JavaScript

Основы программирования и интерактивности

Занятие 6

Frontend Course 2025

Что такое JavaScript?

> **Вопрос:** У нас уже есть HTML (структура) и CSS (стили). Зачем еще нужен JavaScript?

JavaScript добавляет `поведение` и `интерактивность` веб-страницам.

Что делает JavaScript:

- Реагирует на действия пользователя
- Изменяет контент динамически
- Загружает данные без перезагрузки
- Создает анимации и интерактивные элементы

Краткая история JavaScript

> **Вопрос:** Почему JavaScript не имеет ничего общего с Java?

Маркетинговый ход! Java была популярна в 1995 году.

Ключевые моменты:

- **1995** - Брендан Эйх создал JavaScript за 10 дней
- **1997** - Стандартизация ECMAScript
- **2015** - ES6 принес современные возможности
- **2025** - Ежегодные обновления языка

Где работает JavaScript?

> **Вопрос:** JavaScript только для веб-сайтов?

Нет! JavaScript стал универсальным языком.

Где используется JavaScript:

- **Frontend:** интерактивность сайтов
- **Backend:** серверы (Node.js)
- **Mobile:** приложения (React Native)
- **Desktop:** программы (Electron)
- **IoT:** умные устройства
- **ML:** машинное обучение (TensorFlow.js)

Первая программа

```
// Объявление переменных
let name = "Иван";
const age = 25;

// Вывод в консоль
console.log("Привет, " + name + "!");

// Базовые операции
const sum = 10 + 5;
const message = age >= 18 ? "Да" : "Нет";

console.log(sum);
console.log(message);
```

💡 Откройте консоль разработчика (F12) и попробуйте!

Переменные: let vs const

> **Вопрос:** В чем разница между let, const и var?

let - переменную можно изменить
const - нельзя изменить после присвоения
var - устаревший способ, не использовать

```
let score = 10; // можно изменить
score = 15;

const pi = 3.14; // нельзя изменить
// pi = 3.14159; // ошибка!
```

Основные типы данных

Примитивные типы:

- `string` : текст в кавычках
- `number` : числа (целые и дробные)
- `boolean` : true/false
- `null` : пустое значение
- `undefined` : неопределенное значение
- `symbol` : уникальный идентификатор

Объектные типы:

- `object` : наборы свойств
- `array` : упорядоченные списки
- `function` : исполняемые объекты

Операторы в JavaScript

Арифметические:

```
let x = 10 + 5; // 15
let y = x * 2; // 30
let z = y / 3; // 10
```

Сравнения:

```
5 == "5"; // true (равенство с приведением)
5 === "5"; // false (строгое равенство)
5 != "5"; // false
5 !== "5"; // true
```

Логические:

```
true && false; // false (И)
true || false; // true (ИЛИ)
!true; // false (НЕ)
```

Условные операторы

> **Вопрос:** Как компьютер принимает решения?

Условные операторы! if/else выполняют разный код.

```
const age = 20;
if (age >= 18) {
  console.log("Доступ разрешен");
} else {
  console.log("Доступ запрещен");
}
```

Тернарный оператор:

```
const message = age >= 18 ? "Да" : "Нет";
```

Циклы в JavaScript

Цикл for:

```
for (let i = 0; i < 5; i++) {
  console.log("Итерация " + i);
}
```

Цикл while:

```
let count = 0;
while (count < 3) {
  console.log("Count: " + count);
  count++;
}
```

Функции - основа всего

> **Вопрос:** Зачем нужны функции?

Переиспользование и организация!

Function Declaration:

```
function greet(name) {
  return "Привет, " + name + "!";
}
```

Arrow Function (современный синтаксис):

```
const add = (a, b) => a + b;
```

Вызов функций

```
// Объявление функции
function greet(name) {
  return "Привет, " + name + "!";
}

// Вызов функции
console.log(greet("Анна")); // "Привет, Анна!"
console.log(greet("Петр")); // "Привет, Петр!"

// Стрелочная функция
const add = (a, b) => a + b;
console.log(add(5, 3)); // 8
```

Функцию нужно сначала объявить, потом вызвать!

Массивы - упорядоченные списки

> **Вопрос:** Как хранить много похожих данных?

Массивы! Упорядоченные списки элементов.

```
// Создание массива
const fruits = ["яблоко", "банан", "апельсин"];
console.log(fruits[0]); // "яблоко"
console.log(fruits.length); // 3

// Добавление элементов
fruits.push("груша"); // ["яблоко", "банан", "апельсин", "груша"]
```

Объекты - наборы свойств

> **Вопрос:** Как хранить связанные данные?

