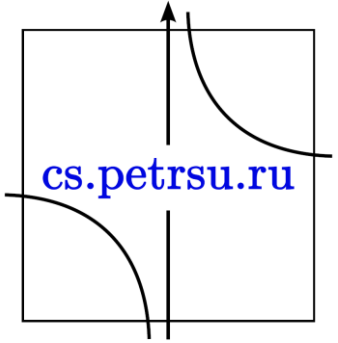
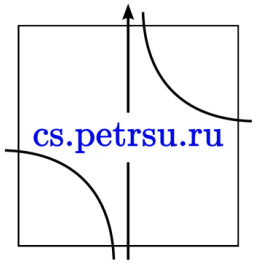


JavaScript

Тема №4

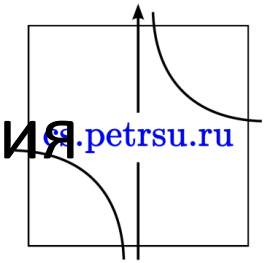
Лекция №4

- *JavaScript* был создан, чтобы «*сделать веб-страницы живыми*».
- Программы на этом языке называются *скриптами*.

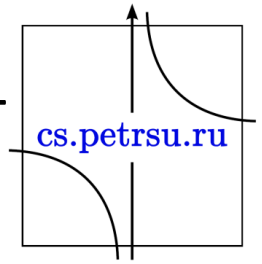


JavaScript

- Относительно простой объектно-ориентированный язык, предназначенный для создания небольших клиентских и серверных приложений для Web.
- Встраивается в HTML и выполняться автоматически при загрузке веб-страницы.
- Скрипты распространяются и выполняются, как простой текст
- Браузеры распознают встроенные в текст документа программы и выполняют их.
- Интерпретируемый язык программирования



- JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript.
- У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».
  - Движок читает («парсит») текст скрипта.
  - Затем он преобразует («компилирует») скрипт в машинный язык.
  - После этого машинный код запускается и работает достаточно быстро.



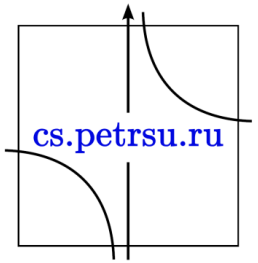
- Современный JavaScript – это «безопасный» язык программирования.
  - Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

- Возможности JavaScript сильно зависят от окружения, в котором он работает.
  - Например, Node.JS поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

- В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

В браузере JavaScript может:

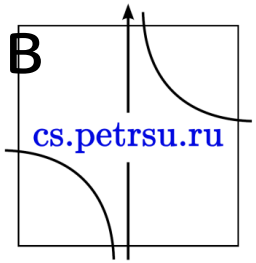
- Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
- Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
- Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (например AJAX).
- Получать и устанавливать куки, задавать вопросы посетителю, показывать сообщения.
- Запоминать данные на стороне клиента («local storage»).



Три сильные стороны JavaScript:

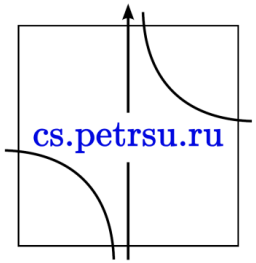
- Полная интеграция с HTML/CSS.
- Простые вещи делаются просто.
- Поддерживается всеми основными браузерами и включён по умолчанию.

→ JS стал самым распространённым инструментом для создания интерфейсов в браузере.

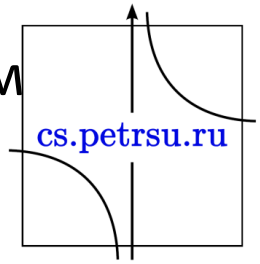


Синтаксис

- Синтаксически язык похож на *Cu* и *Java*.
- Чувствителен к регистру.
- Структурно состоит из трех частей:
  - **ядро** (*ECMAScript*),
  - **объектная модель браузера** (*Browser Object Model* или *BOM*),
  - **объектная модель документа** (*Document Object Model* или DOM).



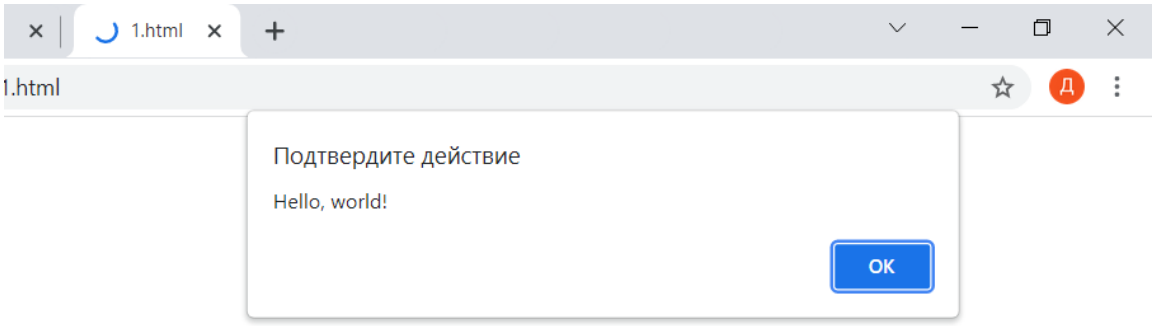
- **Ядро**
  - Ядром JavaScript является спецификация ***ECMAScript***, описывающая типы данных, инструкции, операторы, объекты, регулярные выражения и т.д.
  - Спецификация ECMA-262 содержит самую детальную и формализованную информацию о JavaScript.
- **Объектная модель браузера**
  - Каждое из окон браузера представляется объектом **window**.
  - Браузеры управляют окнами, фреймами, адресом открытой страницы, поддерживают работу с cookie.
- **Объектная модель документа**
  - Объектная модель документа - *интерфейс программирования приложений*.
  - Согласно DOM документу можно поставить в соответствие дерево объектов, обладающих рядом свойств, которые позволяют производить с ним различные манипуляции.

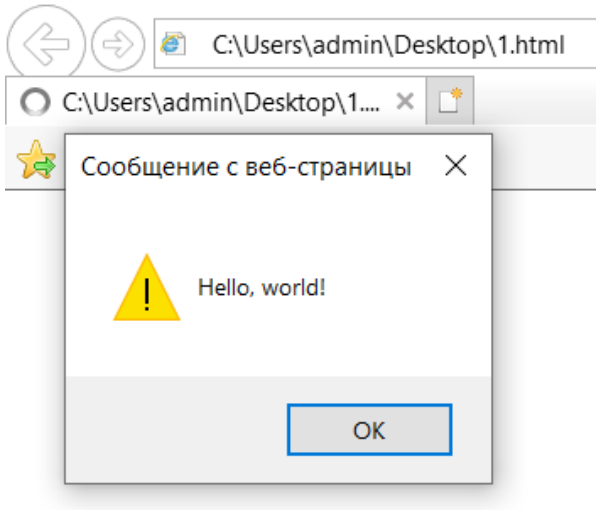


Способы размещения сценариев:

- Встраивание кода JavaScript в web-страницы:

```
<!DOCTYPE html>
<html>
 <head>
  <title></title>
  <script type="text/javascript">
   alert("Hello, world!");
  </script>
 </head>
 <body>
 </body>
</html>
```

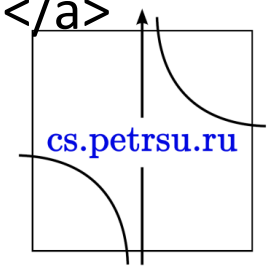




cs.petrsu.ru

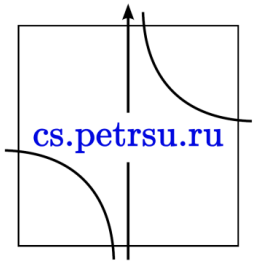
- Расположение кода JavaScript внутри HTML-тега

```
<!DOCTYPE html>
<html>
 <head>
  <title></title>
 </head>
 <body>
   <a href="page.html"
        onclick="return confirm('Подтвердите ввод?');">Готово</a>
 </body>
</html>
```



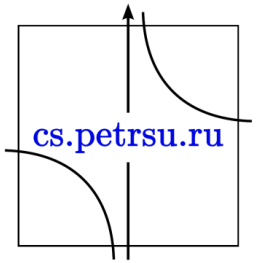
- Создание обработчика события

```
<!DOCTYPE html>
<html>
 <head>
  <title></title>
  <script type="text/vbscript">
  window.onload = function() {
   var myLink = document.getElementById("orderLink");
   myLink.onclick = function() {
      return confirm('Вы уверены?');
   };
  };
  </script>
 </head>
 <body>
  <a href="/action/" id="orderLink">Готово</a>
 </body>
</html>
```



- Вынесение JavaScript в отдельный файл

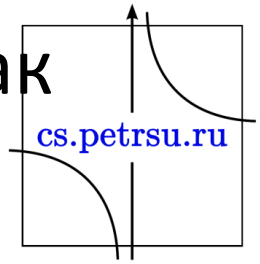
```
<script type="text/javascript"
src="//Путь/к/файлу/со/скриптом"> </script>
```



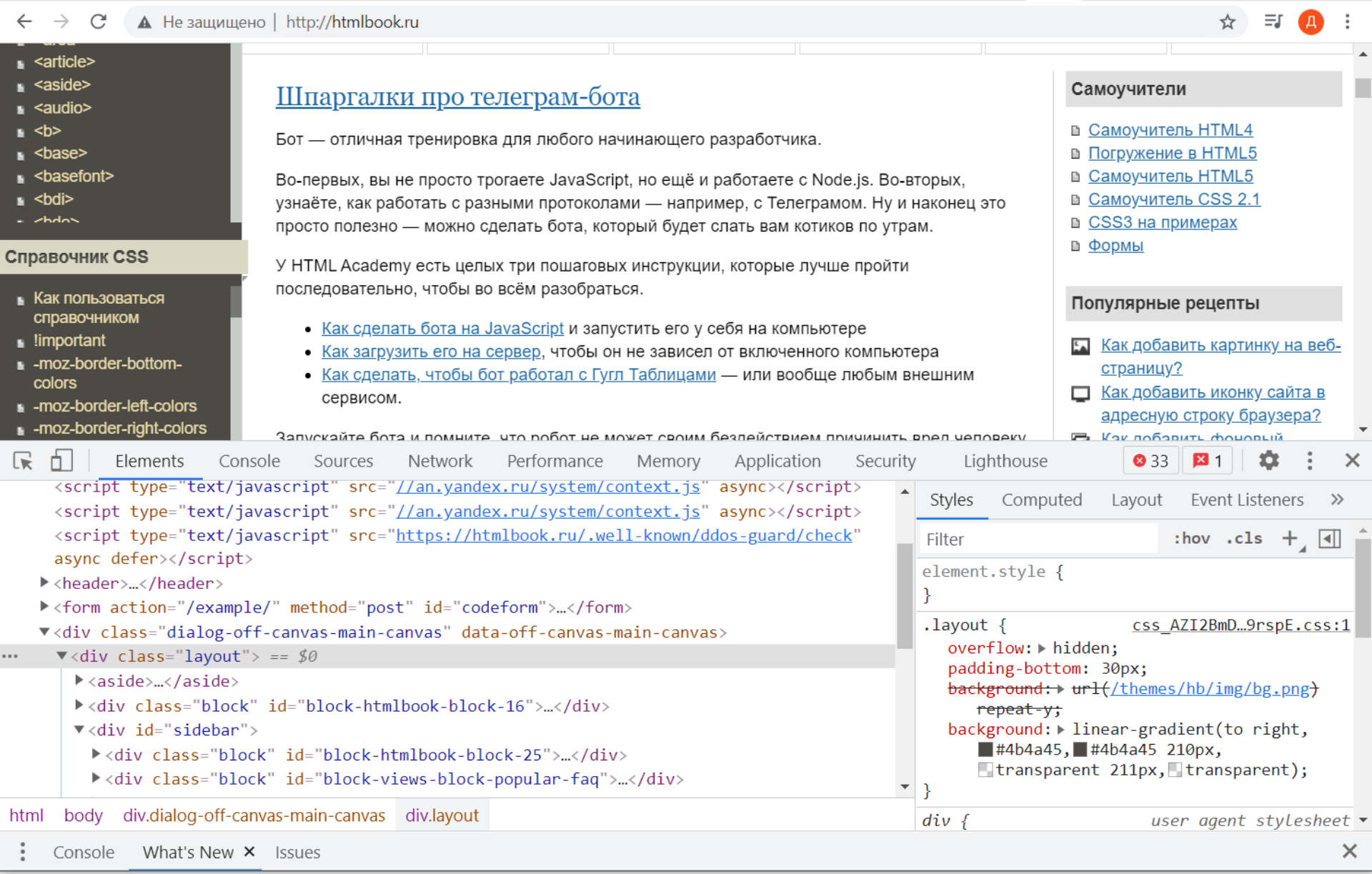
IDE и редакторы кода

- Visual Studio Code (бесплатно).
- WebStorm (платно/бесплатно).

- Atom (кроссплатформенный, бесплатный).
- Sublime Text (кроссплатформенный, условно-бесплатный).
- Notepad++ (Windows, бесплатный).
- Vim и Emacs тоже хороши, если знать, как ими пользоваться.

