назначение и примеры.**
- 2. Функция-конструктор: понятие, правила объявления.**
- 3. Ограничения функций-конструкторов.**
- 4. Встроенные конструкторы (Array, Date, Function и др.).**

1. Фабричные функции: назначение и примеры.

Фабричная функция (factory function) — это обычная функция, которая возвращает объект.

Её задача — "штамповать" новые экземпляры объектов с одинаковой структурой.

Используется, когда нужно:

- создавать несколько однотипных объектов;
- инкапсулировать логику инициализации;
- скрывать детали реализации.

```
// фабричная функция
function createUser(name, age) {
  return {
    name: name,
    age: age,
    sayHello() {
      console.log(`Привет! Меня зовут ${this.name}, мне ${this.age} лет.`);
    }
  };
}
```

```
// создаём объекты через фабричную функцию
// фабрика избавляет нас от копипаста при создании новых пользователей.
const user1 = createUser("Анна", 25);
const user2 = createUser("Иван", 30);

user1.sayHello(); // Привет! Меня зовут Анна, мне 25 лет.
user2.sayHello(); // Привет! Меня зовут Иван, мне 30 лет.
```

Плюсы фабричных функций:

- Код более читаемый, чем ручное создание объектов.
- Можно использовать замыкания для скрытия данных (псевдоинкапсуляция).
- Поддерживаются во всех версиях JS.

Минус:

- Дублирование методов. Каждый вызов `createUser` создаёт новые копии функций `sayHello`, хотя их код одинаков. Это неэффективно с точки зрения памяти.

```
const userA = createUser("Алиса", 20);  
const userB = createUser("Борис", 22);  
  
console.log(userA.sayHello === userB.sayHello);  
// false -> это разные функции в памяти
```


Оптимизация: вынесение методов наружу.

Чтобы не дублировать код, методы можно вынести за пределы фабрики:

```
function sayHello() {  
  console.log(`Привет! Меня зовут ${this.name}, мне ${this.age} лет.`);  
}  
  
function createUser(name, age) {  
  return {  
    name,  
    age,  
    sayHello // ссылка на одну и ту же функцию  
  };  
}
```

```
const user1 = createUser("Анна", 25);  
const user2 = createUser("Иван", 30);  
  
console.log(user1.sayHello === user2.sayHello);  
// true -> обе ссылки на одну и ту же функцию
```

Вывод:

- Фабричные функции — первый шаг к организации кода в стиле ООП.
- Они удобны, но создают проблему дублирования методов.
- Чтобы её решить, в JavaScript появились функции-конструкторы и прототипы.

2. Функция-конструктор: понятие, правила объявления.

Функция-конструктор — это специальный способ создавать объекты в JavaScript.

По сути это обычная функция, но есть соглашения и особенности её использования:

- имя пишут с заглавной буквы (например, User, Car);
- вызывается только через оператор **new**;
- автоматически создаёт и возвращает объект.

Как работает new?

Когда мы пишем `new User("Анна")`, происходит следующее:

- 1. Создаётся новый пустой объект: `{}`.
- 2. Этот объект связывается с **this** внутри функции.
- 3. Функция-конструктор выполняется, и в объект записываются свойства.
- 4. Если конструктор явно не вернул объект, возвращается созданный на шаге 1.

```
// Функция-конструктор
function User(name, age) {
  this.name = name;
  this.age = age;

  this.sayHello = function() {
    console.log(`Привет! Меня зовут ${this.name}, мне ${this.age} лет.`);
  };
}
```

```
// Создание экземпляров
const user1 = new User("Анна", 25);
const user2 = new User("Иван", 30);
```

```
user1.sayHello(); // Привет! Меня зовут Анна, мне 25 лет.
user2.sayHello(); // Привет! Меня зовут Иван, мне 30 лет.
```

Свойства и методы, определённые в конструкторе.

- **Свойства** — переменные, которые хранятся внутри объекта.

Объявляются через **this.name = ...;**

- **Методы** — функции, привязанные к объекту.

