Лекция №2: JavaScript

Web-программирование / ПГНИУ

JavaScript (JS)

- Реализация стандарта **ECMAScript (ECMA-262)**
- Исторически - язык сценариев веб-страниц
- Мульти-парадигменный язык программирования (объектно-ориентированный, императивный, функциональный)
- Интерпретируемый ЯП (или JIT-компилируемый)
- Прототипно-ориентированный ЯП
- Событийно-ориентированный ЯП
- Автоматическое управление памятью
- С-подобный синтаксис

ECMAScript

- Старый JS: ES3
- Должно основная версия (~2012): ES5
- Большие изменения в ES6 = ES2015
- С 2015 ежегодный цикл
- Текущая версия ES2020
- `ActionScript`, `QtScript`, `ExtendScript`, `Unityscript`

Типизация

- Динамическая типизация
- Слабая типизация
- Утиная типизация
- Основные стандартные типы:
  Number, *String*, *Boolean*, Array, Object, Function
  Symbol, Set, Map, WeakSet, WeakMap, *BigInt*
- Типы делятся на мутабельные и *иммутабельные*
- Все мутабельные типы - объекты

Значения

```
1, 1.0, 15n                         // Number, BigInt

'hello', "world", `Hello, ${x}`     // String

true, false                         // Boolean

[1, 2, 'c', [3, 4]]                 // Array

obj = { a: 1, b: 2 }                // Object

undefined, null, NaN, Infinity      // Special
```

Объявление переменных

```
variable = 'value'
var x, y, z = 0,
    k = '', arr = [];
{
    var a = 1; // Объявлено везде
}
{
    let b = '2'; // Объявлено только в этой области
}
{
    const c = 3;    // Объявлено только в этой области
    const d = [1, 2, 3];
    d.push(4);      // Можно
    d = ['a', 'b']; // Нельзя
}
```


Какое значение у переменной?
A: [object Object]
B: "42"
C: undefined
D: NaN

Операторы

- Арифметические: `+`, `-`, `*`, `/`, `%`, `**`
- Логические: `!`, `&&`, `||`
- Сравнения: `>`, `>=`, `<`, `<=`, `==`, `===`, `!=`, `!==`
- Бинарные: `~`, `|`, `&`, `^`
- Тернарный: `condition ? val1 : val2`
- `++`, `--`, `+=`, `-=`, `/=` …
- Optional Chaining: `mayBeNull?.property?.method?.()`
- Nullish Coasting: `mayBeNull ?? defaultValue`

Rest оператор

```
const X = {
    a: 1,
    b: 2
};

const Y = {
    ...X,
    b: 3,
};

const { a, b } = Y; // 1, 3
```

Rest оператор

```
const X = [1, 2, 3];
const Y = [...X, 4, 5];

const [a, b, ...rest] = Y;
// a = 1
// b = 2
// rest = [3, 4, 5]
```

Оператор ветвления (условия)

```
if (condition) action;

if (condition) {
    action;
    action;
} else {
    action;
    action;
}
```

Циклы

```
while (condition) action;

while (condition) { action; }

do { action; } while (condition);

for (let i = 0; i < arr.length; i++) { console.log(arr[i]); }

for (let key in arr) { console.log(arr[key]); }

for (let val of arr) { console.log(val); }
```

Функции

```
function f1(a, b, c = 1) {
    action(a, b, c);
}

const f2 = function() {
    action();
};

const f3 = f2;

(function(a, b) {
    action(a, b);
})(1, 2);
```

Функции

```
function f1() {
    function f2() {
        function f3() {
            return 1;
        }

        return f3();
    }

    return f2();
}
```

Объекты

```
const x = 1;
const obj = {
    shortName: 1,
    'long name': 2,
    ['Hello, ' + 'World']: 3,
    f1() { },
    x,
};
obj.x // 1
obj.y // undefiend
obj.shortName = 1;
obj['long name'] = 2;
obj['Hello, ' + 'World'] = 3;
obj.f1();
obj['New Value'] = 4;
```