Объекты! Наборы свойств и их значений.

```
// Создание объекта
const person = {
  name: "Иван",
  age: 30,
  city: "Москва",
};

// Доступ к свойствам
console.log(person.name); // "Иван"
console.log(person["age"]); // 30
```

Массивы объектов

```
// Массив объектов - очень распространенный паттерн
const users = [
  { name: "Анна", age: 25, city: "Москва" },
  { name: "Петр", age: 30, city: "СПб" },
  { name: "Мария", age: 28, city: "Казань" },
];

// Доступ к данным
console.log(users[0].name); // "Анна"
console.log(users[1].age); // 30
```

Именно так работают базы данных и API!

Работа с DOM

> **Вопрос:** Как JavaScript изменяет HTML страницу?

DOM (Document Object Model)!

Поиск элементов:

```
const button = document.querySelector("button");
const allButtons = document.querySelectorAll("button");
const header = document.getElementById("header");
```

Изменение элементов:

```
button.textContent = "Нажми меня!";
button.style.color = "blue";
```

Обработчики событий

```
// Найти элемент
const button = document.querySelector("button");

// Добавить обработчик клика
button.addEventListener("click", () => {
  alert("Кнопка нажата!");
});

// Изменение текста при наведении
button.addEventListener("mouseover", () => {
  button.textContent = "Наведен!";
});
```

События - основа интерактивности!

Основные типы событий

Мышь:

- `click` - клик мышью
- `mouseover` - наведение курсора
- `mouseout` - уход курсора

Клавиатура:

- `keydown` - нажатие клавиши
- `keyup` - отпускание клавиши

Формы:

- `submit` - отправка формы
- `input` - изменение поля ввода

Создание элементов динамически

```
// Создать новый элемент
const newDiv = document.createElement("div");
newDiv.textContent = "Новый элемент";
newDiv.className = "box";

// Добавить на страницу
document.body.appendChild(newDiv);

// Создать несколько элементов
for (let i = 0; i < 3; i++) {
  const item = document.createElement("li");
  item.textContent = `Элемент ${i + 1}`;
  document.querySelector("ul").appendChild(item);
}
```

Асинхронность: Callback функции

> **Вопрос:** Что делать, если операция занимает время?

Асинхронный код!

```
// Callback - функция, которая вызывается позже
function fetchData(callback) {
  setTimeout(() => {
    const data = { name: "Иван", age: 30 };
    callback(data);
  }, 2000);
}

fetchData((result) => {
  console.log("Данные получены:", result);
});
```

Проблема Callback Hell

```
// Вложенные callbacks - сложно читать
loadData((data) => {
  processData(data, (result) => {
    saveResult(result, (response) => {
      showSuccess(response, () => {
        // много вложенности...
      });
    });
  });
});
```

Такой код называют "адом колбэков" (callback hell)

Promises - современный подход

```
// Создание Promise
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Данные загружены!");
  }, 2000);
});

// Использование Promise
promise
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

Promises решают проблему вложенности!

Async/Await - самый современный синтаксис

```
// Async функция всегда возвращает Promise
async function loadData() {
  try {
    const data = await fetch("/api/data");
    const result = await data.json();
    return result;
  } catch (error) {
    console.error("Ошибка:", error);
  }
}

// Использование
const result = await loadData();
```

Async/Await делает асинхронный код похожим на синхронный!

Сравнение подходов

Callback (старый подход):

```
getData((data) => {
  process(data, (result) => {
    display(result);
  });
});
```

Async/Await (современный подход):

```
async function handleData() {
  const data = await getData();
  const result = await process(data);
  display(result);
}
```

Массивы: современные методы

map - преобразование элементов:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((x) => x * 2);
// [2, 4, 6, 8, 10]
```

filter - фильтрация:

```
const users = [
  { name: "Анна", age: 17 },
  { name: "Петр", age: 25 },
];
const adults = users.filter((user) => user.age >= 18);
// [{name: "Петр", age: 25}]
```

Массивы: еще методы

reduce - агрегация значений:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((total, num) => total + num, 0);
// 15
```

find - поиск элемента:

```
const users = [{ name: "Анна" }, { name: "Петр" }];
const peter = users.find((user) => user.name === "Петр");
// {name: "Петр"}
```

Деструктуризация объектов

```
const user = {
  name: "Иван",
  age: 30,
  city: "Москва",
  email: "ivan@example.com",
};

// Старый способ
const name = user.name;
const age = user.age;

// Современный способ (деструктуризация)
const { name, age } = user;
console.log(name); // "Иван"
console.log(age); // 30
```

Деструктуризация массивов

```
const coordinates = [10, 20, 30, 40, 50];

// Старый способ
const x = coordinates[0];
const y = coordinates[1];

// Современный способ
const [x, y, z] = coordinates;
console.log(x); // 10
console.log(y); // 20
console.log(z); // 30
```

Пропуск элементов: `const [first, , third] = array;`

Spread оператор для массивов

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

// Объединение массивов
const combined = [...arr1, ...arr2];
// [1, 2, 3, 4, 5, 6]

// Добавление элемента
const newArr = [...oldArr, newItem];

// Копирование массива
const copy = [...original];
```

Spread operator (...) очень удобен!