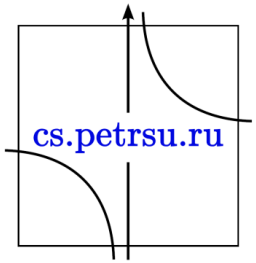


Разработка и отладка



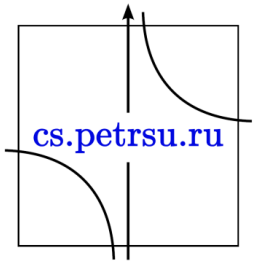
Основы JavaScript

- Комментарии:
  - // Комментарий
  - /* Многострочный комментарий */
- Директива "use strict"
  - при указании на первой строке скрипта, весь сценарий работает в «современном» режиме.



Переменные

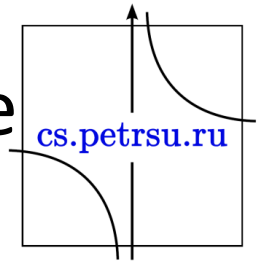
- Переменная – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения различных данных.
- Для создания переменной в JavaScript используйте ключевое слово let (var – устаревшее ключевое слово).



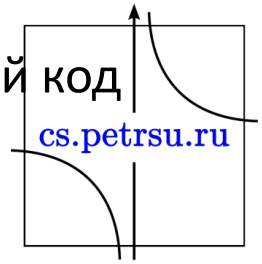
Ограничения:

- Имя переменной должно содержать только буквы, цифры или символы $ и _.
- Первый символ не должен быть цифрой.

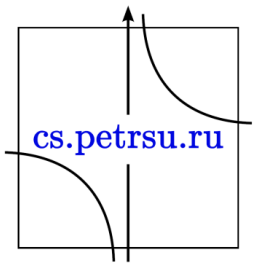
- Переменная может быть объявлена только один раз.
- Регистрозависмость (Переменные с именами apple и AppLE – это две разные переменные.)



- Если имя содержит несколько слов, обычно используется camelCase (“верблюжья” нотация):
  - слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: myVeryLongName.
- Создавать переменные с ключевым словом let
  - Без "use strict" можно было создавать переменную простым присвоением.
- Чтобы объявить константную (неизменяемую переменную) используйте const:
  - Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта.
  - Названия таких констант пишутся с использованием заглавных букв и подчёркивания.
- Название переменной должно иметь ясный и понятный смысл, говорить о том, какие данные в ней хранятся.
  - это один из самых важных и сложных навыков в программировании.
  - Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой – опытным разработчиком.



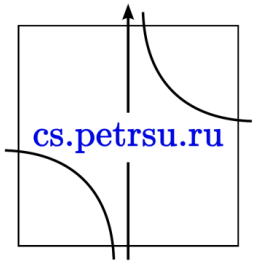
```
<sctipt>
    let message;
    message = 'Hello';

    let user = 'John', age = 25;

    alert(message); // показывает содержимое message
    alert(message+',  '+user+'!'); // показывает Hello, John!

    let $ = 1; // объявили переменную с именем "$"
    let _ = 2; // а теперь переменную с именем "_"

    console.log($ + _); // выведет в консоль число 3
</sctipt>
```



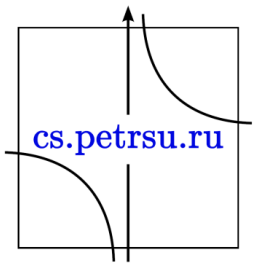
Типы данных. Число.

```
let n = 123;
n = 12.345;
```

- Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой.
- Операций для чисел:
  - умножение *, деление /, сложение +, вычитание - и т.д.
- «Специальные числовые значения»:
  - Infinity, -Infinity и NaN.

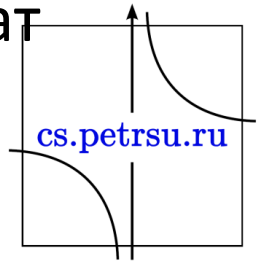


- Infinity представляет собой математическую бесконечность ∞.
  - Это особое значение, которое больше любого числа.
  - alert(1 / 0); // Infinity
  - alert(Infinity); // Infinity
- NaN означает вычислительную ошибку.
  - Это результат неправильной или неопределённой математической операции.
  - Любая операция с NaN возвращает NaN.
  - alert("не число" / 2);



- Математические операции в JavaScript «безопасны».
  - Мы можем делать что угодно: делить на ноль, обращаться с нечисловыми строками как с числами и т.д.

- Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»).
  - В худшем случае мы получим NaN как результат выполнения.

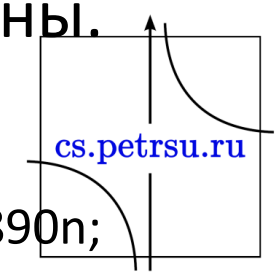


BigInt

- В JavaScript тип «number» не может содержать числа больше, чем (2^{53}-1) (т. е. 900719925474099l), или меньше, чем -(2^{53}-1) для отрицательных чисел.
  - Техническое ограничение вызвано внутренним представлением.
  - Но иногда нужны действительно гигантские числа (криптография, использовании метки времени с микросекундами).

- Тип BigInt - добавлен чтобы дать возможность работать с целыми числами произвольной длины.
  - Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

```
const bigInt = 12345678901234567890123456789012345678901234567890n;
```

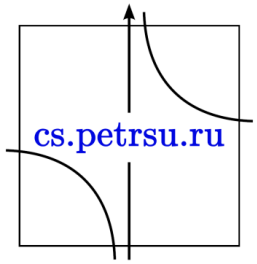


Типы данных. Строка.

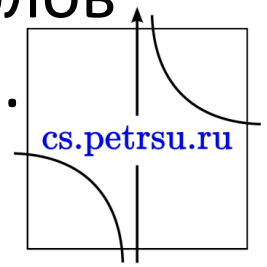
- В JavaScript существует три типа кавычек.
  - Двойные кавычки: "Привет".
  - Одинарные кавычки: 'Привет'.
  - Обратные кавычки: `Привет`.

Обратные кавычки имеют расширенную функциональность: позволяют встраивать выражения в строку, заключая их в ${...}

```
alert( `Привет, ${name}!` );
```



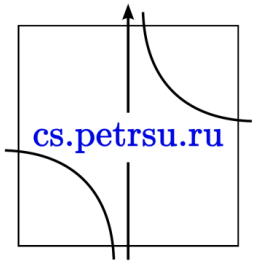
- Нет отдельного типа данных для одного символа.
  - В некоторых языках, например C и Java, для хранения одного символа, например "a" или "%", существует отдельный тип. В языках C и Java это char.
  - В JavaScript подобного типа нет, есть только тип string. Строка может содержать ноль символов (быть пустой), один символ или множество.



Булевый тип

- Булевый тип (boolean) может принимать только два значения:
  - true (истина)
  - false (ложь).

```
let nameFieldChecked = true; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
let isGreater = 4 > 1;
```

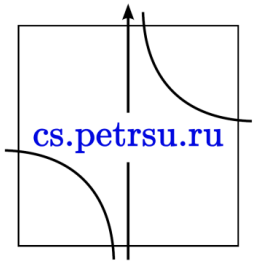


Значение «null»

- Специальное значение null не относится ни к одному из типов, описанных выше.
- Оно формирует отдельный тип, который содержит только значение null:

```
let age = null; // значение переменной age неизвестно
```

- Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

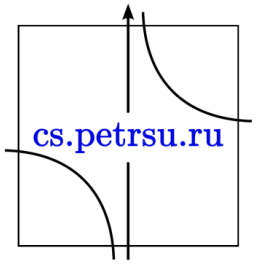


Значение «undefined»

- Специальное значение undefined также стоит особняком.
- Формирует тип из самого себя так же, как и null.
- Означает, что «значение не было присвоено».

- Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

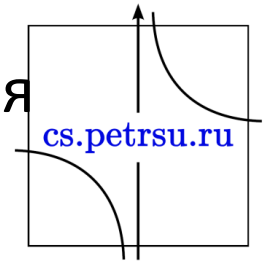
```
let age;
alert(age); // выведет "undefined"
```



- Технически можно присвоить значение undefined любой переменной:

```
let age = 123;
age = undefined; // изменяем значение на undefined
alert(age); // "undefined"
```

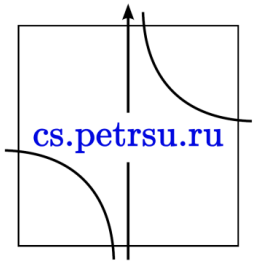
- Так делать не рекомендуется:
  - null используется для присвоения переменной «пустого» или «неизвестного» значения
  - undefined – для проверок, была ли переменная назначена.



Объекты и символы

- Тип object (объект):
  - В объектах же хранят коллекции данных или более сложные структуры.
  - Объекты занимают важное место в языке и требуют особого внимания.

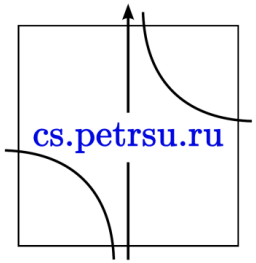
- Тип symbol (символ):
  - Используется для создания уникальных идентификаторов в объектах



Оператор typeof

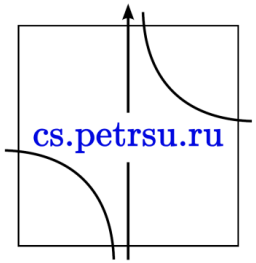
- Возвращает тип аргумента (строку с именем типа).
  - Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.

- Есть две синтаксические формы:
  - Синтаксис оператора: typeof x.
  - Синтаксис функции: typeof(x).

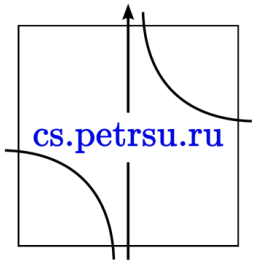


Пример:

```
typeof undefined        // "undefined"
typeof 0                // "number"
typeof 10n              // "bigint"
typeof true             // "boolean"
typeof "foo«             // "string"
typeof Symbol("id")     // "symbol"
typeof Math             // "object"  (1)
typeof null             // "object"  (2)
typeof alert            // "function"  (3)
```

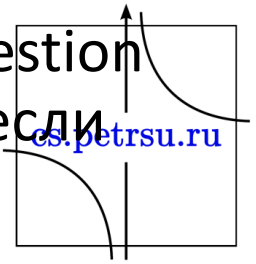


- Math — это встроенный объект, который предоставляет математические операции и константы.
- Результатом вызова typeof null является "object".
  - Это официально признанная ошибка в typeof, ведущая начало с времён создания JavaScript и сохранённая для совместимости.
- Вызов typeof alert возвращает "function", потому что alert является функцией.
  - Фактически в JavaScript нет специального типа «функция».
  - Функции относятся к объектному типу.
  - typeof обрабатывает их особым образом, возвращая "function".
  - Так повелось от создания JavaScript.



Модальные окна

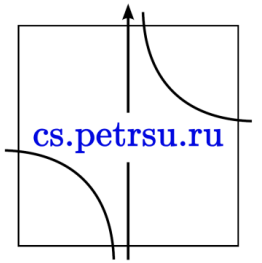
- В JavaScript взаимодействие с пользователем может быть осуществлено посредством модальных окон:
- alert(title)
  - показывает сообщение и ждёт, пока пользователь нажмёт кнопку «ОК».
- result = prompt(title, [default])
  - отобразит модальное окно с текстом, полем для ввода текста и кнопками ОК/Отмена.
- result = confirm(question)
  - отображает модальное окно с текстом вопроса question и двумя кнопками: ОК и Отмена. Результат – true, если нажата кнопка ОК. В других случаях – false.



- Методы являются модальными: то есть останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

  - Расположение окон определяется браузером. Обычно окна находятся в центре.
  - Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.



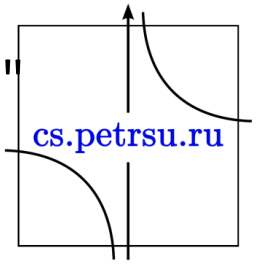
Преобразование типов

- Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.

  alert(value) преобразует значение к строке.

- Также мы можем использовать функцию String(value), чтобы преобразовать значение к строке:

```
let value = true;
alert(typeof value);       // boolean
value = String(value);     // теперь value это строка "true"
alert(typeof value);       // string
```

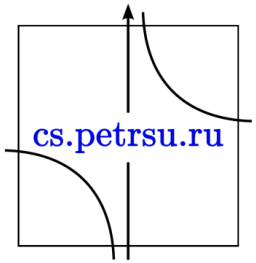


- Численное преобразование происходит в математических функциях и выражениях.

```
alert( "6" / "2" );          // 3, строки преобразуются в числа
```

- Мы можем использовать функцию Number(value), чтобы явно преобразовать value к числу:

```
let str = "123";
alert(typeof str);           // string
let num = Number(str);       // становится числом 123
alert(typeof num);           // number
```



- Явное преобразование часто применяется, когда мы ожидаем получить число из строкового контекста, например из текстовых полей форм.

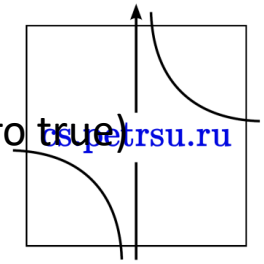
Значение	**Преобразуется в …**
undefined	NaN
null	0
true / false	1 / 0
string	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN.

Логическое преобразование

- Происходит в логических операциях.
- Может быть выполнено явно с помощью функции Boolean(value).