This и контекст

- `this` - объект контекста
- Содержит контекст в котором **выполняется** код функции
- Если функция вызывается как есть, то контекст - глобальный (либо `undefined`)
- Если функция вызывается, как метод объекта, то контекст - этот объект
- Контекст можно устанавливать методами функции: `call`, `apply`, `bind`

```
function getNameGlobal() { return this.name; }

const Person = {
    name: 'Bob',
    getName1() {
      return this.name;
    },
    getName2() {
        function printName() { return this.name; }
        printName();
    },
    getName3: getNameGlobal,
};
const Person2 = { name: 'Alice' };

Person.getName1();
Person.getName2();
Person.getName3();
Person2.getName = Person.getName1();
Person2.getName();
```

```
function getNameGlobal() { return this.name; }

const Person = {
    name: 'Bob',
    getName1() {
      return this.name;
    },
    getName2() {
        function printName() { return this.name; }
        printName();
    },
    getName3: getNameGlobal,
};
const Person2 = { name: 'Alice' };

Person.getName1();                    // Bob, this === Person
Person.getName2();
Person.getName3();
Person2.getName = Person.getName1();
Person2.getName();
```

```
function getNameGlobal() { return this.name; }

const Person = {
    name: 'Bob',
    getName1() {
      return this.name;
    },
    getName2() {
        function printName() { return this.name; }
        printName();
    },
    getName3: getNameGlobal,
};
const Person2 = { name: 'Alice' };

Person.getName1();                  // Bob, this === Person
Person.getName2();                  // error, this === undefined
Person.getName3();
Person2.getName = Person.getName1();
Person2.getName();
```

```
function getNameGlobal() { return this.name; }

const Person = {
    name: 'Bob',
    getName1() {
      return this.name;
    },
    getName2() {
        function printName() { return this.name; }
        printName();
    },
    getName3: getNameGlobal,
};
const Person2 = { name: 'Alice' };

Person.getName1();                  // Bob, this === Person
Person.getName2();                  // error, this === undefined
Person.getName3();                  // Bob, this === Person
Person2.getName = Person.getName1();
Person2.getName();
```

```
function getNameGlobal() { return this.name; }

const Person = {
    name: 'Bob',
    getName1() {
      return this.name;
    },
    getName2() {
        function printName() { return this.name; }
        printName();
    },
    getName3: getNameGlobal,
};
const Person2 = { name: 'Alice' };

Person.getName1();                  // Bob, this === Person
Person.getName2();                  // error, this === undefined
Person.getName3();                  // Bob, this === Person
Person2.getName = Person.getName1();
Person2.getName();                  // Alice, this === Person2
```

```
const Person = {
    name: 'Bob'
};

function getName(prefix) {
    return prefix + this.name;
}

getName.call(Person, 'Mr. ');           // Mr. Bob
getName.apply(Person, ['Mr. ']);        // Mr. Bob
getPersonName = getName.bind(Person);
getPersonName('Mr. ');                  // Mr. Bob
```

Стрелочные функции

- Не влияет на контекст
- Может использоваться, как lambda-функция

```
// this == obj
function F(arg) {
    return this;
}
F(); // undefined
const f = (arg) => {
    return this;
};
f(); // obj
const square = x => x ** 2;
```

Методы массива

- Мутирующие массив: `push`, `pop`, `splice`, `sort`, `reverse`...
- Иммутабельные: `filter`, `map`, `some`, `every`, `reduce`...

```
const numbers = [1, 2, 3, 4]
const result = numbers.map(x => x ** 3).filter(x => x > 10).reduce((acc, x) => acc + x, 0);
// 1, 2, 3, 4 => 1, 8, 27, 64 => 27, 64 => 91
```

Геттеры и сеттеры

```
const Person = {
    firstName: 'Ivan',
    lastName: 'Ivanov',
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    },
    set fullName(s) {
        const parts = s.split();
        this.firstName = parts[0];
        this.lastName = parts[1];
    }
}
```

Ключевое слово new

- Функцию можно вызвать с `new`
  - Выполнится код функции (может модифицировать `this`)
  - Вернётся значение `this`
- Похоже на конструктор

Ключевое слово new

```
function User(name) {
    this.name = name;
    this.getName = function() {
        return this.name;
    }
    function _upperCase(s) {
        return s.toUpperCase();
    }
    this.getUpperCaseName = () => _upperCase(this.name);
}

const bob = new User('Bob');
bob.name                // Bob
bob.getName()           // Bob
bob.getUpperCaseName() // BOB
```

