

# Путь к карьере Frontend Fullstack разработчика

## Модуль 1. WEB CORE

Уровень 18. Функции, массивы и объекты.  
Часть 2.



# Объекты в JavaScript

Объект в JavaScript – это неупорядоченная коллекция пар ключ-значение. Ключом может быть только строка или символ, а значением – любой тип данных, включая другие объекты.

Это некая сущность, обладающая свойствами (характеристиками) и методами (действиями). Это как ящик с инструментами: у каждого инструмента свое название (свойство) и свое назначение (метод).

Не все в JavaScript является объектом в строгом смысле этого слова.

Примитивные типы данных (`undefined`, `null`, `boolean`, `number`, `string`, `symbol`) не являются объектами. Однако, благодаря механизму автоматического преобразования типов, они могут временно вести себя как объекты в определенных контекстах.

# Создание объектов

**Литеральная нотация:** самый распространенный способ создания объектов

**Пустой объект:** `const emptyObject = {};`

Когда нужно создать объект со свойствами, то их описывают внутри фигурных скобок. Свойства указываются в формате **имяСвойства: значение**, между свойствами ставится запятая:

```
// Объект со свойствами:
const car = {
  make: 'Toyota',
  model: 'Camry',
  // Вложенные объекты:
  owner: {
    name: 'John Doe',
    age: 42
  }
};
```

## Конструктор Object

Создать объект также можно с помощью конструктора **Object**. Это объектно-ориентированный стиль программирования.

```
const car = new Object();
car.make = 'Toyota';
car.model = 'Camry';
```

Этот способ менее популярен, так как литеральная нотация более лаконична.

# Функция-конструктор

Функция-конструктор позволяет создавать несколько экземпляров объектов с одинаковыми свойствами и методами.

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
}  
const person = new Person('Bob', 25);
```

```
// Пример: создание объекта "Книга" с дополнительными методами  
function Book(title, author, year) {  
    this.title = title;  
    this.author = author;  
    this.year = year;  
    this.getSummary = function() {  
        return `${this.title}  
by ${this.author} (${this.year})`;  
    };  
}  
  
const myBook = new Book('1984', 'George Orwell', 1949);  
console.log(myBook.getSummary());
```

## Специальные свойства:

- **this**: Ссылается на текущий объект внутри метода объекта.
- Оператор **new** создает новый объект и связывает его с конструктором.

# Доступ к свойствам и методам

## Точечная нотация

Этот способ наиболее удобен, когда имя свойства известно заранее и не содержит специальных

```
const person = { name: 'Alice', age: 30 };  
console.log(person.name); // Выведет: Alice
```

## Квадратные скобки:

```
const person = { 'first name': 'Alice', age: 30 };  
console.log(person['first name']); // Выведет: Alice  
  
const propertyName = 'age';  
console.log(person[propertyName]); // Выведет: 30
```

Квадратные скобки позволяют использовать динамические имена свойств, которые хранятся в переменных, а также имена со специальными символами.

## Изменение и удаление свойств:

```
// Изменение  
person.age = 31;  
// Добавление  
person.city = 'New York';  
// Удаление  
delete person.age;
```

# Итерация по свойствам объекта

Позволяет обрабатывать данные, хранящиеся в объектах, выполнять различные вычисления и манипуляции.

Когда использовать какой метод?

- **for...in**: Для простого перебора свойств, когда порядок не важен.
- **Object.keys()**: Когда нужен массив ключей для дальнейшей обработки или сортировки.
- **Object.values()**: Когда нужны только значения свойств.
- **Object.entries()**: Когда нужно одновременно работать с ключами и значениями.

Цикл **for...in**:

Перебирает все перечисляемые свойства объекта, включая свойства из прототипа.

```
let person = {
  name: 'John',
  age: 30,
  isStudent: false
};

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

# Проверка наличия свойств

Проверка наличия свойства в объекте перед доступом к нему позволяет избежать ошибок, связанных с неопределенными свойствами. Это особенно важно при работе с динамическими данными, когда структура объектов может меняться.

## Оператор `in`:

- Проверяет, есть ли свойство в самом объекте или в его прототипе.
- Возвращает `true`, если свойство существует, иначе `false`.

## Метод `hasOwnProperty()`:

- Проверяет, есть ли свойство как собственное свойство объекта (не унаследованное).
- Возвращает `true`, если свойство определено непосредственно в объекте, иначе `false`.

```
const person = {
  name: 'Alice',
  age: 30
};

console.log('name' in person); // true (есть в объекте)
console.log(person.hasOwnProperty('name')); // true (собственное свойство)

console.log('toString' in person); // true (унаследовано от Object.prototype)
console.log(person.hasOwnProperty('toString')); // false (не собственное свойство)
```

## Когда использовать какой метод?

- `in`: Когда нужно проверить, существует ли свойство вообще, независимо от того, где оно определено (в самом объекте или в прототипе).
- `hasOwnProperty()`: Когда нужно проверить, является ли свойство собственным свойством объекта, а не унаследованным.