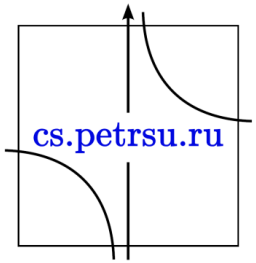
- Правило преобразования:
  - Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false.
  - Все остальные значения становятся true.

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false
alert( Boolean("Привет!") ); // true
alert( Boolean("") ); // false
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // пробел это тоже true (любая непустая строка это true)
```



Базовые операторы

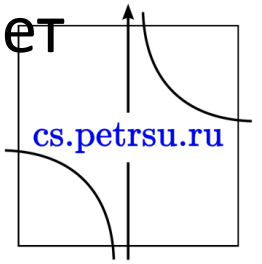
- Поддерживаются следующие математические операторы:
  - Сложение +,
    - Применяется как к числам так и строкам
    - alert(2 + 2 + '1'); // будет "41", а не "221«
    - Можно использовать как унарный оператор приведения к числу, аналогично Number(...)
  - Вычитание -,
  - Умножение *,
  - Деление /,
  - Взятие остатка от деления %,
  - Возведение в степень **.
    - Можно использовать для не целых чисел
    - alert(4 ** (1/2))



Присваивание

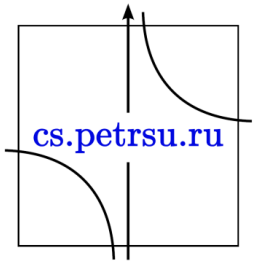
- Присваивание является оператором, а не «магической» конструкцией языка, что имеет интересные последствия.
  - Большинство операторов в JavaScript возвращают значение.
  - Для некоторых это очевидно, например сложение + или умножение *.
  - Но и оператор присваивания не является исключением.

- Вызов x = value записывает value в x и возвращает его.
- Так же возможно присваивание по цепочке.



```
let a = 1;
let b = 2;
let c = 3 - (a = b + 1);
alert( a );   // 3
alert( c );   // 0
```

```
let a, b, c;
a = b = c = 2 + 2;
alert( a );   // 4
alert( b );   // 4
alert( c );   // 4
```



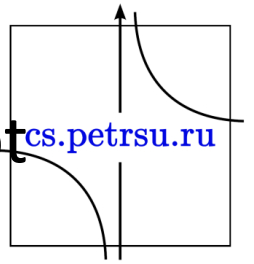
Операторы сравнения

- Больше/меньше: a > b, a < b.
- Больше/меньше или равно: a >= b, a <= b.
- Равно: a == b.
  - Обратите внимание, для сравнения используется двойной знак равенства ==.
  - Один знак равенства a = b означал бы присваивание.
- Не равно записывается как a != b.

Операторы сравнения

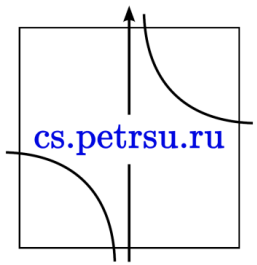
- Все операторы сравнения возвращают значение логического типа:
  - true – означает «да», «верно», «истина».
  - false – означает «нет», «неверно», «ложь».

- Для сравнения строк JavaScript использует «алфавитный» или «лексикографический» порядок.
  - Другими словами, строки сравниваются посимвольно.
  - Используется кодировка Unicode, а не настоящий алфавит.
- При сравнении значений разных типов JavaScript приводит каждое из них к числу.



Забавное следствие

```
let a = 0;
alert( Boolean(a) );          // false

let b = "0";
alert( Boolean(b) );          // true

alert(a == b);                // true!
```



Строгое сравнение

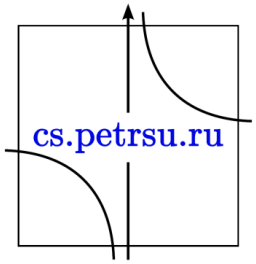
- Оператор строгого равенства === проверяет равенство без приведения типов.
  - Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

- Поведение null и undefined при сравнении:
  - При строгом равенстве ===

    Эти значения различны, так как различны их типы.
  - При нестрогом равенстве ==

    Эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.
  - При использовании математических операторов и других операторов сравнения < > <= >=

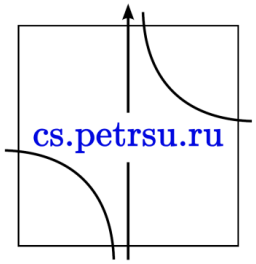
    Значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.



Условное ветвление: if, '?'

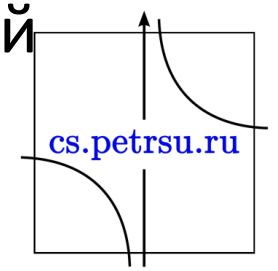
Для выполнения различных действий в зависимости от условий можно использовать инструкцию if и условный оператор ?, который также называют оператором «вопросительный знак».

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');
if (year == 2015) alert( 'Вы правы!' );
```



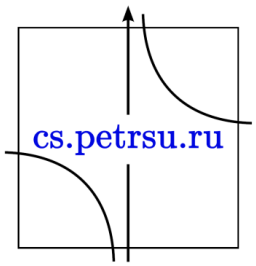
- Если необходимо выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки.
  - рекомендуется использовать фигурные скобки {} всегда, даже если выполняется только одна команда. Это улучшает читабельность кода.

- Инструкция if (…) вычисляет выражение в скобках и преобразует результат к логическому типу.
- Инструкция if может содержать необязательный блок «else» («иначе»). Он выполняется, когда условие ложно.



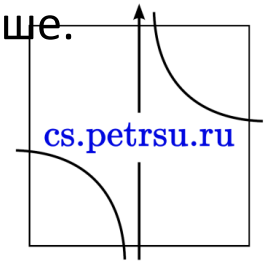
- Когда нужно проверить несколько вариантов условия можно использивать блок else if.
- «Условный» оператор «вопросительный знак» позволяет нам сделать определить что-то в зависимости от условия.
  - Он же «тернарный» - имеет три аргумента.

```
let result = условие ? значение1 : значение2;
```



Логические операторы

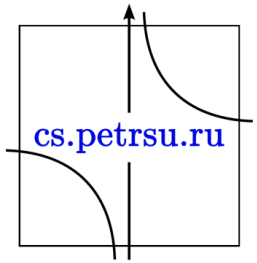
- || (ИЛИ)
  - Вычисляет операнды слева направо.
  - Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
  - Если все операнды являются ложными (false), возвращает последний из них.
- && (И)
  - Вычисляет операнды слева направо.
  - Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
  - Если все операнды были истинными, возвращается последний.
  - Приоритет && больше, чем ||, так что он выполняется раньше.
- ! (НЕ)
  - Сначала приводит аргумент к логическому типу true/false.
  - Затем возвращает противоположное значение.



Цикл “while”

```
while (condition)
{
        // код
        // также называемый "телом цикла"
}
```

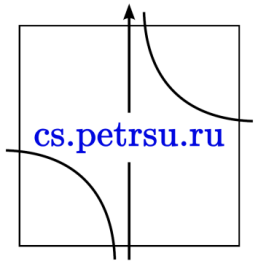
```
let i = 0;
while (i < 3)  {
        // выводит 0, затем 1, затем 2
        alert( i );
        i++;
}
```



Цикл “do...while”

```
do {
    // тело цикла
} while (condition);
```

```
let i = 0;
do {
        alert( i );
        i++;
} while (i < 3);
```



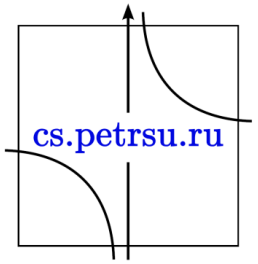
Цикл “for”

```
for (начало; условие; шаг) {
    // ... тело цикла ...
}
```

```
for (let i = 0; i < 3; i++) {
        // выведет 0, затем 1, затем 2
        alert(i);
}

for (;;) {
        // будет выполняться вечно
}
```

Прерывание цикла – break

Переход к следующей итерации - continue

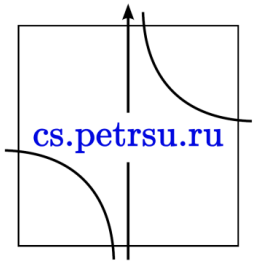


Конструкция “Switch”

- Заменяет собой сразу несколько if (наглядный способ сравнить с несколькими вариантами).

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]
  case 'value2':  // if (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
```

(!) Тип имеет значение.



Функции

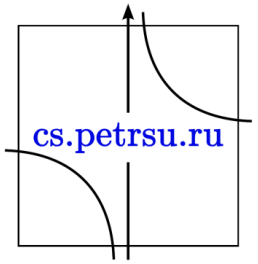
function имя(параметры)

{

/* ...тело... */

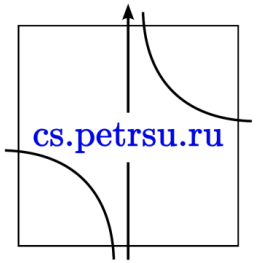
}

```
let userName = 'Вася'; /* внешняя переменная */
function showMessage(text) {
        /* локальная переменная */
        let message = 'Привет, ' + userName;
        alert(message);

        return 0;
}

showMessage();
showMessage(“Тест”);
```



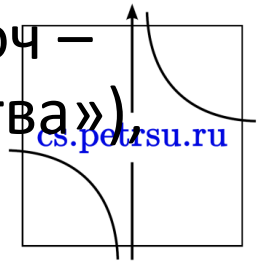
Лекция №5

JAVASCRIPT. ОБЪЕКТЫ, МАССИВЫ

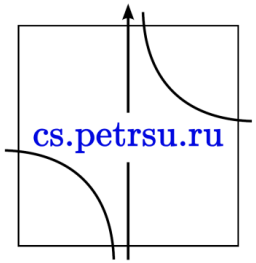


Объекты

- Объекты используются для хранения коллекций различных значений и более сложных сущностей.
- В JavaScript объекты используются очень часто, это одна из основ языка.
- Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств.
  - Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.



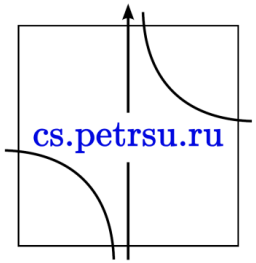
```
let user = new Object(); // синтаксис "конструктор объекта"
let user = {}; // синтаксис "литерал объекта"

let user = { // объект
        name: "John", // под ключом "name" хранится значение "John"
        age: 30, // под ключом "age" хранится значение 30
};

delete user.age;

user["likes birds"] = true;

 "key" in object  /* специальный оператор для проверки
                     существования свойства в объекте */
```



Цикл “for .. in”

```
for (key in object) {
    /* тело цикла выполняется для
       каждого свойства объекта */
}
```

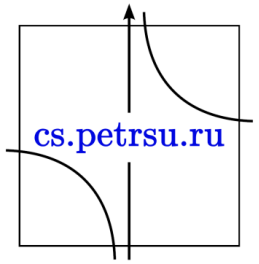
```
let user = {
        name: "John",
        age: 30,
        isAdmin: true
};
for (let key in user) {
        // ключи
        alert( key ); // name, age, isAdmin
        // значения ключей
        alert( user[key] ); // John, 30, true
}
```



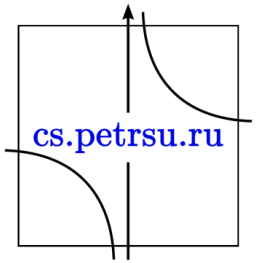
Массивы

- Объявление:

```
let arr = new Array();
let arr = [];
let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] );

fruits[2] = 'Груша';
// теперь ["Яблоко", "Апельсин", "Груша"]

fruits[3] = 'Лимон';

alert( fruits.length );
```

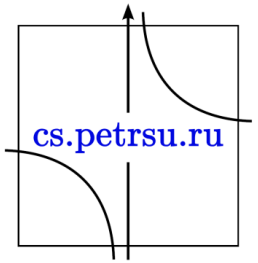


```
// квадратные скобки (обычно)
let arr = [item1, item2...];

// new Array (очень редко)
let arr = new Array(item1, item2...);