Прототипное наследование

- Мощный инструмент наследования
- Каждый объект имеет внутреннюю ссылку на другой объект, называемый его прототипом
- На самом верху объект с прототипом `null`
- При обращении к свойству объекта, оно будет искаться у самого объекта, затем прототипа, его прототипа и т.д.

```
const obj = { a: 1 };
// obj.__proto__ === Object.prototype -> null

const arr = [1, 2, 3];
// arr.__proto__ === Array.prototype -> Object.prototype -> null

function func() { }
// func.__proto__ === Function.prototype -> Object.prototype -> null

function User(name) {
  this.name = name;
}
User.prototype = {
  getName() {
    return this.name;
  }
}

const bob = new User('Bob');
// bob.__proto__ === User.prototype -> Object.prototype -> null
```

Классы

Синтаксический сахар над прототипами

Классы

```
class Admin extends User {
  static role = 'Admin';
  static staticMethod() { }

  name = 'Noname';
  #createdAt = null;

  constructor(name) {
    super(name);
    this.#createdAt = new Date();
  }

  #privateMethod() {
    return [this.name, this.#createdAt];
  }
};
```

Исключения

```
function test(x) {
  if (x % 2 === 0) {
    throw new Error('X must be odd');
  }
}

try {
  test(10);
} catch (e) {
  console.log(e.message); // X must be odd
} finally {
  console.log('Done');
}
```

