ПМ3 Разработка модулей ПО.

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

Тема 1. Введение в ООП.

Лекция 2. Контекст выполнения и ключевое слово this.

Цель занятия:

Ознакомиться с понятием контекста выполнения в JavaScript, правилами работы ключевого слова this в разных ситуациях.

Учебные вопросы:

1. Понятие контекста выполнения.
2. Ключевое слово this.
3. Явное управление контекстом.

1. Понятие контекста выполнения.

Контекст выполнения (execution context) — это среда, в которой выполняется JavaScript-код.

Он определяет:

- какие переменные доступны;
- какой объект связан с ключевым словом this;
- где будет происходить выполнение кода (глобально, внутри функции, внутри метода объекта и т. д.).

Контекст выполнения — это как отдельная "комната" в памяти компьютера, где JavaScript хранит всё необходимое для работы с твоим кодом:

- Все переменные
- Все функции
- Понимание того, где он сейчас находится в коде

Виды контекста:

Глобальный контекст.

Это "верхний уровень" выполнения программы.

В браузере глобальный объект — это **window**.

В Node.js — это **global**.

В глобальном контексте **this** указывает на глобальный объект.

Локальный контекст функции.

Создаётся при вызове каждой функции.

Внутри функции у нас есть собственные переменные и значение **this**.

Значение **this** зависит от того, как вызвана функция.

```
var globalVar = "Я везде доступна";

function myFunc() {
    var localVar = "Я только здесь";

    console.log(globalVar); // Работает!
    console.log(localVar); // Работает!
}

myFunc();
console.log(globalVar); // Работает!
console.log(localVar); // ОШИБКА! Переменной здесь нет
```

2. Ключевое слово this.

Ключевое слово **this** в JavaScript — это специальная ссылка на объект, в контексте которого выполняется текущий код.

Главная особенность:

👉 значение **this** определяется во время вызова функции, а не во время её объявления.

this в глобальном контексте.

В браузере глобальный объект — window.

В Node.js — объект global.

```
console.log(this);
// в браузере -> window
// в Node.js -> {}
```

this внутри метода объекта.

Если функция вызвана как метод объекта, то **this** ссылается на этот объект.

```
const user = {
  name: "Иван",
  sayName() {
    console.log(this.name);
  }
};

user.sayName(); // "Иван"
```

this в обычной функции.

Вызов функции напрямую (не как метод объекта):

```
function showThis() {
  console.log(this);
}

showThis();
// без "use strict" -> window (в браузере)
// В Node.js: global объект
```

this в стрелочных функциях.

Стрелочные функции не имеют своего **this**.

Они берут его из внешнего (лексического) контекста

```
const user = {
  name: "Анна",
  sayName: () => {
    console.log(this.name);
  }
};

user.sayName();
// undefined, потому что стрелка взяла this из глобального контекста
```

this в стрелочных функциях.

```
const user = {
  name: "Анна",
  sayName() {
    const arrow = () => console.log(this.name);
    arrow();
  }
};

user.sayName(); // "Анна"
```

Потеря контекста.

Если метод объекта сохранить в переменную или передать как колбэк — **this** теряется.

```
const user = {
  name: "Иван",
  sayName() {
    console.log(this.name);
  }
};

const fn = user.sayName;
fn(); // undefined (или ошибка), так как this потерян
```

Таблица: Значения this в JavaScript

Где и как вызвана функция	Чему равно this (без strict)	Чему равно this (со strict)
Глобальный контекст (скрипт)	window (в браузере)	undefined
Обычная функция	window (в браузере)	undefined
Метод объекта	Сам объект	Сам объект
Вложенный вызов метода (колбэк)	window (теряется контекст)	undefined
Стрелочная функция	Берёт this у родителя	Берёт this у родителя
Конструктор (через new)	Новый объект	Новый объект
Явный вызов (call, apply, bind`)	Указанный объект	Указанный объект

📌 Итоги по this:

- В глобальной области → глобальный объект (или undefined в strict mode).
- В методе объекта → сам объект.
- В обычной функции → зависит от режима (global / undefined).
- В стрелочной функции → берётся из внешнего контекста.