// разные типы значений
let arr = [
      'Яблоко',
      { name: 'Джон' },
      true,
      function() { alert('привет'); },
];
// получить элемент с индексом 1 (объект) и
затем показать его свойство
alert( arr[1].name ); // Джон
```

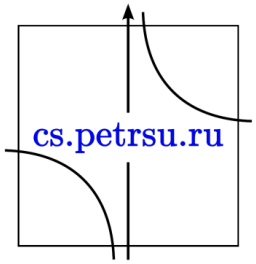


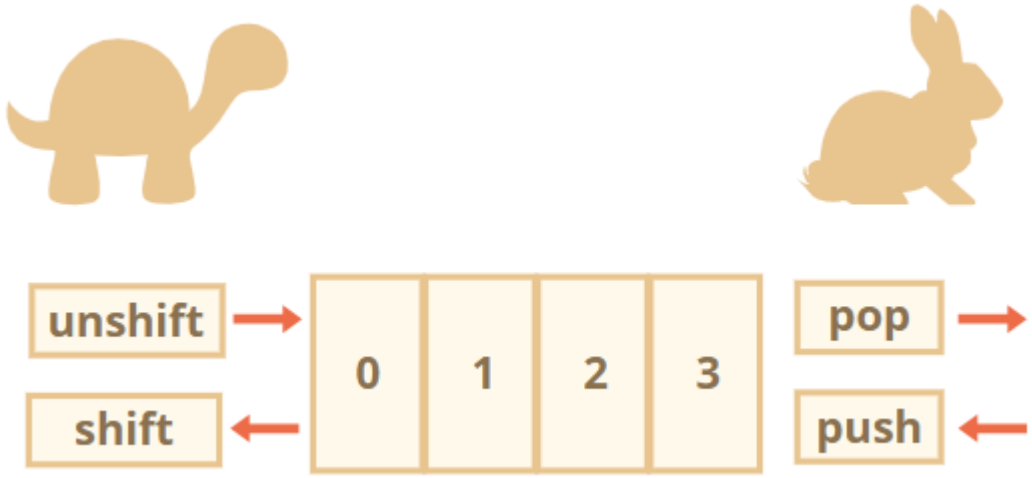
Методы работы с массивом

- push - добавляет элемент в конец
- pop - удаляет последний элемент
- unshift **-** добавляет элемент в начало массива
- shift - удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первымjoin
- reverse – меняет порядок элементов на обратный
- sort – сортировка массива
- split – разбивает строку на массив с заданным разделителем
- join – создает строку из массива с заданным разделителем
- concat – создает массив объединяя аргументы
- slice – копирование массива
- toString - возвращает список элементов, разделённых запятыми
- forEach – запуск функции дл каждого элемента
- indexOf/lastIndexOf/includes/find/findIndex – вариации поиска элемента в массиве
- isArray – проверка является ли переменная массивом
- Свойство length - общее число элементов массива



```
let fruits = ["Яблоко", "Апельсин", "Груша"];
alert( fruits.pop() ); // удаляем "Груша" и выводим его
alert( fruits ); // Яблоко, Апельсин

let fruits = ["Яблоко", "Апельсин"];
fruits.push("Груша");
alert( fruits ); // Яблоко, Апельсин, Груша


let fruits = ["Яблоко", "Апельсин", "Груша"];
alert( fruits.shift() ); // удаляем Яблоко и выводим его
alert( fruits ); // Апельсин, Груша

let fruits = ["Апельсин", "Груша"];
fruits.unshift('Яблоко');
alert( fruits ); // Яблоко, Апельсин, Груша


let fruits = ["Яблоко"];
fruits.push("Апельсин", "Груша");
fruits.unshift("Ананас", "Лимон");
// ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]
alert( fruits );
```



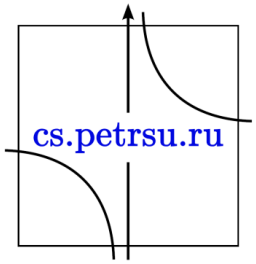


Перебор элементов

```
let arr = ["Яблоко", "Апельсин", "Груша"];
for (let i = 0; i < arr.length; i++) {
        alert( arr[i] );
}
```

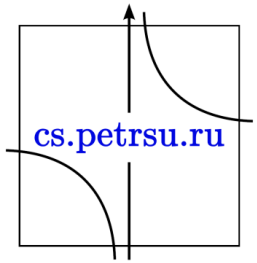
```
let fruits = ["Яблоко", "Апельсин", "Слива"];
// проходит по значениям
for (let fruit of fruits) {
        alert( fruit );
}
```

```
let arr = ["Яблоко", "Апельсин", "Груша"];
for (let key in arr) {
        alert( arr[key] ); // Яблоко, Апельсин, Груша
}
```



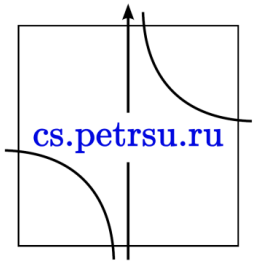
Многомерные массивы

```
let matrix = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
];
alert( matrix[1][1] ); // результат 5
```



toString

```
let arr = [1, 2, 3];
alert( arr );          // 1,2,3
alert( String(arr) === '1,2,3' ); // true

alert( [] + 1 );       // "1"
alert( [1] + 1 );      // "11"
alert( [1,2] + 1 );    // "1,21"
```



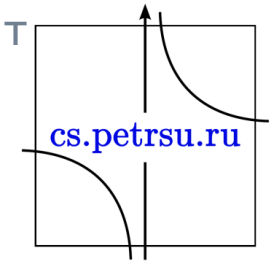
Удаление

```
let arr = ["I", "go", "home"];
delete arr[1]; // удалить "go"
alert( arr[1] ); // undefined
// теперь arr = ["I", , "home"];
alert( arr.length ); // 3
```

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

С позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

```
let arr = ["Я", "изучаю", "JavaScript"];
arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент
alert( arr ); // осталось ["Я", "JavaScript"]
```

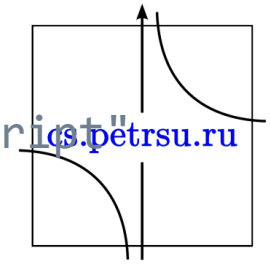


- Удаление со вставкой

```
let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];
// удалить 2 первых элемента
let removed = arr.splice(0, 2, "Использую");
alert( removed ); // "Я", "изучаю" - массив удалённых элементов
alert( arr ); // "Использую", "JavaScript", "прямо", "сейчас"
```

- Вставка без удаления

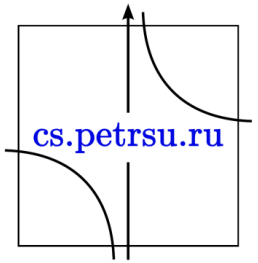
```
let arr = ["Я", "изучаю", "JavaScript"]; // с позиции 2
// удалить 0 элементов
// вставить "сложный", "язык"
arr.splice(2, 0, "сложный", "язык");
alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"
```



```
arr.slice([start], [end])
```

- возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end).
- Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива.

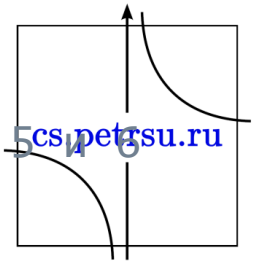
```
let arr = ["t", "e", "s", "t"];
alert( arr.slice(1, 3) ); // e,s (копирует с 1 до 3)
alert( arr.slice(-2) ); // s,t (копирует с -2 до конца)
```



```
arr.concat(arg1, arg2...)
```

- Создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.
- Принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями.
- В результате получаем новый массив, включающий в себя элементы из arr, а также arg1, arg2 и т.д.

```
let arr = [1, 2];
// создать массив из: arr и [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4
// создать массив из: arr и [3,4] и [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6
// создать массив из: arr и [3,4], потом добавить значения 5 и 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

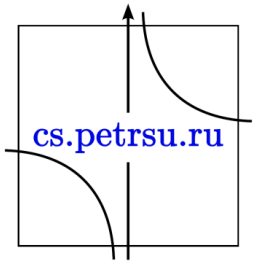


Перебор: forEach

```
arr.forEach(function(item, index, array) {
      // ... делать что-то с item
});
```

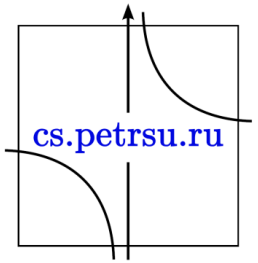
позволяет запускать функцию для каждого элемента массива.

```
["Яблоко","Апельсин","Банан"].forEach((item,index,array) => {
       alert(`${item} имеет позицию ${index} в ${array}`);
});
```



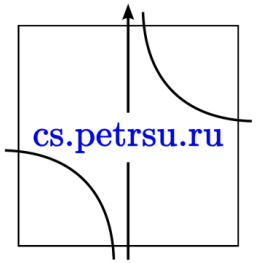
Поиск в массиве

- arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
- arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
- arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.



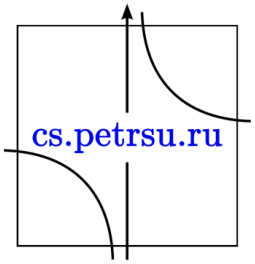
Поиск

```
let arr = [1, 0, false]; alert(
arr.indexOf(0) ); // 1 alert(
arr.indexOf(false) ); // 2 alert(
arr.indexOf(null) ); // -1 alert(
arr.includes(1) ); // true
```



```
let result = arr.find(function(item, index, array) {
        // если true - возвращается текущий элемент и
        // перебор прерывается
        // если все итерации оказались ложными,
        // возвращается undefined
});

let users = [
        {id: 1, name: "Вася"},
        {id: 2, name: "Петя"},
        {id: 3, name: "Маша"}
];
let user = users.find(item => item.id == 1);
alert(user.name); // Вася
```

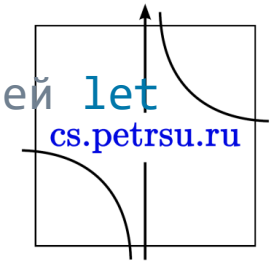
- arr.findIndex – работает аналогично, но возвращает индекс



```
let results = arr.filter(function(item, index, array) {
        // если true - элемент добавляется к результату,
        // и перебор продолжается
        // возвращается пустой массив в случае, если
        // ничего не найдено
});
```

- Синтаксис схож с find, но возвращает массив из всех подходящих элементов.

```
let users = [
        {id: 1, name: "Вася"},
        {id: 2, name: "Петя"},
        {id: 3, name: "Маша"}
];
// возвращает массив, состоящий из двух первых пользователей let
someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```



Сортировка

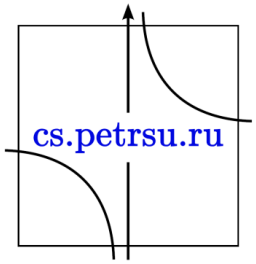
```
let arr = [ 1, 2, 15 ];
// метод сортирует содержимое arr
arr.sort();
alert( arr ); // 1, 15, 2
```

- !!! По умолчанию элементы сортируются как строки.

```
function compareNumeric(a, b) {
        if (a > b) return 1;
        if (a == b) return 0;
        if (a < b) return -1;
}
let arr = [ 1, 2, 15 ];
arr.sort(compareNumeric);
alert(arr); // 1, 2, 15
```

Или коротко:

```
arr.sort(function(a, b) { return a - b; });

arr.sort( (a, b) => a - b );
```

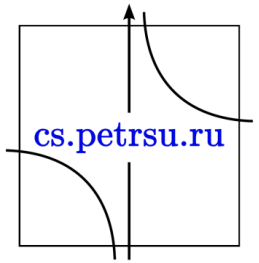


- Смена порядка элементов на обратный:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();
alert( arr ); // 5,4,3,2,1
```

- Создать строку из массива с разделителем:

```
let arr = ['Вася', 'Петя', 'Маша'];
let str = arr.join(';'); // объединить массив в строку через ;
alert( str ); // Вася;Петя;Маша
```



- Разбить строку на массив:

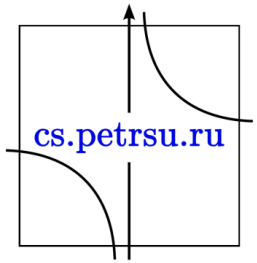
```
let names = 'Вася, Петя, Маша';
let arr = names.split(', ');
for (let name of arr) {
        alert( `Сообщение получат: ${name}.` );
        // Сообщение получат: Вася
        // Сообщение получат: Петя
        // Сообщение получат: Маша
}
```

- Необязательный параметр – ограничение на количество элементов в массиве

```
let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2);
alert(arr); // Вася, Петя
```

- Разбить строку по буквам:

```
let str = "тест";
alert( str.split('') ); // т,е,с,т
```



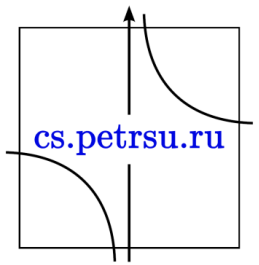
Array.isArray

- Массивы не образуют отдельный тип языка они основаны на объектах.

```
alert(typeof {}); // object
alert(typeof []); // тоже object
```

- Поэтому реализована проверка на массив:

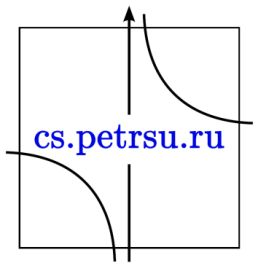
```
alert(Array.isArray({})); // false
alert(Array.isArray([])); // true
```



Объекты Date

- Для создания нового объекта Date нужно вызвать конструктор new Date().
- Без аргументов – создать объект Date с текущими датой и временем

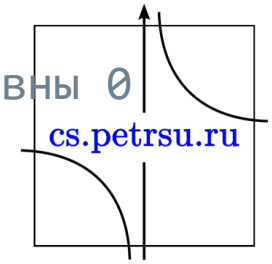
```
let now = new Date();
alert( now ); // показывает текущие дату и время
```



Другие варианты конструктора:

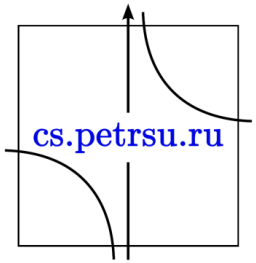
- **new Date(milliseconds)** – миллисекунды с 1 января 1970 года, называется *таймстамп* (англ. timestamp).
- **new Date(datestring)**
- **new Date(year, month, date, hours, minutes, seconds, ms)**

```
// 0 соответствует 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

let date = new Date("2017-01-26");
alert(date);

new Date(2021, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2021, 0, 1); // то же самое, т.к. часы и проч. равны 0
alert(date); // 1.01.2021, 00:00:00.000
```



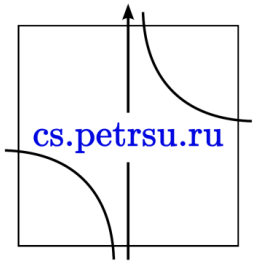
Получение компонентов даты

- getFullYear() - получить год (4 цифры)
- getMonth() - получить месяц, от 0 до 11.
- getDate() - получить день месяца, от 1 до 31, что несколько противоречит названию метода.
- getHours() – получитьи часы.
- getMinutes() – получить минуты.
- getSeconds() – получить секунды.
- getMilliseconds() - получить миллисекунды.
- getDay() – получить день недели.
- getTime() – получить таймстапм.

