

Web: страница в браузере, JavaScript

Пугачёв Константин (K.V.Pugachev@inp.nsk.su)

2018-11-28 19h

История web и мира JavaScript

(подробнее – среда, 14:30 к.508, гл.к. ИЯФ)

- 1960е: зарождение сети Интернет
- 1989-1991: зарождение WWW

Тим Бернерс-Ли разработал HTTP, HTML и URI.

- 1993: первый браузер
- 1990е: начало войн браузеров
- 1995: создание языка JavaScript (Брендан Эйх)

сделано за короткий срок, недостатки архитектуры правят и сейчас

впоследствии автор языка пострадал за гомофобию

- 2006: создание jQuery

API браузеров кривые и разные, абстрагируемся

История web и мира JavaScript

- 2009: создание Node.js (Райан Дал)
JavaScript на сервере
предшественники: SSJS (1996), родственники: HTA (1999), App.js (2012)
- 2009: CoffeeScript, 2012: TypeScript
JavaScript неудобен, давайте писать на другом языке
- 2009: ECMAScript 5 – большое обновление
- не позже 2014: Babel
старый JavaScript неудобен, давайте писать на новом
- 2015-2018: ECMAScript 2015, 2016, 2018
ECMAScript 2016 – большое обновление
jQuery совсем не нужна?

Современный Web

- засилье хипстеров и фреймворков
 - активно заимствуются абстракции из ФП
 - изобретается много новых терминов и инструментов
 - фреймворк становится неинтересным за полтора года
- медленность и нелогичность
 - слишком большой уровень косвенности
 - состыковка тяжёлых инструментов
 - *кросивэй дезойнерский* интерфейс
 - упор на визуальную “простоту” (с выхода Win8 в 2012) и урезание функциональности

Мораль: нужно изучать парадигмы и паттерны. Конкретный фреймворк изучать, когда есть либо задача, либо угроза места с хорошей зарплатой.

HTML-страница

```
<!DOCTYPE html>
<html>
  <head>
    <title>Заголовок</title>
    <script type="text/javascript"
      src="script.js"></script>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <p>
      <b>Полужижный</b> <i>Курсив</i>
      <u>Подчёркнутый</u>
    </p>
    <p>
      
    </p>
  </body>
</html>
```

объявление версии
документ
заголовок

тело страницы
параграф

второй параграф
картинка

HTML-страница

- тэги вкладываются друг в друга
- `<tag attr="value">innerHTML</tag>` – пример тэга
 - `<tag attr="value" />` – в XHTML, всё чётко как в XML
 - `<tag attr="value">` – в HTML, если стандарт допускает для tag
- `head` – метаданные, заголовок, подключение скриптов/стилей и т.п.
- `body` – отображаемая страница

Некоторые тэги

- блочные элементы (`p`, `div`, `table`) по умолчанию занимают всю ширину и не рвут текст, строчные элементы (`i`, `b`, `span`) по умолчанию могут быть в тексте
- `p` – параграф, `b` – полужирный, `i` – курсив, ...
- `div` – блок, `span` – строчный элемент
- `table` (+ `thead`, `tbody`, `caption`, `tr` – строка, `th` – ячейка шапки, `td` – ячейка) – таблица
- `img` – изображение
- `frame`, `frameset` – работа с фреймами (вставка страниц в страницу)
- `form`, `input`, `button`, `textarea` – элементы для создания форм
- `br` – перевод строки

см. <http://htmlbook.ru>

Скрипты

Скрипт из URL vs скрипт из тэга script:

```
<p>X</p>
```

1. создаётся элемент

```
<script type="text/javascript"  
  src="script.js"></script>
```

2. запускается скрипт

```
<p>Y</p>
```

3. создаётся элемент

```
<script>  
  var x = 0;  
  x += 4;  
  console.log(x);  
</script>
```