# Создание методов объектов

Метод объекта — это функция, которая принадлежит объекту и позволяет выполнять определенные действия с этим объектом или его данными.

**Прямо в литерале объекта.** Это самый простой и распространенный способ.

```
const person = {  
  name: 'Alice',  
  age: 30,  
  greet() {  
    console.log(`Hello, my name is ${this.name}`);  
  }  
};
```

**После создания объекта.** Позволяет добавлять методы динамически.

```
const person = { name: 'Bob' };  
person.greet = function() { ... };
```

**Использование функций-конструкторов.** Используется для создания множества объектов с одинаковыми методами.

```
function Person(name) {  
  this.name = name;  
  this.greet = function() { ... };  
}
```



# Использование методов объекта

Методы объектов позволяют нам выполнять различные действия с объектами.

## Доступ к методам через точку

```
let person = {
  name: 'John',
  age: 30,
  greet() {
    return `Hello, my name is ${this.name}`;
  }
};

console.log(person.greet());
// Выведет: Hello, my name is John
```

## Доступ к методам через квадратные скобки

```
let person = {
  name: 'John',
  age: 30,
  greet() {
    return `Hello, my name is ${this.name}`;
  }
};

let result = person['greet']();
console.log(result);
// Выведет: Hello, my name is John
```

# Использование методов объекта

## Использование `this` в методах

Ключевое слово `this` в методах объекта ссылается на сам объект, что позволяет обращаться к его свойствам и другим методам:

```
let car = {  
  brand: 'Toyota',  
  model: 'Camry',  
  getInfo() {  
    return `Brand: ${this.brand}, Model: ${this.model}`;  
  }  
};  
  
console.log(car.getInfo());
```

# Контекст `this` в callback

Когда мы передаем функцию (метод объекта) в качестве callback, ее контекст выполнения меняется. В большинстве случаев, `this` внутри callback будет ссылаться на глобальный объект (в строгом режиме - на `undefined`). Это происходит потому, что функция вызывается не в контексте объекта, а в контексте, где она была передана.

```
let person = {
  name: 'John',
  age: 30,
  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

setTimeout( person.greet, 1000 ); // Выведет: Hello, my name is undefined
```

Как сохранить правильное значение `this` в callback?

Метод `bind()` создает новую функцию, где `this` жестко привязывается к указанному объекту.

```
setTimeout( person.greet.bind(person), 1000 );
```

Стрелочные функции: наследуют контекст `this` из окружающего лексического окружения.

```
setTimeout(() => person.greet(), 1000);
```

# Использование методов из других объектов

Методы одного объекта могут быть вызваны для другого объекта с использованием метода `call()` или `apply()`:

```
let person1 = {
  name: 'John',
  age: 30,
  greet() {
    return `Hello, my name is ${this.name}`;
  }
};

let person2 = {
  name: 'Jane',
  age: 25
};

console.log(person1.greet.call(person2)); // Выведет: Hello, my name is Jane
```

# Деструктуризация массивов и объектов

В JavaScript есть две чаще всего используемые структуры данных – это **Object** и **Array**.

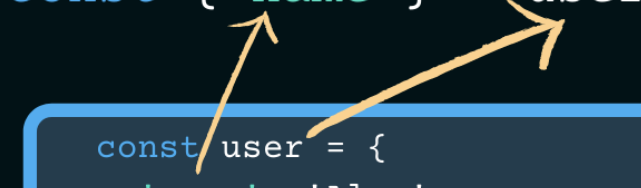
- Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам.
- Массивы позволяют нам собирать элементы данных в упорядоченный список.

Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

*Деструктурирующее присваивание* – это специальный синтаксис, который позволяет «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

```
const { name } = user;
```

```
const user = {  
  'name': 'Alex',  
  'address': '15th Park Avenue',  
  'age': 43  
}
```

A diagram with two yellow arrows. One arrow points from the 'name' property in the object literal of the second code block to the 'name' variable in the first code block. The other arrow points from the 'user' variable in the first code block to the 'user' variable in the second code block.