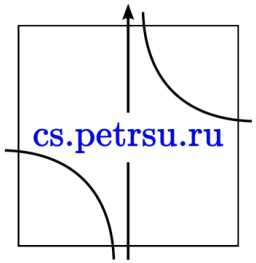


Установка компонентов даты

- setFullYear(year, [month], [date])
- setMonth(month, [date])
- setDate(date)
- setHours(hour, [min], [sec], [ms])
- setMinutes(min, [sec], [ms])
- setSeconds(sec, [ms])
- setMilliseconds(ms)
- setTime(milliseconds)



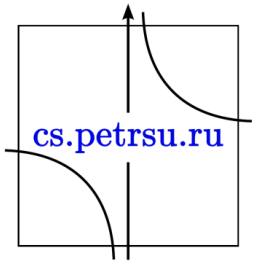
```
let today = new Date();
today.setHours(0);
alert(today); // выводится сегодняшняя дата, но час = 0
today.setHours(0, 0, 0, 0);
alert(today); // выводится сегодняшняя дата, но время 00:00:00.
```



- Существует особый метод Date.now(), возвращающий текущую метку времени.

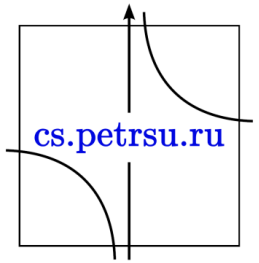
  Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик мусора.

```
let start = Date.now(); // текущий таймстапм
// выполняем некоторые действия
for (let i = 0; i < 100000; i++) {
        let doSomething = i * i * i;
}
let end = Date.now(); // заканчиваем отсчёт времени
alert( `Цикл отработал за ${end - start} миллисекунд` );
```



- Автоисправление – это очень полезная особенность объектов Date.
- Можно устанавливать компоненты даты вне обычного диапазона значений, а объект сам себя исправит.

```
let date = new Date(2021, 0, 32); // 32 января 2021 ?!?
alert(date); // ... 1 февраля 2021!
```



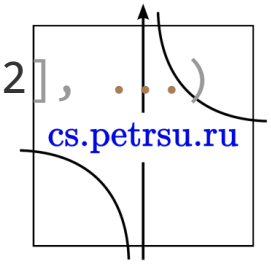
Планирование: setTimeout и setInterval

- setTimeout позволяет вызвать функцию один раз через определённый интервал времени.

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

- setInterval позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)

clearTimeout(timerId); // Отмена выполнения
```

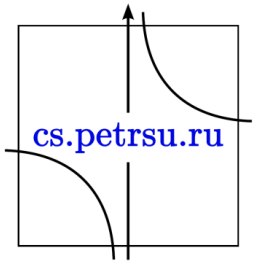


- Вызов функции без аргументов:

```
function sayHi() {
        alert('Привет');
}
setTimeout(sayHi, 1000);       // правильно!
setTimeout(sayHi(), 1000);     // не правильно!
```

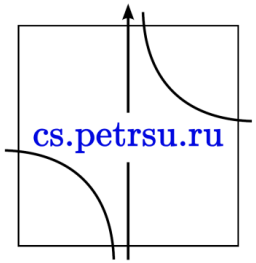
- С аргументами:

```
function sayHi(phrase, who) {
        alert( phrase + ', ' + who + '!' );
}
setTimeout(sayHi, 1000, "Привет", "мир"); // Привет, мир
setTimeout("alert('Привет')", 1000); // допустимый вариант
```

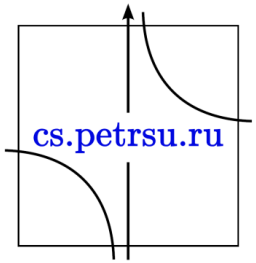


Объекты Math

- Math.acos
- Math.asin
- Math.atan
- Math.atan2
- Math.exp
- Math.min
- Math.random
- Math.sqrt
- Math.log
- Math.round
- Math.floor
- Math.ceil
- Math.sin
- Math.cos
- Math.tan
- Math.pow
- Math.max
- Math.abs



```
Math.random(); // 0.1940108106937259 4

function getRandom() {
        return Math.random();
}
function getRandomFloat(min, max) {
        return Math.random() * (max - min) + min;
}
getRandomFloat(11, 101); // 75.31898734299466


function getRandomInt(min, max) {
        return Math.floor(Math.random() * (max - min)) + min;
}
getRandomInt(10, 20); // 12
```

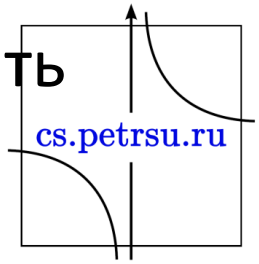


Класс: базовый синтаксис

```
class MyClass {
        // методы класса
        constructor() { ... }
        method1() { ... }
        method2() { ... }
        method3() { ... }
        ...
}
```

В JavaScript класс – это разновидность функции.

- Необходимо использовать вызов new MyClass() для создания нового объекта со всеми перечисленными методами.
- При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.



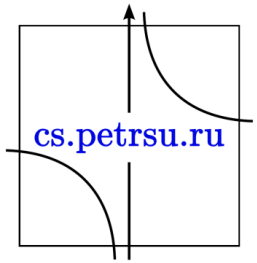
Пример использования

```
class User {
        constructor(name) {
                this.name = name;
        }
        sayHi() {
                alert(this.name);
        }
}
// Использование:
let user = new User("Иван");
user.sayHi();
```

При вызове new User("Иван"):

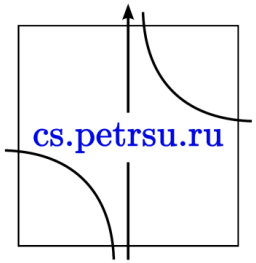
- Создаётся новый объект.
- constructor запускается с заданным аргументом и сохраняет его в this.name.

… и затем можно вызывать на объекте методы, такие как user.sayHi().

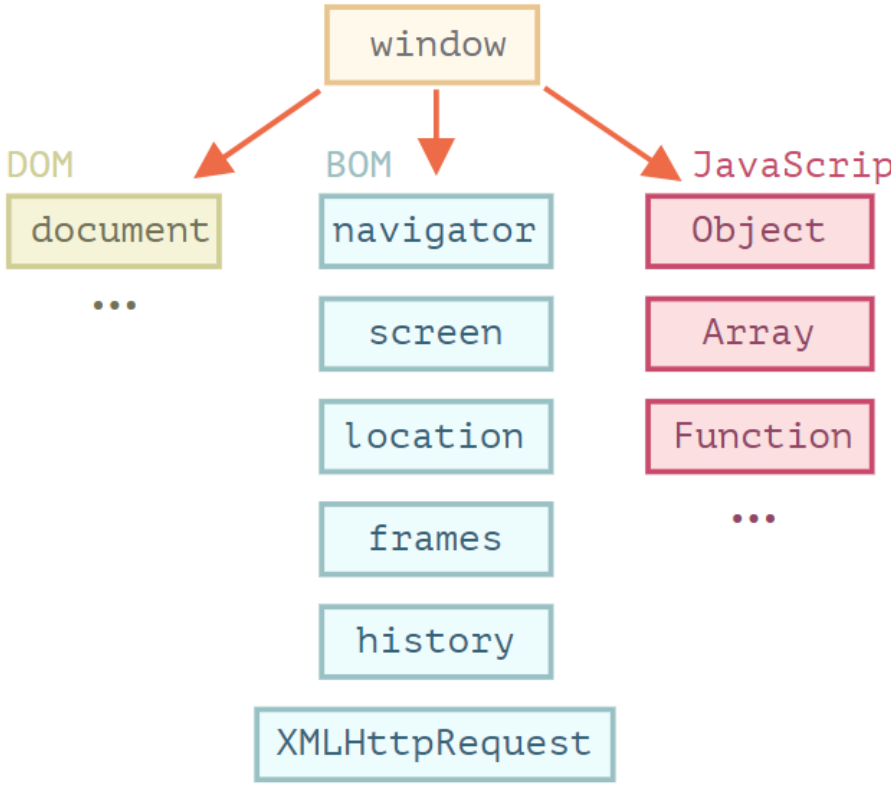


Лекция №6

JAVASCRIPT. DOM



Объект window:



- Это глобальный объект для JavaScript-кода.

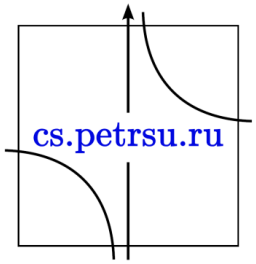
- Также представляет собой окно браузера и располагает методами для управления им.

Объектная модель браузера (Browser Object Model, BOM)

– это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

- Объект navigator даёт информацию о самом браузере и операционной системе (напр.: navigator.userAgent – информация о браузере).
- Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

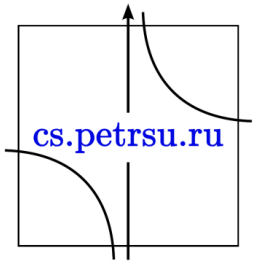
```
alert(location.href); // показывает текущий URL
if (confirm("Перейти на Wikipedia?"))
{
        location.href = "https://wikipedia.org";
        // перенаправляет браузер на другой URL
}
```



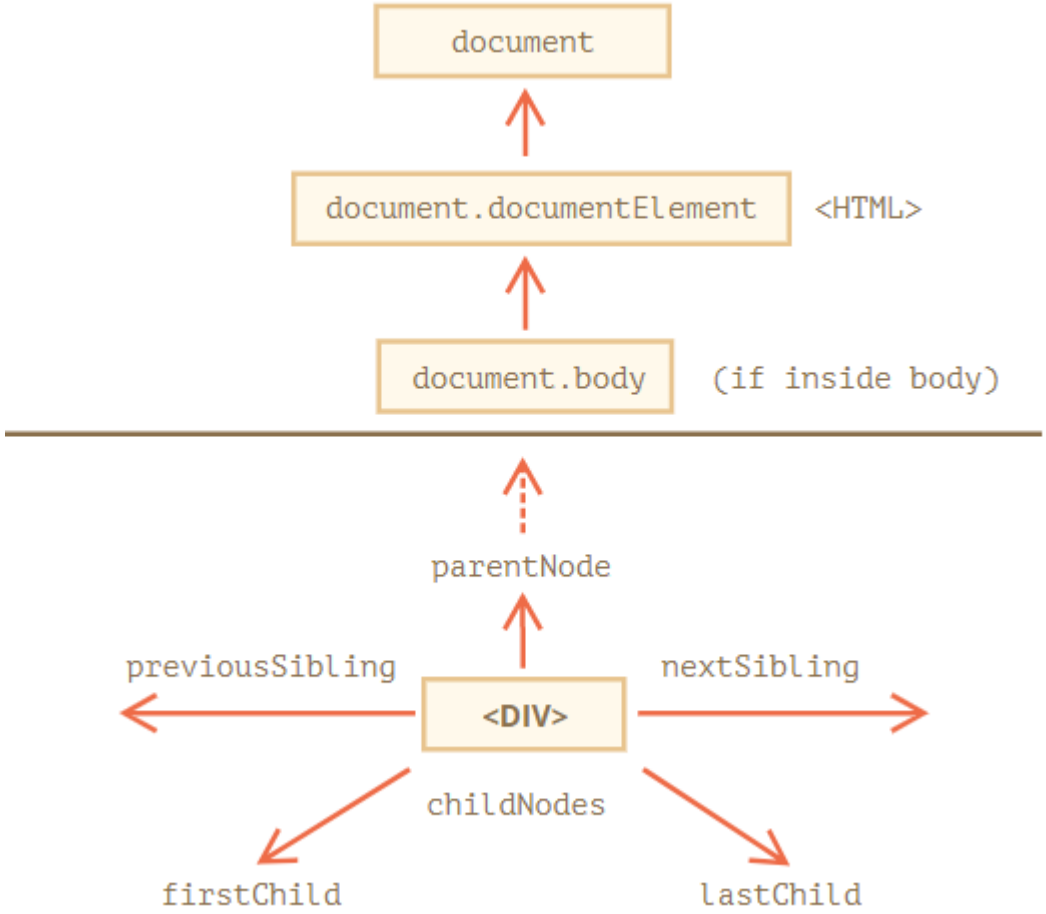
Document Object Model, сокращённо DOM

- – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.
- Объект document – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