4. запускается скрипт

```
<p>Z</p>
```

5. создаётся элемент

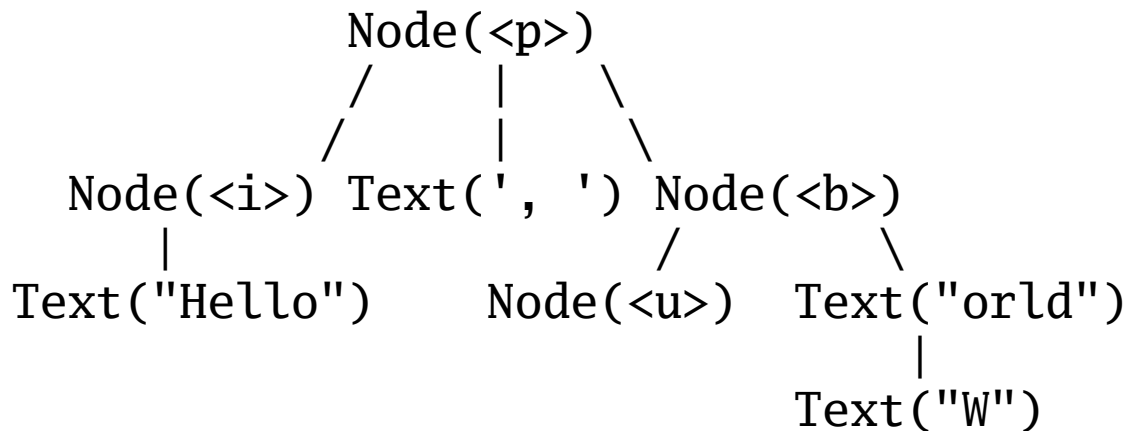
Скрипты запускаются сразу в том месте, где указаны (если без атрибута async).

DOM

- `window` – глобальный объект в браузере
- `window.document` aka `document` – дерево страницы

В `document` содержится древовидная модель документа:

`<p><i>Hello<i>, <u>W</u>orld</p>`



см. <https://developer.mozilla.org/ru/docs/Web/Reference/API>, да и всю MDN.

Работа с деревом:

- `.children` – коллекция детей-элементов

```
<div>Hello, <i>world</i></div> → [<i>world</i>]
```

- `.childNodes` – коллекция детей (+ текстовые узлы)

```
<div>Hello, <i>world</i></div> → ["Hello, ", <i>world</i>]
```

- `.parentNode` – родительский элемент
- `.appendChild`, `.insertBefore`, `.removeChild` – модификация

Внимание, `NodeList` – массивоподобные живые объекты:

```
var b = document.body, ch = b.children;
ch.forEach(console.log); // Exception
[].forEach.call(ch, console.log.bind(console)); // OK

for(var i=0; i<ch.length; ++i) b.removeChild(ch[i])//Error
while(ch.length) b.removeChild(ch[0]); // better
[].slice.call(ch).forEach(x => b.removeChild(x)); // OK
```

DOM, получение элемента

```
<p class="first">  
  <i id="hello" class="inlines">Hello<i>,  
  <b class="inlines">  
    <u class="first inlines">W</u>orld  
  </b>  
</p>
```

Работа с элементом:

- `.getAttribute`, `.setAttribute` – работа с атрибутами тэга
- `.className` – классы текстом, `.classList` – классы коллекцией
- `.id` – уникальный (программист!) идентификатор объекта
- `.name`, `.value`, `.checked`, `.enabled`, ... – для полей форм
- `.textContent`, `.innerHTML`, `.outerHTML` – работа с содержимым
- ещё некоторые поля, связанные с атрибутами

“Запросы”:

- `document.getElementById` – выбор элемента по ID
- `.getElementsByTagName`, `.getElementsByClassName`,
`.getElementsByName` – выбор набора потомков узла по критериям

DOM, селекторы

```
<p class="first">  
  <i id="hello" class="inlines">Hello<i>,  
  <b class="inlines">  
    <u class="first inlines">W</u>orld  
  </b>  
</p>
```

селектор	толкование
#a	элемент с ID a (один)
.b	элементы с классом b
c	элементы с тегом c
c.b	элементы с тегом c и классом b сразу (AND)
c > d	дети d тэга c
c d	потомки d тэга c
c + d	элементы d, перед которыми есть c
c, d	элементы c и элементы d (OR)