3. Явное управление контекстом.

JavaScript позволяет управлять значением **this** вручную с помощью встроенных методов:

- call
- apply
- bind

Метод call

Метод **call** позволяет вызвать функцию с явно указанным значением **this** и передать аргументы по отдельности.

Ключевые применения:

- Установка контекста. Позволяет "одолжить" метод одного объекта для другого. Решает проблему потери контекста
- Заимствование методов. Использование методов массивов для массивоподобных объектов. Переиспользование методов между объектами
- Цепочка конструкторов. Вызов родительского конструктора в дочернем. Наследование в функциональном стиле.
- Работа с массивоподобными объектами arguments, NodeList, и др. Применение методов массивов к ним

Простой пример:

```
function greet() {
    console.log(`Привет, меня зовут ${this.name}`);
}

const person1 = { name: 'Алексей' };
const person2 = { name: 'Мария' };

// Вызываем функцию greet с разными значениями this
greet.call(person1); // "Привет, меня зовут Алексей"
greet.call(person2); // "Привет, меня зовут Мария"
```

Передача аргументов:

```
function introduce(age, city) {
    console.log(`Меня зовут ${this.name},
        мне ${age} лет,
        живу в городе ${city}`);
}

const user1 = { name: 'Дмитрий' };
const user2 = { name: 'Елена' };

// call принимает аргументы по отдельности
introduce.call(user1, 25, 'Астана');
introduce.call(user2, 30, 'Алматы');
```

Заимствование методов:

```
const student = {
    name: 'Иван',
    grade: 5,
    getInfo() {
        return `Студент ${this.name}, оценка: ${this.grade}`;
    }
};

const teacher = {
    name: 'Анна',
    grade: 'Отлично'
};

// Заимствуем метод getInfo у объекта student для объекта teacher
console.log(student.getInfo.call(teacher)); // "Студент Анна Петровна, оценка: Отлично"
```

Работа с массивоподобными объектами:

```
// Массивоподобный объект - это объект с числовыми индексами и свойством length
const arrayLike = {
    0: 'яблоко',
    1: 'банан',
    2: 'апельсин',
    length: 3
};

console.log('Массивоподобный объект:', arrayLike);
console.log('Это массив?', Array.isArray(arrayLike)); // false
console.log('Длина:', arrayLike.length); // 3
console.log('Первый элемент:', arrayLike[0]); // 'яблоко'

// Попробуем использовать методы массива - НЕ РАБОТАЕТ!
// Ошибка при push: arrayLike.push is not a function
try {
    arrayLike.push('груша'); // Ошибка!
} catch (error) {
    console.log('Ошибка при push:', error.message);
}
```

Используем CALL для заимствования методов :

```
// Превращаем в настоящий массив с помощью slice
const realArray = Array.prototype.slice.call(arrayLike);
console.log('Настоящий массив:', realArray);
console.log('Теперь это массив?', Array.isArray(realArray)); // true

// Используем join для объединения элементов
const joined = Array.prototype.join.call(arrayLike, ' - ');
console.log('Объединенные элементы:', joined); // 'яблоко - банан - апельсин'

// Используем forEach для перебора
console.log('\nПеребор элементов:');
Array.prototype.forEach.call(arrayLike, function(item, index) {
    console.log(`${index}: ${item}`);
});
```

Когда использовать call:

- Нужно вызвать функцию с определенным this прямо сейчас
- Аргументы известны заранее и их немного
- Заимствование методов
- Цепочка вызовов конструкторов
- Точная проверка типов через Object.prototype.toString.call()

Метод apply

Похож на **call**, но принимает аргументы в виде **массива**.

```
function greet(greeting, punctuation) {
  console.log(greeting + ", " + this.name + punctuation);
}

const user = { name: "Анна" };

greet.apply(user, ["Здравствуйте", "!!"]);
// Здравствуйте, Анна!!
```