# Деструктуризация объектов

Деструктуризация объектов позволяет извлекать значения свойств объекта и присваивать их переменным.

```
//Синтаксис:  
const {свойство1, свойство2} = объект;
```

```
// Базовый пример деструктуризации  
const developer = {  
  firstName: 'John',  
  lastName: 'Smith',  
  skills: ['HTML', 'CSS', 'JavaScript'],  
  experience: 2  
};  
  
// Извлекаем нужные свойства  
const { firstName, lastName, skills } = developer;  
console.log(firstName); // John  
console.log(skills);     // ['HTML', 'CSS', 'JavaScript']
```

## Ключевые моменты деструктуризации:

- Извлечение нужных свойств в переменные
- Возможность переименования переменных
- Установка значений по умолчанию
- Работа с вложенными объектами

# Деструктуризация объектов

## Переименование свойств

Если мы хотим присвоить свойство объекта переменной с другим названием, то мы можем использовать двоеточие:

```
// Переименование при деструктуризации
const { firstName: name, experience: yearsOfWork } =
  developer;
console.log(name);           // John
console.log(yearsOfWork);    // 2
```

## Значения по умолчанию

Для отсутствующих свойств мы можем установить значения по умолчанию, используя "=".

```
const { salary = 3000 } = developer;
console.log(salary);          // 3000 (значение по умолчанию)
```

Значениями по умолчанию могут быть любые выражения или даже функции. Они выполнятся, если значения отсутствуют.

# Вложенная деструктуризация

Вложенная деструктуризация позволяет извлекать значения из свойств, которые сами являются объектами. Это особенно полезно при работе со сложными структурами данных.

```
let { свойствоОбъекта: { свойствоВложенногоОбъекта } } = объект;
```

```
const user = {
  name: 'Alice',
  address: {
    street: 'Main Street',
    city: 'New York',
    zip: '10001'
  }
};

// Извлекаем значения из вложенных объектов
const { name, address: { street, city } } = user;

console.log(name);    // Alice
console.log(street);  // Main Street
console.log(city);    // New York
```



# Деструктуризация в параметрах функции

Деструктуризацию можно использовать в параметрах функции:

```
// В функциях
function displayUserInfo({ firstName, lastName, skills = [] }) {
  console.log(`${firstName} ${lastName} knows ${skills.join(', ')} `);
}

// Деструктуризация в циклах
const teamMembers = [
  { name: 'Alice', role: 'Designer' },
  { name: 'Bob', role: 'Developer' }
];

for (const { name, role } of teamMembers) {
  console.log(`${name} works as ${role}`);
}
```

# Деструктуризация массивов

Это удобный способ извлечь элементы из массива и сразу присвоить их отдельным переменным.

```
/// Синтаксис:  
let [переменная1, переменная2, ...] = массив;  
  
// Пример:  
let colors = ['red', 'green', 'blue'];  
let [firstColor, secondColor] = colors;  
  
console.log(firstColor); // red  
console.log(secondColor); // green
```

## Дополнительные возможности:

- Пропуск элементов: `let [year, , day] = arr;`
- Значения по умолчанию: `let [year, month = 1] = arr;`
- Остаточный синтаксис: `let [first, ...rest] = arr;`

## Сравнение со стандартным подходом:

```
// Стандартный  
let color1 = colors[0];  
let color2 = colors[1];  
  
// Деструктуризация  
let [color1, color2] = colors;
```

# Остаток объекта

Остаток объекта позволяет собирать все оставшиеся свойства объекта в новый объект. Это полезно, когда вам нужны только некоторые свойства, а остальные вы хотите сохранить для дальнейшего использования.

Оператор **rest (...)** позволяет собирать все оставшиеся элементы в массив. Это особенно полезно при работе с функциями, где количество аргументов заранее неизвестно.

```
// Синтаксис:
let { свойство1, ...остаток } = объект;

// Пример:
let options = {
  title: 'Menu',
  width: 100,
  height: 200,
  color: 'blue'
};

let { title, ...rest } = options;

console.log(title);    // Menu
console.log(rest);
// { width: 100, height: 200, color: 'blue' }
```

## Ключевые моменты:

- Оператор **Rest** должен быть последним параметром в списке параметров функции.
- Результатом работы оператора **Rest** всегда является массив.
- Оператор **Rest** часто используется в сочетании с оператором Spread для создания гибких функций и манипуляций с данными.

# Оператор Spread

Оператор **spread (...)** позволяет «распаковывать» массивы и объекты в местах, где ожидаются нуль или более элементов. Это может быть полезно при копировании, объединении массивов или объектов, а также при передаче аргументов в функции.

## Ключевые моменты:

- Оператор **Spread** создает поверхностную копию объектов. Для глубокого копирования сложных структур данных могут потребоваться другие методы.
- Позволяет передавать произвольное количество аргументов в функцию.
- Позволяет создавать новые объекты, расширяя существующие.

## Копирование массива

В этом примере **copiedArray** является копией **originalArray**.

```
const originalArray = [1, 2, 3];  
const copiedArray = [...originalArray];  
  
console.log(copiedArray); // Выведет: [1, 2, 3]
```

# Домашнее задание

Уровень 18. Функции, массивы и объекты.  
Лекции 5-8