DOM, селекторы

<code><p id="hello" class="first"></code>	<code>.first</code>
<code> <i class="inlines">Hello<i>,</code>	<code>p i + мышь</code>
<code> <b class="inlines"></code>	
<code> <u class="inlines">W</u>orld,</code>	<code>b>.inlines:nth-child(1)</code>
<code> <u class="inlines">W</u>orld,</code>	
<code> <u class="inlines">W</u>orld</code>	
<code> <i class="first">.</i></code>	<code>.first</code>
<code> </code>	
<code></p></code>	

`.querySelector`, `.querySelectorAll` – выбор по селекторам:

```
var elements = document.querySelectorAll(
  'p i:hover, b > .inlines:nth-child(1), .first');
var b = elements[0].querySelector('b');

[].forEach.call(elements, x =>
  console.log(x.tagName, x.className));
```

Стили: из URL, в style, inline

Параметры отображения и оформления выносятся в стили. Стили задаются декларативно для селектора или конкретного элемента.

```
<link rel="stylesheet" href="style.css" />
```

```
<style>
```

```
  div {
```

```
    color: red;
```

```
    /* цвет текста */
```

```
    background-color: black;
```

```
    /* цвет фона */
```

```
    width: 200px;
```

```
    /* ширина */
```

```
  }
```

```
  div i: hover {
```

```
    text-decoration: underline; /* подчёркнутый текст */
```

```
  }
```

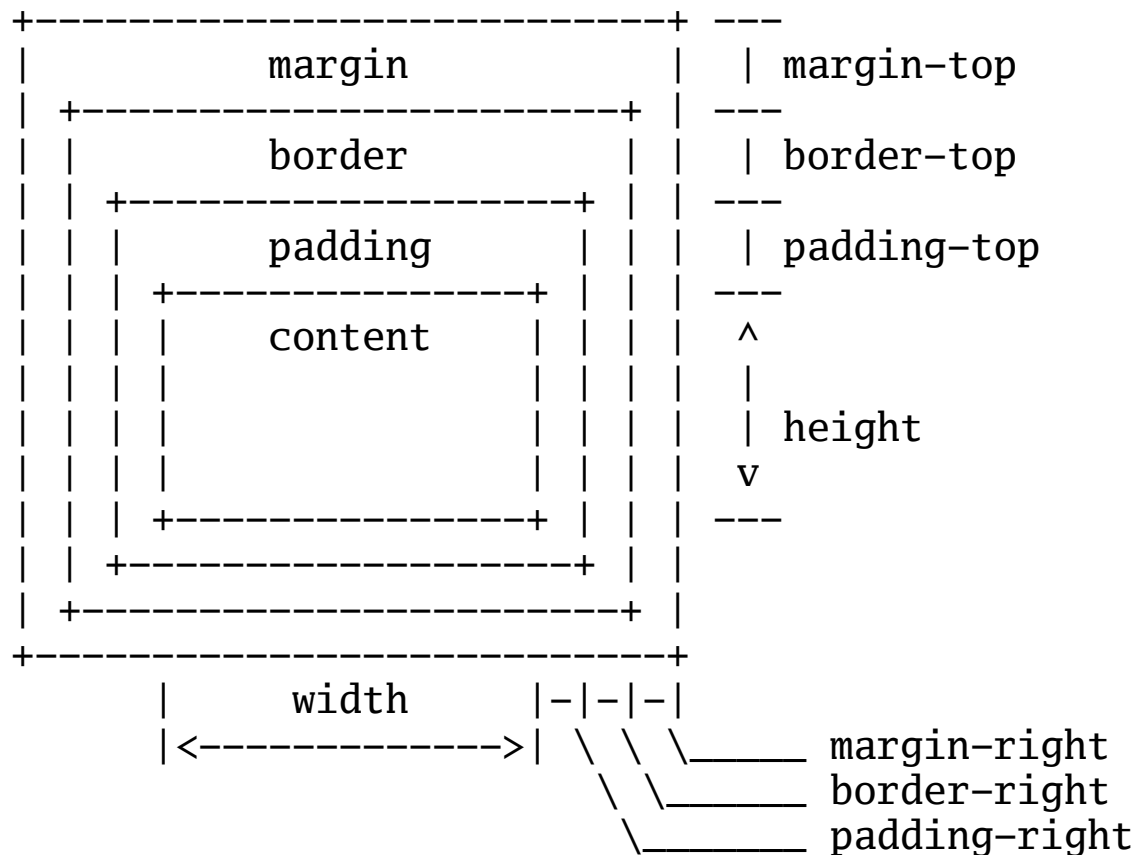
```
</style>
```

```
<div style="font-weight: bold">Bold text</div>
```