```
// заменим цвет фона на красный
document.body.style.background = "red";
```



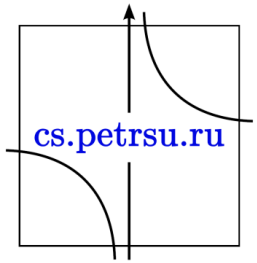
Навигация по DOM-элементам



cs.petrsu.ru

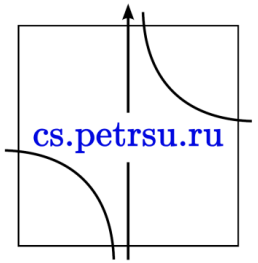
Навигация по DOM-элементам

- Все операции с DOM начинаются с объекта document.
- <html> = document.documentElement
- <body> = document.body
- <head> = document.head



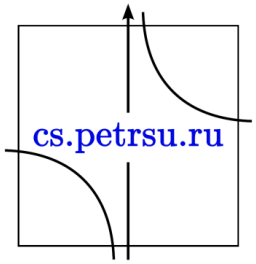
Дети: childNodes, firstChild, lastChild

- Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного.
- Потомки – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.



- Коллекция childNodes содержит список всех детей, включая текстовые узлы

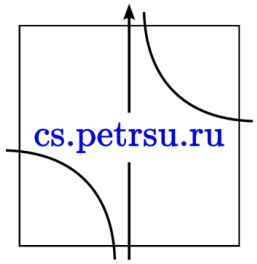
```
<html>
  <body>
    <div>Начало</div>
    <ul>
      <li>Информация</li>
    </ul>
    <div>Конец</div>
    <script>
      for (let i = 0; i < document.body.childNodes.length; i++) {
        alert( document.body.childNodes[i] );
        // [object Text], [object HTMLDivElement], [object Text],
        // [object HTMLUListElement], [object Text],
        // [object HTMLDivElement], [object Text],
        // [object HTMLScriptElement] < Последний, дальше не видит
      }
    </script>
    ...какой-то HTML-код...
  </body>
</html>
```



- Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу.
- По сути, являются всего лишь сокращениями:

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

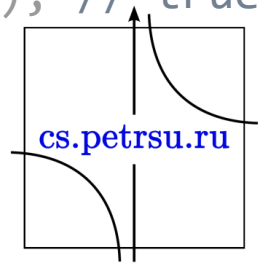
- Для проверки наличия дочерних узлов существует также специальная функция elem.hasChildNodes()



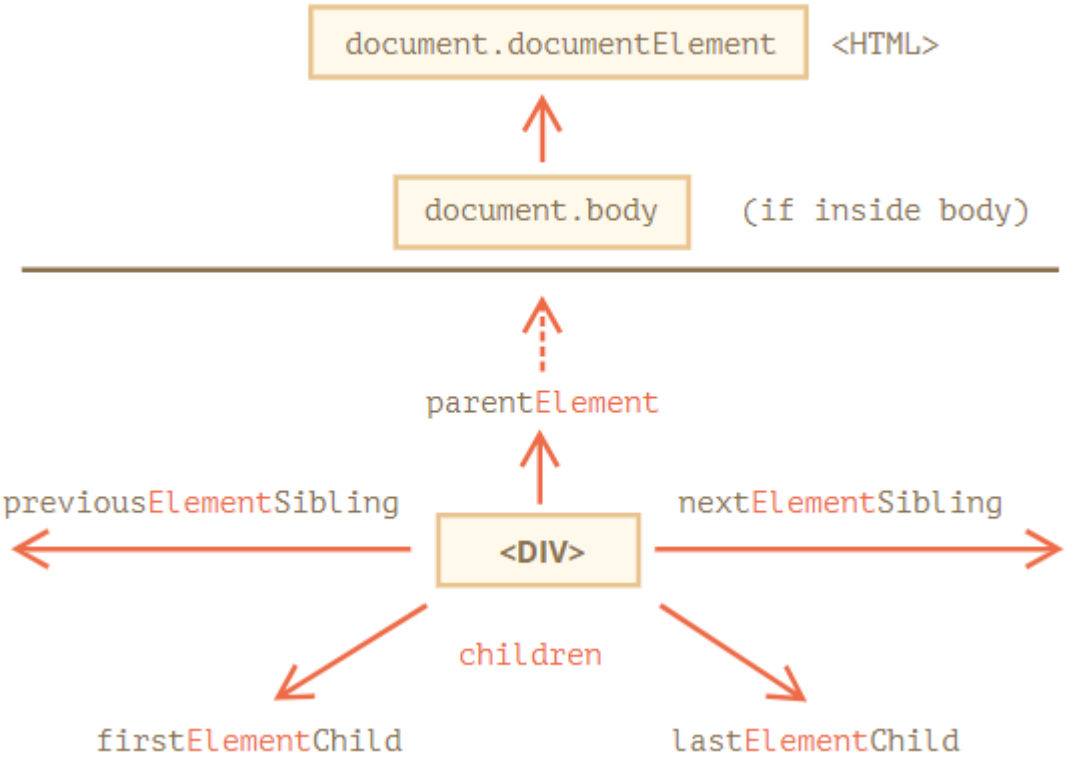
Соседи и родитель

- *Соседи* – это узлы, у которых один и тот же родитель.
- Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling.
- Родитель доступен через parentNode.

```
<html>
  <head>...</head><body>...</body>
</html>
// родителем <body> является <html>
alert( document.body.parentNode === document.documentElement ); // true
// после <head> идёт <body>
alert( document.head.nextSibling ); // HTMLBodyElement
// перед <body> находится <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```



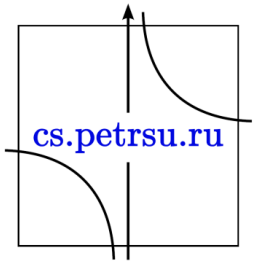
Навигация по элементам:



- Эта картинка похожа на предыдущую, только в ряде мест стоит слово Element

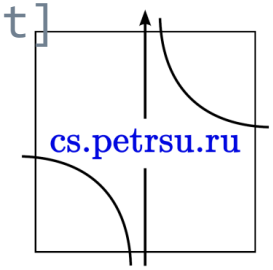
cs.petrsu.ru

- children – коллекция детей, которые являются элементами.
- firstElementChild, lastElementChild – первый и последний дочерний элемент.
- previousElementSibling, nextElementSibling – соседи-элементы.
- parentElement – родитель-элемент.



- Пример только по элементам

```
<html>
  <body>
    <div>Начало</div>
    <ul>
      <li>Информация</li>
    </ul>
    <div>Конец</div>
    <script>
    for (let elem of document.body.children) {
      alert(elem);
      // [object HTMLDivElement], [object HTMLUListElement],
      // [object HTMLDivElement], [object HTMLScriptElement]
    }
    </script>
    ...
  </body>
</html>
```



Элемент <table>

- table.rows – коллекция строк <tr> таблицы.
- table.caption/tHead/tFoot – ссылки на элементы таблицы <caption>, <thead>, <tfoot>.
- table.tBodies – коллекция элементов таблицы <tbody> (по спецификации их может быть больше одного).

<thead>, <tfoot>, <tbody> предоставляют свойство rows:

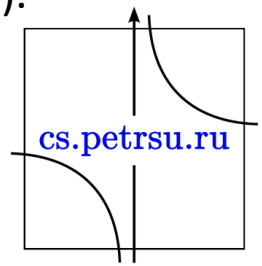
- tbody.rows – коллекция строк <tr> секции.

<tr>:

- tr.cells – коллекция <td> и <th> ячеек, находящихся внутри строки <tr>.
- tr.sectionRowIndex – номер строки <tr> в текущей секции <thead>/<tbody>/<tfoot>.
- tr.rowIndex – номер строки <tr> в таблице (включая все строки таблицы).

<td> and <th>:

- td.cellIndex – номер ячейки в строке <tr>.

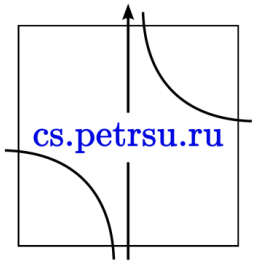


Поиск: getElement*, querySelector*

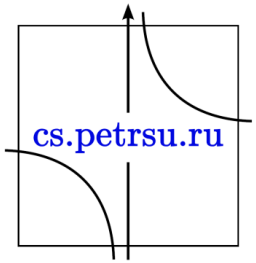
- Если у элемента есть атрибут id, то мы можем получить его вызовом document.getElementById(id).
- Также есть глобальная переменная с именем, указанным в id.

(!) Но только если не объявлена переменная с таким же именем, Использовать не рекомендуется.

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>
<script>
  // получить элемент
  let red = document.getElementById('elem');
  // сделать его фон красным
  red.style.background = 'red';
  // или так
  elem.style.background = 'red';
  window['elem-content'].style.background = 'red';
</script>
```



- Значение id должно быть уникальным. В документе может быть только один элемент с данным id.
- Метод getElementById можно вызвать только для объекта document. Он осуществляет поиск по id по всему документу.



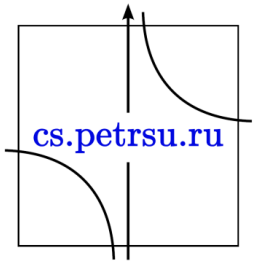
querySelectorAll

- Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

```
<script>
  let elements = document.querySelectorAll('ul > li:last-child');
  for (let elem of elements) {
    alert(elem.innerHTML); // "тест", "пройден"
  }
</script>
```

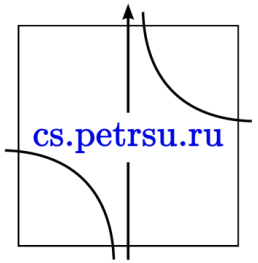
Псевдоклассы в CSS-селекторе, в частности :hover и :active, также поддерживаются.

Например, document.querySelectorAll(':hover') вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.



querySelector

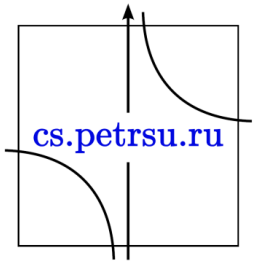
- Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.
- Аналогичен elem.querySelectorAll(css)[0]



matches

- Метод elem.matches(css) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false.

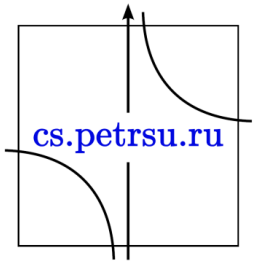
```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // может быть любая коллекция вместо document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("Ссылка на архив: " + elem.href );
    }
  }
</script>
```



closest

- Предки элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины.
- Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск.

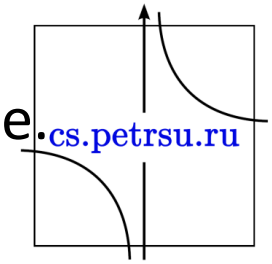
```
<h1>Содержание</h1>
<div class="contents">
<ul class="book">
  <li class="chapter">Глава 1</li>
  <li class="chapter">Глава 2</li>
</ul>
</div>
<script>
  let chapter = document.querySelector('.chapter'); // LI
  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV
  alert(chapter.closest('h1')); // null (потому что h1 - не предок)
</script>
```



getElementsBy*

На данный момент, они скорее исторические, так как querySelector более чем эффективен.

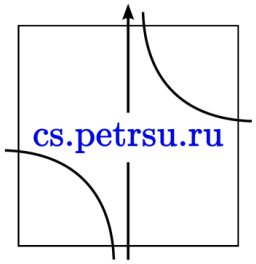
- elem.getElementsByTagName(tag)
  - ищет элементы с данным тегом и возвращает их коллекцию. Передав "*" вместо тега, можно получить всех потомков.
- elem.getElementsByClassName(className)
  - возвращает элементы, которые имеют данный CSS-класс.
- document.getElementsByName(name)
  - возвращает элементы с заданным атрибутом name. Очень редко используется.