Математические операции с массивами

```
// ===== КЛАССИЧЕСКИЙ ПРИМЕР: MATH.MAX/MIN =====
const numbers = [45, 12, 89, 3, 67, 23];

// Math.max НЕ принимает массив
console.log('Math.max(numbers):', Math.max(numbers)); // NaN

// НО мы можем использовать apply!
const maxNumber = Math.max.apply(null, numbers);
const minNumber = Math.min.apply(null, numbers);

console.log('Максимальное число:', maxNumber); // 89
console.log('Минимальное число:', minNumber);  // 3
```

Цепочка конструкторов

```
// ===== ЦЕПОЧКА КОНСТРУКТОРОВ =====
function Vehicle(type, wheels) {
    this.type = type;
    this.wheels = wheels;
    console.log(`Создан транспорт: ${type} с ${wheels} колесами`);
}

function Car(brand, model, year) {
    // Вызываем конструктор Vehicle с apply
    Vehicle.apply(this, ['автомобиль', 4]);
    this.brand = brand;
    this.model = model;
    this.year = year;
}

const car = new Car('Toyota', 'Camry', 2022);
console.log('Автомобиль:', car);
```

Современные альтернативы:

```
// Вместо Math.max.apply(null, arr)
Math.max(...arr)

// Вместо arr1.push.apply(arr1, arr2)
arr1.push(...arr2)
```

Главные применения:

1. Математические операции с массивами.

javascriptMath.max.apply(null, [1, 5, 3]) // Найти максимум в массиве

2. Объединение массивов.

javascriptArray.prototype.push.apply(arr1, arr2) // Добавить все элементы

3. Работа с arguments.

javascriptMath.max.apply(null, arguments) // Максимум среди всех аргументов

4. Цепочка конструкторов.

javascriptParentConstructor.apply(this, [arg1, arg2])

Когда использовать apply:

- ✅ Аргументы уже находятся в массиве
- ✅ Нужно передать неизвестное количество аргументов
- ✅ Работа с Math.max/min для массивов
- ✅ Объединение или расширение массивов

Метод bind

Возвращает новую функцию с закреплённым this.

В отличие от call и apply, функция не вызывается сразу, а только при дальнейшем вызове.

Метод **bind** в JavaScript делает одну простую вещь — привязывает функцию к определённому объекту, чтобы эта функция всегда знала, что такое **this**.

this — это то, к чему привязана функция в момент её вызова. Но иногда this "теряется". Например:

```
const user = {
  name: "Настя",
  sayHi() {
    console.log(`Привет, я ${this.name}`);
  }
};

const hi = user.sayHi; // Просто сохранили функцию в переменную
hi(); // Привет, я undefined (потеряли "this")
```

Почему undefined? Потому что this больше не ссылается на объект user, а на что-то другое (в данном случае — на глобальный объект или undefined в строгом режиме).

bind позволяет "привязать" функцию к нужному объекту, чтобы this всегда указывал на него.

Пример:

```
const user = {
  name: "Настя",
  sayHi() {
    console.log(`Привет, я ${this.name}`);
  }
};

const hi = user.sayHi.bind(user); // Привязываем user к функции
hi(); // Привет, я Настя
```

bind также может заранее подставить какие-то аргументы в вашу функцию.

```
function mult(a, b) {
  return a * b;
}

// null – это значение для this
// 2 – это значение, которое фиксируется для первого аргумента a
const doubleNum = mult.bind(null, 2); // Всегда умножает на 2
console.log(doubleNum(5)); // 10
console.log(doubleNum(10)); // 20
```

Еще пример. Потеря контекста в колбэках:

```
const counter = {
  count: 0,
  inc() {
    this.count++;
    console.log(this.count);
  }
};

setTimeout(counter.inc, 1000);
// Ошибка: this потерян (NaN)

setTimeout(counter.inc.bind(counter), 1000);
// 1 (this сохранён)
```

Метод bind возвращает новую функцию с закреплённым this.

В отличие от call и apply, функция не вызывается сразу, а только при дальнейшем вызове.

```
function greet(greeting, punctuation) {
  console.log(greeting + ", " + this.name + punctuation);
}

const user = { name: "Анна" };

const boundGreet = greet.bind(user);
boundGreet("Привет", "!"); // Привет, Анна!
```

Главное, что нужно знать:

- bind возвращает новую функцию с привязанным this (или аргументами).
- Эта новая функция всегда знает, к какому объекту относится this.