Box model

У элемента есть поля (padding), рамка (border), поля (margin).

Указанные в CSS размеры (width, height) соответствуют размеру контента.



Сверстать страницу

- Табличная вёрстка (устарело, семантически неверно)

блоки на странице – ячейки таблицы, растянутой на экран

- Блочная вёрстка (сайт строится из блоков)

блоки на странице – блоки `div`, их расположение – в CSS

- Семантическая вёрстка (каждый элемент – по сути)
отталкиваемся от логики, а не от внешнего вида

`` (акцентирование) вместо `<i>` (курсив) для акцентирования

`<div>` (блок) вместо `<table>` для формирования структуры сайта

`<table>` – для таблиц

Итог: хорошо инвалидам, владельцам необычных устройств, поисковикам

- Вёрстка на `flexbox`'ах
- Ещё варианты...

Сверстать, чтобы не распухло

- Контролировать body, html:

```
body, html { padding: 0; margin: 0;
              width: 100%; height: 100%; }
```

- Контролировать свои границы:

```
.my{ margin: 1ex; padding: 2ex; /* 6 = 1+1+2+2 */
      width:calc(100% - 6ex); height:calc(100% - 6ex); }
```

- Или включить padding и border в расчёт:

```
.my{ box-sizing: border-box; /* размеры от border */
      margin:0; padding:2ex; width:100%; height:100%; }
```

- Или вложиться в кого-то, у кого есть padding (если по горизонтали):

```
<div style="padding: 1ex">
  <div style="width: 100%">растянулся</div>
</div>
```

- Ещё варианты...

Пример: Колонки

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      /* страница - на всё окно */
      body, html { padding: 0; margin: 0; width: 100%; height: 100%; }
      /* контейнер растянется сам на 100%, columns внутри - с полями */
      .container { padding: 1em; margin: 0; background-color: #8f8; }
      /* у columns нет padding/margin - значит изнутри можно 100% */
      .column-container { display: inline-block; padding: 0; margin: 0; width: 50%; }
      /* симметричный полуотступ, чтобы как у container 1em сложилось */
      .left { margin-right: 0.5em; }
      .right { margin-left: 0.5em; }
      /* внутриколоночные поля, чтобы текст к границам не лез */
      .content { background-color: white; padding: 1em; }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="columns">
        <div class="column-container">
          <div class="left content">Контент левой колонки.</div>
        </div><!--
          внимание, если тут не закомментировать, влезет пробел!
        --><div class="column-container">
          <div class="right content">Контент правой колонки.</div>
        </div>
      </div>
    </div>
  </body>
</html>
```

Пример: Набор номера

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      /* страница - на всё окно */
      body, html { padding: 0; margin: 0; width: 100%; height: 100%; }
      /* контейнер с полями, которые вычли из размера */
      .container { padding: 1ex; margin: 0; background-color: #8f8;
        width: calc(100% - 2ex); height: calc(100% - 2ex); }
      /* лог - с полями, оставляет внизу место под панель кнопок */
      .log { background-color: white; padding: 0.5ex; margin: 0;
        width: 100% - 1ex; height: calc(100% - 2ex - 2em); overflow-y: auto; }
      /* панель кнопок высотой в 2em */
      .panel { width: 100%; height: 2em; padding: 0; margin: 1ex 0 0 0; }
      /* кнопка - по ширине панели */
      .panel button { height: 100%; }
      /* кнопка Позвонить прижимается к правому краю */
      .call { float: right; }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="log"></div>
      <div class="panel">
        <button>1</button> <button>2</button> <button>3</button>
        <button>4</button> <button>5</button> <button>6</button>
        <button>7</button> <button>8</button> <button>9</button>
        <button>0</button>
        <button class="call">Позвонить</button>
      </div>
    </div>
  </body>
</html>
```