```
<script>
  // получить все элементы div в документе
  let divs = document.getElementsByTagName('div');
</script>

<script>
  let inputs = table.getElementsByTagName('input');
  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

(!) Возвращают коллекцию, а не элемент

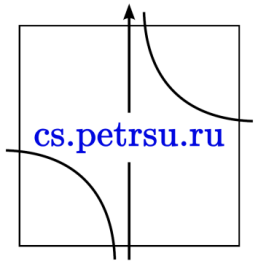
```
// не работает
document.getElementsByTagName('input').value = 5;
// работает (если есть input)
document.getElementsByTagName('input')[0].value = 5;
```



Живые коллекции

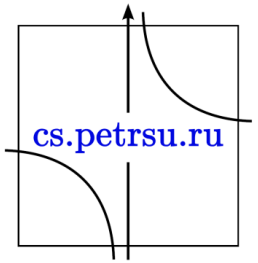
- Живые коллекцииВсе методы "getElementsBy*" возвращают живую коллекцию.
- Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении.

```
<div>First div</div>
<script>
  let divs = document.getElementsByTagName('div');
  alert(divs.length); // 1
</script>
<div>Second div</div>
<script>
  alert(divs.length); // 2
</script>
```



- querySelectorAll возвращает статическую коллекцию. Это похоже на фиксированный массив элементов.

```
<div>First div</div>
<script>
  let divs = document.querySelectorAll('div');
  alert(divs.length); // 1
</script>
<div>Second div</div>
<script>
  alert(divs.length); // 1
</script>
```



Метод	Ищет по...	Ищет внутри элемента?	Возвращает живую коллекцию?
querySelector	CSS-selector	✓	-
querySelectorAll	CSS-selector	✓	-
getElementById	id	-	-
getElementsByName	name	-	✓
getElementsByTagName	tag or '*'	✓	✓
getElementsByClassName	class	✓	✓

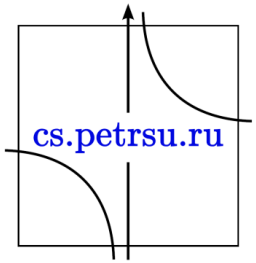
Свойства узлов: тип, тег и содержимое

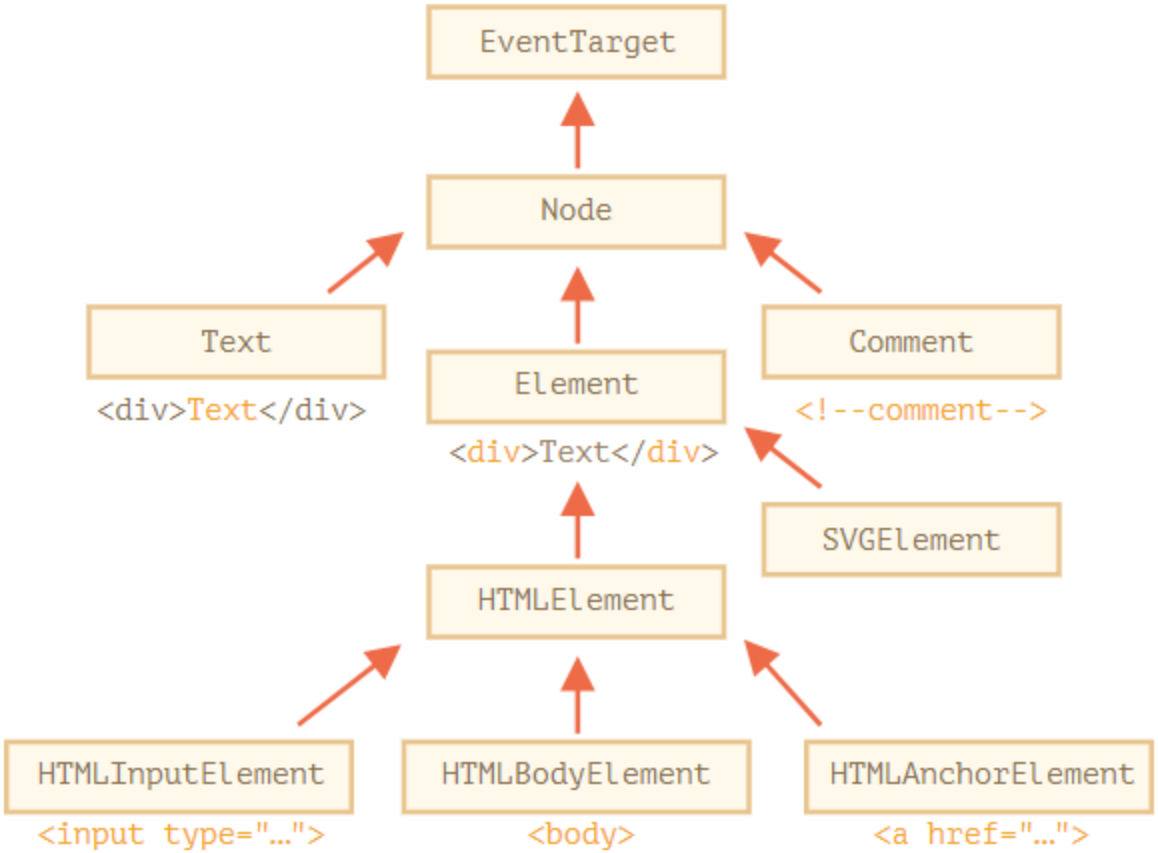
- У разных DOM-узлов могут быть разные свойства.

  Например, у узла, соответствующего тегу <a>, есть свойства, связанные со ссылками, а у соответствующего тегу <input> – свойства, связанные с полем ввода и т.д.

  Текстовые узлы отличаются от узлов-элементов.

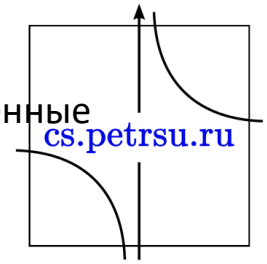
- У всех DOM-узлов есть общие свойства и методы, потому что все классы образуют единую иерархию.
- Каждый DOM-узел принадлежит соответствующему встроенному классу.
- Корнем иерархии является EventTarget



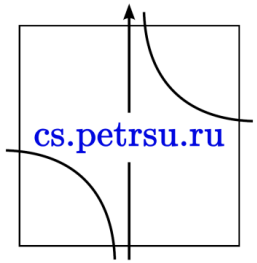

EventTarget
Node
Text
Text
Element
Text
Comment
<!--comment-->
SVGElement
HTMLElement
HTMLInputElement
<input type="…">
HTMLBodyElement
<body>
HTMLAnchorElement
<a href="…">

Существуют следующие классы:

- EventTarget – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события».
- Node – также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определённые классы узлов, которые наследуют от него: Text – для текстовых узлов, Element – для узлов-элементов и более экзотический Comment – для узлов-комментариев.
- Element – это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также XML и SVG. Класс Element служит базой для следующих классов: SVGElement, XMLElement и HTMLElement.
  - HTMLElement – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
  - HTMLInputElement – класс для тега <input>,
  - HTMLBodyElement – класс для тега <body>,
  - HTMLAnchorElement – класс для тега <a>,
  - …и т.д, каждому тегу соответствует свой класс, который предоставляет определённые свойства и методы.



- Тег: nodeName и tagName
- innerHTML: содержимое элемента
- outerHTML: HTML элемента целиком
- nodeValue/data: содержимое текстового узла
- textContent: просто текст
- Свойство «hidden»
- Другие свойства
  - value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement…).
  - href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement).
  - id – значение атрибута «id» для всех элементов (HTMLElement).



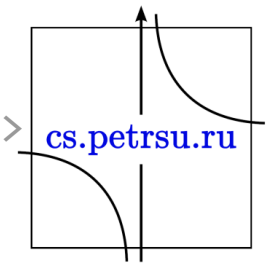
Динамическое редактирование Web-страницы

- С помощью функций можно изменять или возвращать содержимое блочных элементов:

  innerText - содержит весь текст блочного элемента

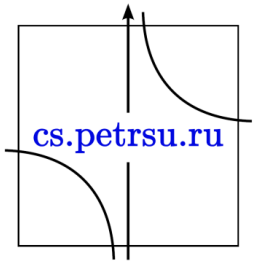
  innerHTML - содержит не только текст, но и другие элементы HTML

```
<button onclick="alert(this.innerHTML)">Нажми меня</button>
```

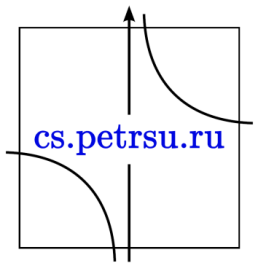


Динамическое редактирование Web -страницы

- **createElement() -** создание элементов Web-страниц
  - **document.createElement('p')**
  - **document.createElement('table')**
  - **document.createElement('td')**
  - и т.д.
- **appendChild()** - добавление этого элемента на Web-страницу.



- Динамическое изменение таблиц:
  - **insertRow(***номер_строки*)
  - **insertCell(***номер_ячейки)*
- *Динамическое изменение списков:*
  - **options.add**(*объект_пункта, индекс).*

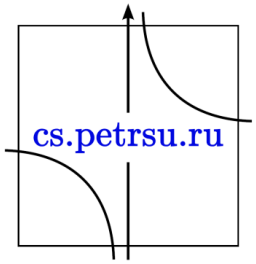


Примеры

```
<input type="text" id="elem" value="значение">
<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // значение
</script>

<div id="elem1"></div>
<div id="elem2"></div>
<script>
  let name = prompt("Введите ваше имя?", "<b>Винни-пух!</b>");
  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

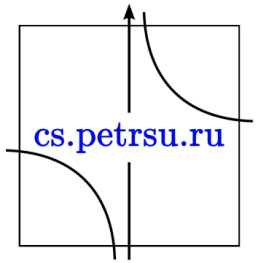
Винни-пух!

Винни-пух!



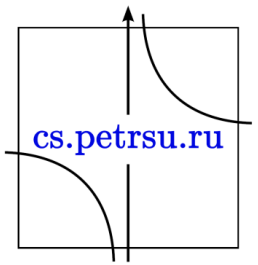
Обработчики событий

- *Событие* – это сигнал от браузера о том, что что-то произошло.
- Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).



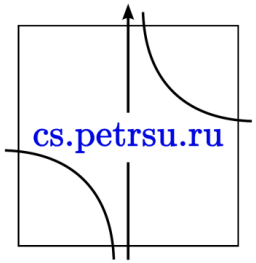
События мыши

- click – происходит, когда нажали на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
- contextmenu – происходит, когда нажали на элемент правой кнопкой мыши.
- mouseover / mouseout – когда мышь наводится на / покидает элемент.
- mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
- mousemove – при движении мыши.
- dblclick – двойное нажатие на элементе.



События на элементах управления:

- submit – пользователь отправил форму <form>.
- focus – вызывается в момент фокусировки на элементе, например нажимает на <input>.
- blur – потеря фокуса на элементе.
- change - срабатывает по окончании изменения элемента.
- input - срабатывает каждый раз при изменении значения (! происходит после изменения).
- cut, copy, paste - происходят при вырезании/копировании/вставке данных.
- reset – срабатывает при сбросе формы.

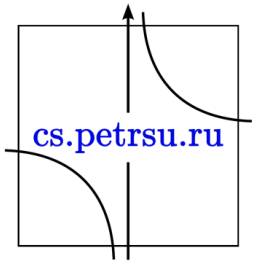


Клавиатурные события:

- keydown и keyup – когда пользователь нажимает / отпускает клавишу.
- keypress – пльзователь удерживает кнопку нажатой.

События документа:

- DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.
- load – загрузка документа или изображение.
- error – ошибка загрузки документа или изображения.
- abort – отказ пользователя от загрузки изображения.
- resize – изменение размеров окна.
- move – перемещение окна.
- dragdrop – перетаскивание мышью объекта в окне браузера.

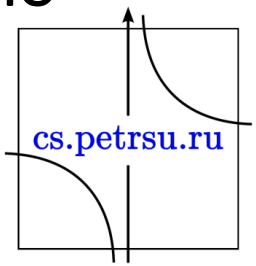


Обработчики событий

- Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло.

- Благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

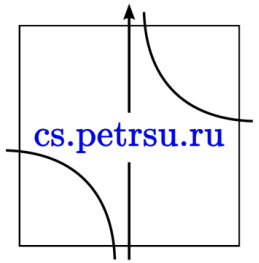
- Есть несколько способов назначить событию обработчик.



1. Использование атрибута HTML

- Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>:

```
<input value="Нажми меня" onclick="alert('Клик!')" type="button">


<script>
  function hello() {
    alert("Hello, world!");
  }
</script>
<input type="button" onclick="hello()" value="Кнопка">
```

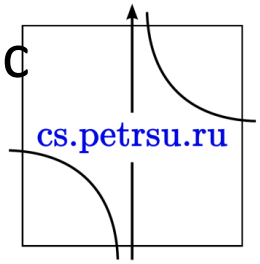


2. Использование свойства DOM-объекта

- Можно назначать обработчик, используя свойство DOM-элемента on<событие>:

```
<input id="el" type="button" value="Нажми меня!">
<script>
  el.onclick = function() {
    alert('Спасибо');
  };
</script>
```

- Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.
- У элемента DOM может быть только одно свойство с именем onclick, поэтому назначить более одного обработчика onclick так нельзя.



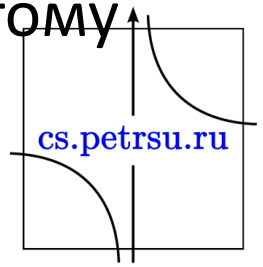
- Обработчиком можно назначить и уже существующую функцию:

```
function sayThanks() {
  alert('Спасибо!');
}
elem.onclick = sayThanks;
```

- Убрать обработчик можно назначением: elem.onclick = null.

- (!) Регистр DOM-свойства имеет значение.

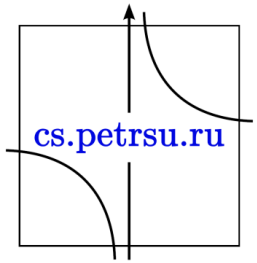
  Используйте elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.



3. addEventListener

- Основной недостаток способов 1 и 2 – невозможность повесить несколько обработчиков на одно событие.
- Специальные методы addEventListener и removeEventListener:

```
element.addEventListener(event, handler[, options]);

element.removeEventListener(event, handler[, options]);
```



```
element.addEventListener(event, handler[, options]);
```

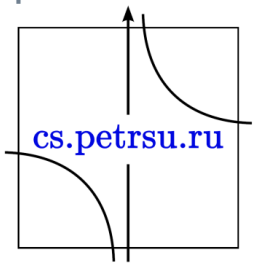
- event - Имя события, например "click".
- handler - Ссылка на функцию-обработчик.
- options - Дополнительный объект со свойствами:

  once: если true, тогда обработчик будет автоматически удалён после выполнения.

  capture: фаза, на которой должен сработать обработчик, подробнее об этом будет рассказано в главе Всплытие и погружение {false/true}.

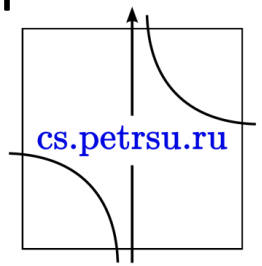
  passive: если true, то указывает, что обработчик никогда не вызовет preventDefault().