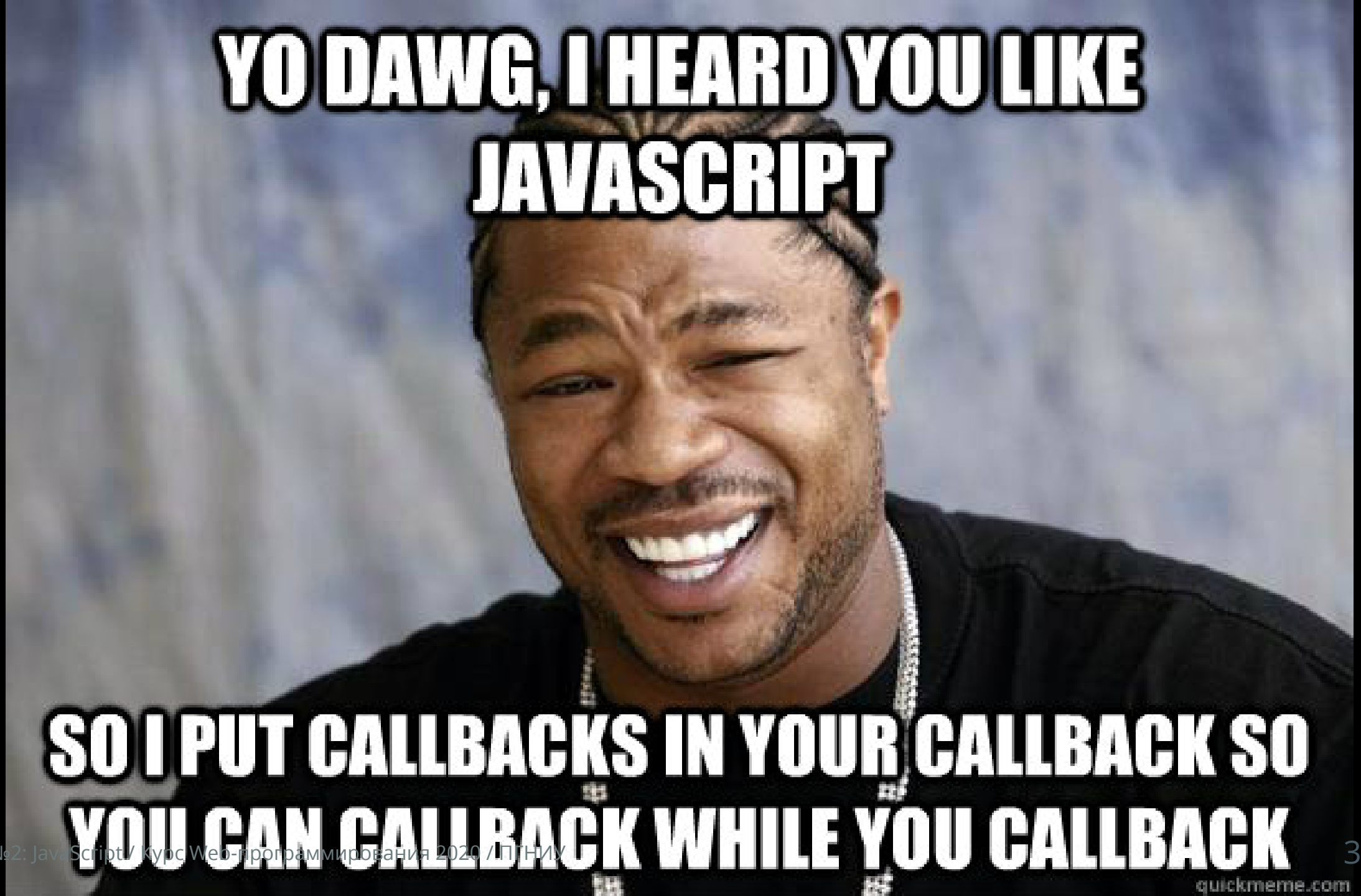
Асинхронность

- Событийно-ориентированный язык. Однопоточное, асинхронное выполнение
- I/O операции обычно асинхронные. Когда I/O операция завершается, вызывается функция - обработчик события (**callback**)

```
function workWithImage(image) { /* ... */ }

loadImage('/path', workWithImage);

const loader = new ImageLoader('/path');
loader.onload = (image) => { workWithImage(image); };
loader.onerror = (error) => { console.log(error); };
```


YO DAWG, I HEARD YOU LIKE JAVASCRIPT
SO I PUT CALLBACKS IN YOUR CALLBACK SO YOU CAN CALLBACK WHILE YOU CALLBACK
quickmeme.com

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

Промисы

- **Promise** - обещание, объект-обёртка для выполнения асинхронных операций
- Может быть в состояниях: ожидание `pending`, исполнено `fulfilled`, отклонено `rejected`
- Через метод `then` устанавливается callback успешного выполнения
- Через метод `catch` устанавливается callback отклонения
- Оба методы возвращают `Promise`, что позволяет делать цепочки
- `promise = new Promise((resolve, reject) => { // resolve(result) });`
- `promise.then(result => asyncWorkWithResult).then(work2).catch();`

Промисы

```
getUserCountry(user, (country) => {
  getCountryCurrency(country, (currency) => {
    getCurrencyCode(currency, (code) => {
      // workWithCode
    });
  });
});

getUserCountry(user)
  .then(getCountryCurrency)
  .then(getCurrencyCode)
  .then(code => {
    // workWithCode
   })
  .catch(catchAllErrors)
```

async / await

- Синтаксический сахар для промисов, который делает код "выглядящим как синхронный"
- `async` перед функцией делает её асинхронной (результат оборачивается в `Promise`)
- `await` перед вызовом асинхронной функции возвращает результат успешного выполнения промиса

async / await

```
async function getUserCode() {
  const country = await getUserCountry(user);
  const currency = await getCountryCurrency(country);
  const code = await getCurrencyCode(currency);
  return code;
}

getUserCode(user).then(code => { /* workWithCode */ });
// or in async
const code = await getUserCode(user);
```

Event Loop

- Есть классический **стек** вызова функций (**call stack**)
- Есть **очередь задач** (список задач, **queued sub-tasks**)
- Выполняется основной стек
- Когда некоторая **асинхронная** функция выполнена, в конец **очереди задач** добавляются все функции-обработчики её завершения
- Когда стек пуст - извлекается **первая** задача из **очереди задач**
- Создаётся новый контекст выполнения, заносится в стек вызовов
- Этот цикл называется **Цикл Событий** (**Event Loop**)

Модули

```
// Каждый модуль может экспортировать именованные значения и значение по умолчанию
export const x = 0;
export class User { };
export default function() {};
```

```
// И импортировать (только статически!)
import { x, User } from 'path/to/module';
import functionName from 'path/to/module';
const promise = import('path/to/module');
```

Модули кэшируются (aka `singletone`).

На самостоятельное изучение

JSON

Ссылки

- Спецификация по Javascript (ECMAScript-262): https://tc39.es/ecma262/
- TC 39: https://github.com/tc39/proposals
- MDN: https://developer.mozilla.org/ru/docs/Web/JavaScript
- Учебник по JavaScript: http://learn.javascript.ru
- Sergey Ufocoder, "Насколько JavaScript сильный?": https://medium.com/devschacht/javascript-coercions-9a36505c1370

In the next episode

Flex-box и вёрстка: подходы, best practices