События

При клике на элемент (click), изменении значения поля (change), активности мыши (mouseover, mouseout, ...), клавиатур, тачскрина ... возникают события, на которые можно подписаться.

```
var div = document.querySelector('div'); // 1й div
```

```
div.onclick = function(event) {  
    alert('меня кликнули!');  
}; // плохой путь: только 1 обработчик
```

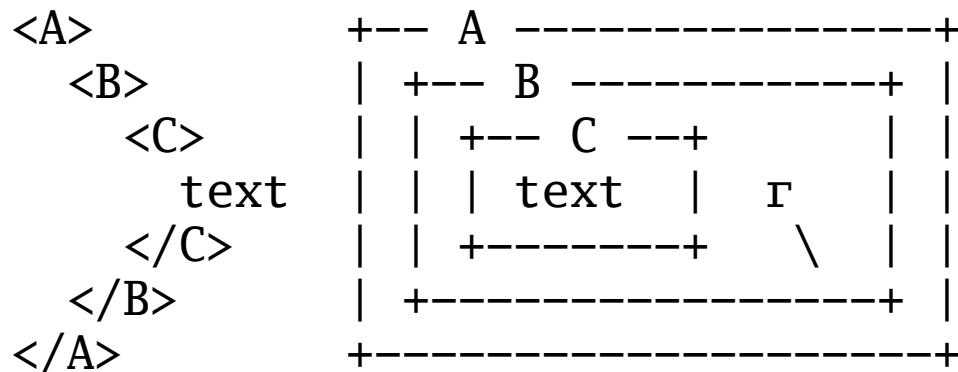
```
div.addEventListener('click', function(event){  
    alert('меня кликнули!');  
});
```

```
// Теперь при клике по элементу  
// или его детям вылезет два сообщения.
```

В event лежит информация про потревоженный элемент (target); элемент, на котором висел обработчик (currentTarget), информация о нажатии кнопки клавиатуры/мыши, координатах экрана, ... (зависит от типа события)

Bubbling & capturing

Сначала срабатывают события родителей (capturing), потом – события элемента, затем – его родителей (bubbling).



// capturing

```
A.addEventListener('click', () => alert(1), true);
B.addEventListener('click', () => alert(2), true);
C.addEventListener('click', () => alert('never'), true);
```

// bubbling

```
A.addEventListener('click', () => alert(4), false);
B.addEventListener('click', () => alert(3), false);
C.addEventListener('click', () => alert('never'), false);
```

Получим 4 сообщения в следующем порядке: 1, 2, 3, 4.

Пример: Набор номера

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      /* страница - на всё окно */
      body, html { padding: 0; margin: 0; width: 100%; height: 100%; }
      /* контейнер с полями, которые вычли из размера */
      .container { padding: 1ex; margin: 0; background-color: #8f8;
        width: calc(100% - 2ex); height: calc(100% - 2ex); }
      /* лог - с полями, оставляет внизу место под панель кнопок */
      .log { background-color: white; padding: 0.5ex; margin: 0;
        width: 100% - 1ex; height: calc(100% - 2ex - 2em); overflow-y: auto; }
      /* панель кнопок высотой в 2em */
      .panel { width: 100%; height: 2em; padding: 0; margin: 1ex 0 0 0; }
      /* кнопка - по ширине панели */
      .panel button { height: 100%; }
      /* кнопка Позвонить прижимается к правому краю */
      .call { float: right; }
    </style>
    <script>
      window.addEventListener('load', e => {
        var number = '', log = document.querySelector('.log');
        for(var btn of document.querySelectorAll('button')) {
          if(btn.classList.contains('call')) {
            btn.addEventListener('click', e => {
              log.innerHTML += 'calling <b>' + number + '</b>...<br>';
              number = '';
            });
          } else {
            btn.addEventListener('click', function(e){
              number += this.textContent;
            });
          }
        }
      });
    </script>
  </head>
  <body>
    <div class="container">
      <div class="log"></div>
      <div class="panel">
        <button>1</button> <button>2</button> <button>3</button>
        <button>4</button> <button>5</button> <button>6</button>
        <button>7</button> <button>8</button> <button>9</button>
        <button>0</button>
        <button class="call">Позвонить</button>
      </div>
    </div>
  </body>
</html>
```

Отложенное исполнение кода