Ещё проще:

- Если функция теряет связь с объектом, используй bind.
- Если нужно заранее подставить аргументы, тоже используй bind.

Сравнение методов

Метод	Что делает	Когда использовать
call	Вызывает функцию с this и аргументами	Когда аргументы заранее известны
apply	То же, но аргументы в массиве	Когда аргументы уже в массиве
bind	Возвращает новую функцию с закреплённым this	Когда надо сохранить контекст для будущего вызова

Вывод:

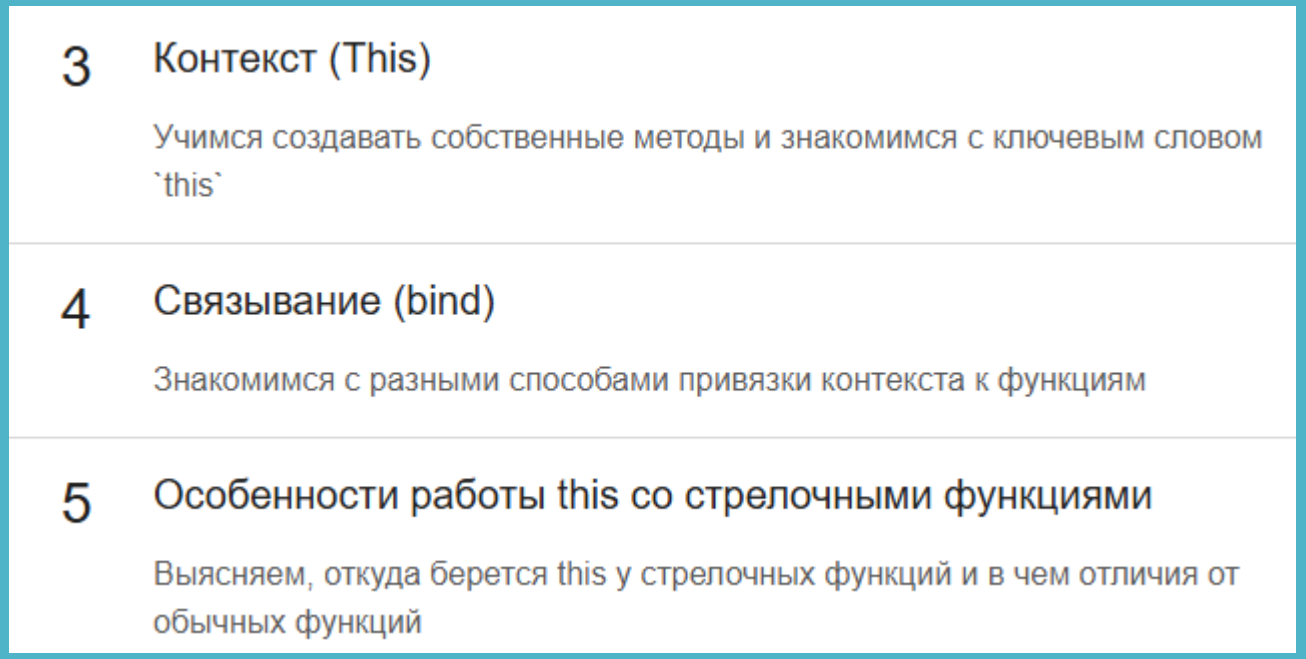
- call и apply вызывают функцию сразу.
- bind создаёт «привязанную» функцию, которую можно использовать позже.
- Эти методы позволяют точно контролировать значение this и избегать ошибок при работе с колбэками и событиями.

Контрольные вопросы:

- Что такое контекст выполнения в JavaScript?
- Чем отличается глобальный контекст от локального?
- Какое значение примет this внутри метода объекта?
- Как ведёт себя this в стрелочных функциях?
- Для чего используется метод bind?
- В чём разница между call, apply и bind?

Домашнее задание:

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



2. Повторить материал лекции.

Материалы лекций:

https://github.com/ShViktor72/Education2025

Обратная связь:

colledge20education23@gmail.com