Объявляются через **this.methodName = function() {...};**

```
function User(name, age) {  
  // свойства  
  this.name = name;  
  this.age = age;  
  
  // метод  
  this.sayHello = function() {  
    console.log(`Привет, меня зовут ${this.name}!`);  
  };  
}
```

```
const user1 = new User("Анна", 25);
```

```
const user2 = new User("Иван", 30);
```

```
console.log(user1.name); // Анна
```

```
console.log(user2.age);  // 30
```

```
user1.sayHello();        // Привет, меня зовут Анна!
```


Отличие от фабричной функции:

- Фабрика (createUser) вручную возвращает объект через **return**.
- Конструктор (User) возвращает объект автоматически при использовании **new**.

Правила объявления конструктора:

- Имя пишется с большой буквы (User, Car, Company).
- Использовать только обычные функции (не стрелочные), потому что у стрелочных нет собственного **this**.
- Внутри конструктора свойства и методы привязываются к **this**.
- Конструктор вызывается через **new**:

Минус: дублирование методов.

Как и у фабричных функций, в примере выше каждый объект получает свою копию метода `sayHello`, что неэффективно.

Решение этой проблемы — прототипы (следующий вопрос лекции).

Вывод:

- Функции-конструкторы — стандартный способ создавать объекты до появления синтаксиса `class`.
- Они удобнее фабричных функций, потому что используют встроенный механизм `new`.
- Но для оптимизации методов используется прототипное наследование.

3. Ограничения функций-конструкторов.

Функции-конструкторы позволяют удобно создавать объекты, но у них есть ряд ограничений и особенностей, которые важно учитывать.

◆ Конструктор должен вызываться с `new`.


Если вызвать функцию-конструктор без `new`, она будет вести себя как обычная функция:

- `this` не создаст новый объект,
- в строгом режиме `this` будет `undefined`,
- в нестрогом (`sloppy mode`) — `this` укажет на `window` (в браузере), что приведёт к ошибкам.

Пример:

```
function User(name) {  
  this.name = name;  
}
```

```
const user1 = new User("Анна"); //  корректно  
console.log(user1.name); // Анна
```

```
const user2 = User("Иван"); //  без new  
console.log(user2); // undefined  
console.log(window.name); // Иван (в нестрогом режиме!)
```

◆ Методы создаются заново для каждого объекта.

Если метод объявлен прямо внутри конструктора через `this.method = function() {...}`, то каждый объект получает свою копию метода.

Это неэффективно, так как функции занимают память.

 Решение — выносить методы в prototype (следующая тема).

Пример:

```
function Car(brand) {  
  this.brand = brand;  
  this.drive = function() {  
    console.log(`${this.brand} едет!`);  
  };  
}  
  
const car1 = new Car("Toyota");  
const car2 = new Car("Tesla");  
  
console.log(car1.drive === car2.drive); // false (разные функции)
```

◆ Стрелочные функции не могут быть конструкторами.

Стрелочные функции не имеют своего `this` и `prototype`.
Поэтому использовать их с `new` нельзя.

```
const Person = (name) => {  
  | this.name = name;  
};  
  
const p = new Person("Анна"); // ✗ TypeError: Person is not a constructor
```


◆ Нельзя использовать `return` для возврата примитивов.

Обычно конструкторы не возвращают значения — они создают объект сами.

Если явно вернуть объект, то он заменит `this`.

Если вернуть примитив, он будет проигнорирован.

Пример:

```
function User(name) {  
  this.name = name;  
  return 42; // примитив  
}
```

```
console.log(new User("Анна")); // User { name: "Анна" }
```

```
function Admin(name) {  
  this.name = name;  
  return { role: "admin" }; // объект  
}
```

```
console.log(new Admin("Иван")); // { role: "admin" }
```

◆ Нет инкапсуляции по умолчанию.

Все свойства, созданные через `this`, являются публичными.

Приватные свойства можно реализовать через замыкания или с помощью `#` (**новый синтаксис**).