```
function myFunc(hello, world) {  
    alert(hello + ', ' + world);  
}
```

// выполнится один раз через 1 секунду:

```
var tid = setTimeout(myFunc, 1000, 'hello', 'world');
```

// будет выполняться вечно каждые 0.2 секунды:

```
var iid = setInterval(alert.bind(null, 'trololo'), 200);
```

// Отмена выполнения:

```
if(Math.random() > 0.5) clearTimeout (tid);  
else                    clearInterval(iid);
```

Событийно-ориентированная архитектура, один поток

- Весь код работает в одном потоке.
- За раз отрабатывает один кусок кода (внутри `<script>`, код обработчика, код из `set*`).
- В перерыве между обработкой событий браузер обновляет страницу.
- Движок ждёт наступления событий и обрабатывает их.

Коллбеки

Функции передают функцию, которую она вызывает при достижении результата:

```
// функция нормального человека, её вызов
```

```
function square (arg) {  
  return arg * arg;  
}
```

```
var sixteen = square(square(2));  
console.log('2^4=' + sixteen);
```

```
// функция коллбечника, её вызов
```

```
function square1 (arg, cb) {  
  cb(arg * arg);  
}
```

```
square1(2, four => {  
  square1(four, sixteen => {  
    console.log('2^4=' + sixteen)  
  });  
});
```

Асинхронные вычисления с коллбеками

// синхронная функция, блокирует исполнение

```
function square (arg) {  
  var aDayAfter = Date.now() + 24 * 60 * 60 * 1000;  
  
  while(Date.now() < aDayAfter); // тормозит  
  
  return arg * arg;  
}
```

// асинхронная функция, не блокирует исполнение

```
function square1 (arg, cb) {  
  setTimeout(() => { // откладывает работу  
    cb(arg * arg);  
  }, 24 * 60 * 60 * 1000);  
}
```

// способы вызова те же, что и раньше

Пример: проверка пароля

```
// function getCsrftoken (cb)
// function checkPassword (user, pass, token, cb)

function checkMyPassword (pass, cb) {
  if(!pass) return void cb(new Error('Empty password'));

  getCsrftoken((err, token) => {
    if(err) return void cb(err); // прокидываем ошибку

    checkPassword('user123', pass, token, (err, ok) => {
      if(err) return void cb(err);
      cb(null, ok);
    });
  });
}
```

Promise (с ECMAScript 2015)

Функции передают функцию, которую она вызывает при достижении результата:

```
function square1 (arg, cb) {  
    cb(arg * arg);  
}  
  
square1(2, four => {  
    square1(four, sixteen => {  
        console.log('2^4=' + sixteen)  
    });  
});
```

Промис обещает вернуть результат и уведомить подписавшихся:

Промис вначале ожидает (pending), а затем переходит в fulfilled или rejected.

```
function square2 (arg) {  
    return new Promise(function(resolve, reject) {  
        resolve(arg * arg);  
    });  
}  
  
square2(2).then(four => {  
    // можно вернуть промис или значение  
    return square2(four);  
}).then(sixteen => {  
    console.log('2^4=' + sixteen);  
});
```

Пример: проверка пароля

```
// function getCsrftoken () -> Promise
// function checkPassword (user, pass, token) -> Promise

function checkMyPassword (pass) {
  if(!pass) return new Promise(function(resolve, reject) {
    // return void reject(new Error('Empty password'));
    throw new Error('Empty password');
  });

  return getCsrftoken().then(token) => {
    return checkPassword('user123', pass, token);
  });
}
```

В отличие от коллбеков, где нужно было ловить исключения, здесь они обрабатываются автоматически (эквивалентно вызову reject).

async/await (с ECMAScript 2017)