Spread оператор для объектов

```
const person = {
  name: "Иван",
  age: 30,
};

// Создание нового объекта с дополнениями
const employee = {
  ...person,
  position: "Разработчик",
  salary: 100000,
};

// Объединение объектов
const merged = { ...obj1, ...obj2, ...obj3 };
```

Шаблонные строки

```
const name = "Иван";
const age = 30;

// Старый способ (конкатенация)
const messageOld = "Меня зовут " + name + " и мне " + age + " лет.";

// Современный способ (шаблонные строки)
const messageNew = `Меня зовут ${name} и мне ${age} лет.`;

// Многострочные строки
const html = `
<div>
  <h1>${name}</h1>
  <p>Возраст: ${age}</p>
</div>
`;
```

Стрелочные функции и this

> **Вопрос:** В чем разница между обычными и стрелочными функциями?

Контекст this! Стрелочные функции не имеют своего this.

```
const person = {
  name: "Иван",
  // Обычная функция - this = person
  greetRegular: function () {
    console.log(`Привет, я ${this.name}`);
  },
  // Стрелочная функция - this = внешний контекст
  greetArrow: () => {
    console.log(`Привет, я ${this.name}`); // undefined
  },
};
```

Работа с формами

```
const form = document.querySelector("#contact-form");

// Валидация при отправке
form.addEventListener("submit", (e) => {
  e.preventDefault(); // отменяем стандартную отправку

  const name = form.querySelector("#name").value;
  const email = form.querySelector("#email").value;

  if (!name || !email) {
    alert("Заполните все поля!");
    return;
  }

  console.log("Форма отправлена:", { name, email });
});
```

LocalStorage - сохранение данных

```
// Сохранение данных
localStorage.setItem("username", "john_doe");
localStorage.setItem("theme", "dark");

// Получение данных
const username = localStorage.getItem("username");
const theme = localStorage.getItem("theme");

// Сохранение объектов (нужно JSON.stringify)
const user = { name: "Иван", age: 30 };
localStorage.setItem("user", JSON.stringify(user));

// Получение объектов (нужно JSON.parse)
const savedUser = JSON.parse(localStorage.getItem("user"));
```

Modern JavaScript APIs

Mutation Observer

> **Вопрос:** Как отслеживать изменения DOM?

Mutation Observer! Наблюдает за изменениями элементов.

```
// Создаем Observer
const observer = new MutationObserver((mutations) => {
  mutations.forEach((mutation) => {
    console.log("Изменение:", mutation.type);
  });
});
```

Mutation Observer: настройки

> **Вопрос:** Какие изменения можно отслеживать?

Много типов изменений DOM!

```
// Наблюдаем за элементом
observer.observe(document.body, {
  childList: true, // добавление/удаление элементов
  subtree: true, // всех потомков
  attributes: true, // изменения атрибутов
  characterData: true, // изменения текста
});

if (mutation.type === "childList") {
  console.log("Добавлен/удален элемент");
}
```

Resize Observer

> **Вопрос:** Как отслеживать изменение размера элементов?

Resize Observer! Более эффективный аналог window.onresize.

```
// Создаем Observer
const resizeObserver = new ResizeObserver((entries) => {
  for (let entry of entries) {
    const { width, height } = entry.contentRect;
    console.log(`Размер: ${width}x${height}`);
  }
});
```

Resize Observer: пример использования

> **Вопрос:** Как это работает на практике?

Наблюдаем за конкретным элементом!

```
// Наблюдаем за элементом
resizeObserver.observe(document.querySelector(".container"));

// Можно наблюдать за несколькими элементами
const elements = document.querySelectorAll(".responsive");
elements.forEach((el) => resizeObserver.observe(el));

// Перестаем наблюдать
resizeObserver.unobserve(element);
```

Отладка кода

> **Вопрос:** Как найти ошибку в коде?

DevTools и console.log!

Основные методы:

```
console.log("Отладочная информация");
console.error("Ошибка!");
console.table(users); // красивая таблица
```

Откройте DevTools (F12) → Console

Вопросы?

JavaScript - мощный инструмент для создания интерактивных веб-приложений.

Не бойтесь экспериментировать и делать ошибки!

Следующее занятие: Продвинутый JavaScript и работа с API