Пример, через замыкания (старый синтаксис):

```
function Account(balance) {  
  let _balance = balance; // приватная переменная через замыкание  
  
  this.getBalance = function() {  
    return _balance;  
  };  
}  
  
const acc = new Account(100);  
console.log(acc.getBalance()); // 100  
console.log(acc._balance);      // undefined
```

Пример, (новый синтаксис):

```
class Account {  
  // Приватное поле - начинается с #  
  #balance;  
  
  constructor(balance) {  
    this.#balance = balance;  
  }  
  
  getBalance() {  
    return this.#balance;  
  }  
}  
  
const acc = new Account(5000);  
console.log(acc.getBalance()); // 5000  
// Property '#balance' is not accessible outside class 'Account'  
// because it has a private identifier.  
// console.log(acc.#balance);
```

Итоги:

Ограничения функций-конструкторов:

- Нужно обязательно использовать **new**.
- Методы дублируются для каждого объекта → лучше `prototype`.
- Стрелочные функции нельзя использовать как конструкторы.
- **return** работает особым образом: объекты заменяют **this**, примитивы игнорируются.
- Нет встроенной приватности для свойств.

4. Встроенные конструкторы.

JavaScript предоставляет набор встроенных функций-конструкторов, которые позволяют создавать объекты стандартных типов.

Большинство из них имеют два способа использования:

- литеральный синтаксис (удобный, чаще применяется);
- через оператор **new** и встроенный конструктор.


Array.

Литерал:

```
const arr1 = [1, 2, 3];
```

Через конструктор:

```
const arr2 = new Array(1, 2, 3);  
// создаст массив длиной 5 (пустые элементы)  
const arr3 = new Array(5);
```

 Использование `new Array(n)` может быть запутанным
→ предпочтительнее использовать литералы.

Date.

Конструктор для работы с датами и временем:

```
const now = new Date(); // текущая дата и время  
const specific = new Date('2023-05-15T10:30:00');  
console.log(now.getFullYear(), now.getMonth(), now.getDate());
```


Умеет преобразовываться в строку (toString(), toDateString() и др.).

Месяцы нумеруются с 0 до 11.

Function.

Можно создавать новые функции динамически:

```
const sum = new Function('a', 'b', 'return a + b');  
console.log(sum(3, 7)); // 10
```

 Такой способ редко используют (опасность XSS, хуже производительность). Обычно используют литералы function или стрелочные функции.

Object.

Базовый конструктор для всех объектов.

```
const obj1 = {};           // литерал  
const obj2 = new Object(); // через конструктор
```

Все объекты в JS наследуют от Object.prototype.

String, Number, Boolean.

Существуют объектные обёртки для примитивов:

```
const str1 = "hello";           // примитив
const str2 = new String("hi");  // объект
console.log(typeof str1);       // "string"
console.log(typeof str2);       // "object"
```

 Обычно не используют конструкторы, достаточно примитивов.

RegExp.

Конструктор регулярных выражений:

```
const re1 = /abc/i;  
const re2 = new RegExp("abc", "i");  
console.log(re1.test("ABC")); // true
```

Итоги:

- Встроенные конструкторы позволяют создавать стандартные объекты.
- Чаще применяют литералы, так как они проще и безопаснее (`[]`, `{}`, `/.../`, `""`).
- Конструкторы (`new Date()`, `new Array()`, `new Function()`) используют, когда нужен расширенный функционал..

Итоги лекции:

- Фабричные функции — обычная функция, возвращающая объект. Удобно, но дублирует методы.
- Функция-конструктор — создаёт объекты через **new**, пишется с заглавной буквы, свойства задаются через **this**. Является частью эффективного, прототипного подхода.
- Ограничения Функции-конструктор— требует **new**, дублирует методы, не работает со стрелочными функциями, может вернуть объект вручную.
- Чаще используют **литералы**, кроме **Date** и динамического **RegExp**.

Контрольные вопросы:

- Что такое фабричная функция? В чём её плюсы и минусы?
- Чем функция-конструктор отличается от фабричной функции?
- Как работает оператор new пошагово?
- Зачем в функциях-конструкторах используют this?
- Какие ограничения есть у функций-конструкторов?
- Почему методы лучше выносить в prototype, а не определять внутри конструктора?
- Какие встроенные конструкторы есть в JavaScript?
- Чем отличаются создание массива/объекта через литерал и через конструктор?
- В каких случаях использование встроенного конструктора обязательно?

Домашнее задание:

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

6 Конструктор

Учимся разным способам создавать объекты в JS и знакомимся с оператором `new`

2. Повторить материал лекции.

Материалы лекций:

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

Обратная связь:

colledge20education23@gmail.com