```
function square2 (arg) {  
  return new Promise(function(resolve, reject) {  
    resolve(arg * arg);  
  });  
}
```

// обработка через промисы

```
square2(2).then(four => {  
  return square2(four); // можно промис или значение  
}).then(sixteen => {  
  console.log('2^4=' + sixteen);  
});
```

// обработка через async/await (синт. сахар над промисами)

```
(async function() {  
  var sixteen = await square2(await square2(2));  
  console.log('2^4=' + sixteen);  
})();
```

Пример: проверка пароля

```
// function getCsrftoken () -> Promise  
// function checkPassword (user, pass, token) -> Promise  
  
async function checkMyPassword (pass) {  
    if(!pass) throw new Error('Empty password');  
    var token = await getCsrftoken();  
    var ok = await checkPassword('user123', pass, token);  
    return ok;  
}
```

В отличие от коллбеков и промисов, наблюдаем линейный код, одобренный словами `await`.

`async/await` стали доступны официально не ранее, чем в 2017-2018, т.к. вошли только в ECMAScript 2017 (добавьте время на реализацию стандарта в движках), до этого были либо коллбеки, либо прогон кода через транспайлер.

Сходства в идеях: коллбеки, async/await, монады, do-нотация

```
var sum = cb => getX(x => getY(y => cb(x + y)));
```

```
var sum2 = async () => {  
  var x = await getX();  
  var y = await getY();  
  return x + y;  
};
```

```
sum = getX >>= \x -> getY >>= \y -> return (x + y)
```

```
sum2 = do { x <- getX; y <- getX; return (x + y); }
```

В императивном коде с монадами при последовательных “разворачиваниях” значений из монады идёт всё более и более глубокое погружение в вызываемые “коллбеки”. Это похоже на асинхронные вычисления. Так же, как do-нотация “поворачивает стрелочку” в монадах, async/await “поворачивают стрелочку” в асинхронных вычислениях. И в монадах, и в промисах обработка ошибок сокрыта внутри. Промис напоминает монадическое значение.

JSON (JavaScript Object Notation) – язык представления структурированных данных

```
{  
  "key1": "value",  
  "key2": 1,  
  "void": null,  
  "children": [ 1, 2, false, 9 ],  
  "options": {"apples": 10, "spoons": 2}  
}
```

Работа с JSON:

```
var data = JSON.parse('["hello", {"world": "!"}]');  
var str  = JSON.stringify(data, null, 2); // pretty  
var str1 = JSON.stringify(data); // без пробелов/отступов
```

JSON (JavaScript Object Notation) – язык представления структурированных данных

JSON – очень узкое подмножество JavaScript, разрешает задавать сущности:

- Строки – в двойных кавычках,
- Числа, `true`, `false`, `null`,
- Массивы – в квадратных скобках, запятая как сепаратор,
- Объекты – в фигурных скобках, запятая как сепаратор, ключ – строка в двойных кавычках.

Нельзя:

- Использовать комментарии,
- Описать функции/геттеры,
- Строки в одинарных кавычках,
- Использовать запятую как терминатор.

Современный ECMAScript (2016 - 2018)

- Аргументы по умолчанию, деструктурирование

```
const {x, y, ...options} = {x: 1, y: 2, z: 3, t: 0};  
const f = ({x = 0, y = 0} = {}, z = 9) => x + y + z;
```

- Генераторы, итераторы, цикл for...of

```
function* ones() {  
  while(1) yield 1; // выдаёт бесконечность значений  
}  
  
var i = 0;  
for(var x of ones())  
  if(++i > 10) break; else console.log(x);
```

- Символы (уникальные идентификаторы, анонимизирующие свойства)
- Шаблонные строки
- Модули
- ...

Задачи

Знаний достаточно для реализации задачи 5 об асинхронных вычислениях.

Внимание, лекции, задания и сниппеты обновляются.