```
<input id="elem" type="button" value="Нажми меня"/>

<script>
  function handler1() {
    alert('Спасибо!');
  };
  function handler2() {
    alert('Спасибо ещё раз!');
  }

  elem.onclick = () => alert("Привет");
  elem.addEventListener("click", handler1); // Спасибо!
  elem.addEventListener("click", handler2); // Спасибо ещё раз!
</script>
```



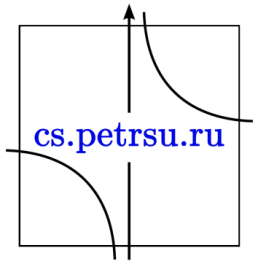
Объект события

- Когда происходит событие, браузер создаёт объект события **event**, записывает в него детали события и передаёт его в качестве аргумента функции-обработчику.
  - event.type - тип события, в данном случае "click".
  - event.currentTarget - элемент, на котором сработал обработчик.
  - event.clientX / event.clientY - координаты курсора в момент клика относительно окна, для событий мыши.
  - И другие свойства …



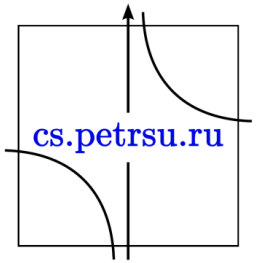
Методы для генерации событий:

- Submit()
- Click()
- Blur()
- Focus()



Лекция №7

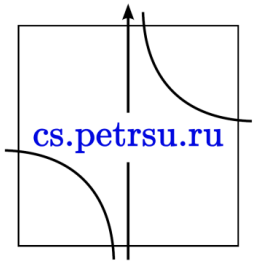
JAVASCRIPT. CANVAS



Элемент <canvas>

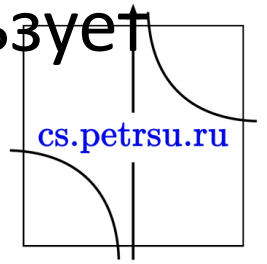
```
<canvas id="tutorial" width="150" height="150"></canvas>
```

- Два атрибута: width и height.
  - не обязательны, могут быть выставлены с использованием свойств DOM, задание через CSS не рекомендуется – ведет к проблемам масштабирования.
- Требует закрывающий тег </canvas>.
- По умолчанию абсолютно прозрачный.



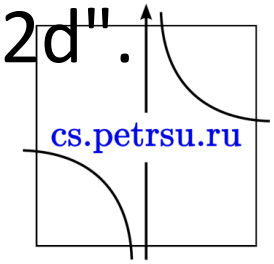
Рендеринг содержимого (контекста)

- Элемент <canvas> создаётся с фиксированным размером элемента для рисования, который может иметь один или несколько контекстов для рендеринга, создавая и манипулируя содержимым для показа.
- На практике мы будем использовать только контекст 2D.
- Другие контексты могут предоставлять разные типы рендеринга, к примеру WebGl использует 3D контекст основанный на OpenGL ES.



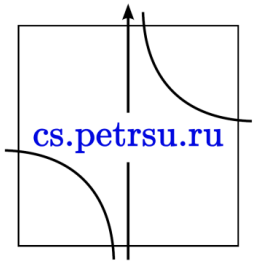
Рендеринг содержимого (контекста)

- Холст изначально пустой и прозрачный.
- Первым делом скрипт получает доступ к контексту и отрисовывает его.
- Элемент <canvas> имеет метод getContext(), используется для получения контекста визуализации и её функции рисования. getContext() принимает один параметр, тип контекста.
- Для 2D графики, которая охвачена этим руководством будем использовать метку "2d".



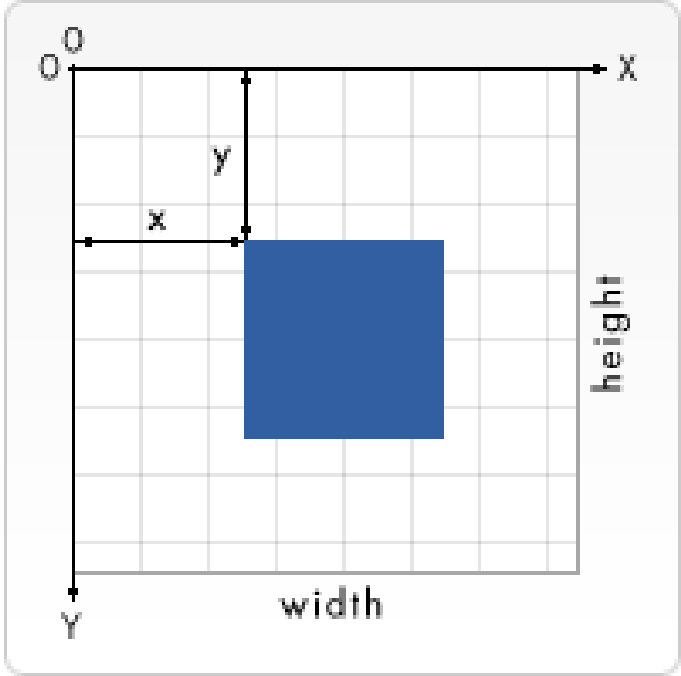
Проверка поддержки

```
<canvas id="tutorial" width="150" height="150"></canvas>
<script>
  let canvas = document.getElementById('tutorial');
  if (canvas.getContext){
    let ctx = canvas.getContext('2d');
    // тут код для рисования
  } else {
    // блок canvas не поддерживается
  }
</script>
```

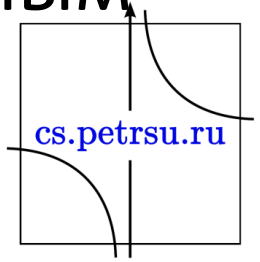


Рисование фигур с помощью

- Сетка canvas или координатная плоскость:
  - Начало координат этой сетки расположено в верхнем левом углу в координате (0,0).
  - Все элементы размещены относительно этого начала.
  - Таким образом, положение верхнего левого угла синего квадрата составляет x пикселей слева и y пикселей сверху, на координате (x, y).



- <canvas> поддерживает только одну примитивную фигуру: прямоугольник.
- Другие фигуры должны быть созданы комбинацией одного или большего количества контуров (paths), набором точек, соединённых в линии.
- В ассортименте рисования контуров у нас есть функции, которые делают возможным составление очень сложных фигур.

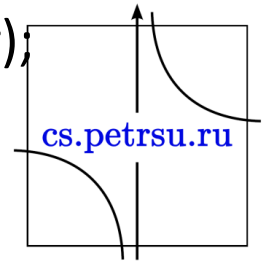


Прямоугольник:

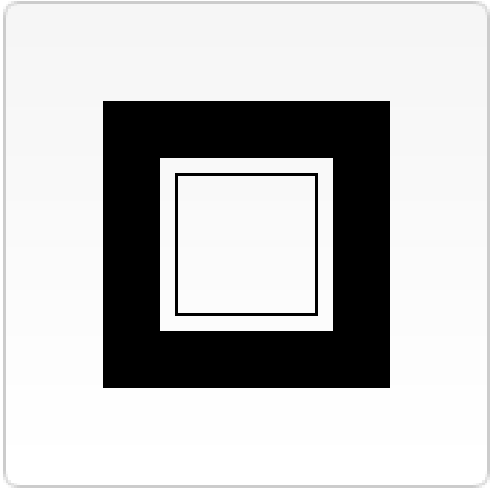
- fillRect(x, y, width, height) - рисование заполненного прямоугольника.
- strokeRect(x, y, width, height) - рисование прямоугольного контура.
- clearRect(x, y, width, height) - очистка прямоугольной области, делая содержимое совершенно прозрачным.
- rect(x, y, width, height) - добавляет прямоугольник, верхний левый угол которого указан с помощью (x, y) с вашими width и height

Параметры:

- x, y - устанавливают положение верхнего левого угла прямоугольника в canvas (относительно начала координат);
- width(ширина) и height(высота) определяют размеры прямоугольника.



```
function draw() {
  let canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    let ctx = canvas.getContext('2d');

    ctx.fillRect(25,25,100,100);
    ctx.clearRect(45,45,60,60);
    ctx.strokeRect(50,50,50,50);
  }
}
```

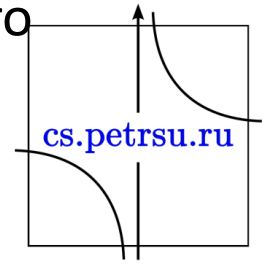


Рисование контуров (path)

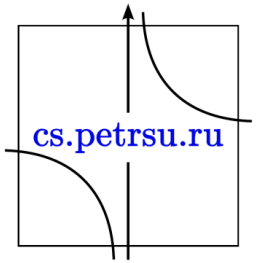
- Остальные примитивные фигуры создаются контурами.
- Контур - это набор точек, которые, соединяясь в отрезки линий, могут образовывать различные фигуры, изогнутые или нет, разной ширины и разного цвета.
- Контур (или субконтур) может быть закрытым.

Шаги создания фигур используя контуры :

1. Сначала создать контур.
2. Затем, используя команды рисования, нарисовать контур.
3. Потом закрывать контур.
4. Созданный контур можете обвести или залить для его отображения.



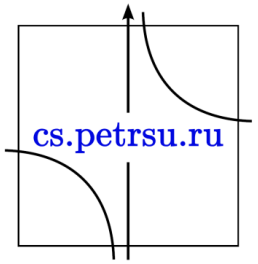
- beginPath() – создаёт новый контур. После создания используется в дальнейшем командами рисования при построении контуров.
- closePath() – закрывает контур, так что будущие команды рисования вновь направлены контекст.
- stroke() – рисует фигуру с внешней обводкой.
- fill() – рисует фигуру с заливкой внутренней области.



Передвижение пера

- moveTo(x, y) – перемещает перо в точку с координатами x и y.
  - Очень полезная функция, которая ничего не рисует, а осуществляет перемещение пера (отрыв пера от холста и его перемещение в указанное место).

- При инициализации canvas или при вызове beginPath(), вам надо будет использовать функцию moveTo() для перемещения в точку начала рисования.
- Можно использовать moveTo() и для рисования несвязанного(незакрытого) контура.



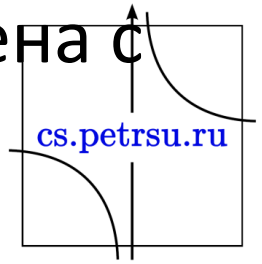
Пример с незамкнутым контуром

```
function draw() {
  let canvas = document.getElementById('canvas');
  if (canvas.getContext){
    let ctx = canvas.getContext('2d');
    ctx.beginPath();
    ctx.arc(75,75,50,0,Math.PI*2,true); // Внешняя окружность
    ctx.moveTo(110,75);
    ctx.arc(75,75,35,0,Math.PI,false); // рот (по часовой стрелке)
    ctx.moveTo(65,65);
    ctx.arc(60,65,5,0,Math.PI*2,true); // Левый глаз
    ctx.moveTo(95,65);
    ctx.arc(90,65,5,0,Math.PI*2,true); // Правый глаз
    ctx.stroke();
  }
}
```

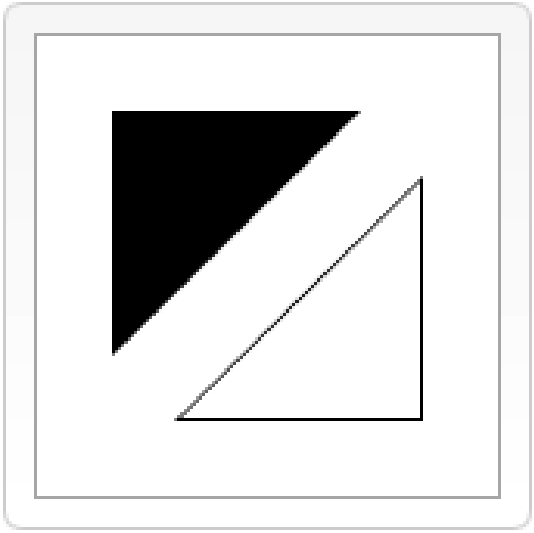
- Если вы захотите увидеть соединение линии, то можете удалить вызов moveTo().

Линии

- lineTo(x, y) – рисует линию с текущей позиции до позиции, определённой x и y.
- Этот метод принимает два аргумента x и y, которые являются координатами конечной точки линии.
- Начальная точка зависит от ранее нарисованных путей, причём конечная точка предыдущего пути является начальной точкой следующего и т. д.
- Начальная точка также может быть изменена с помощью метода moveTo().



```
function draw() {
  let canvas = document.getElementById('canvas');
  if (canvas.getContext){
    let ctx = canvas.getContext('2d');
    ctx.beginPath();
    ctx.moveTo(25,25);
    ctx.lineTo(105,25);
    ctx.lineTo(25,105);
    ctx.fill();

    ctx.beginPath();
    ctx.moveTo(125,125);
    ctx.lineTo(125,45);
    ctx.lineTo(45,125);
    ctx.closePath();
    ctx.stroke();
  }
}
```

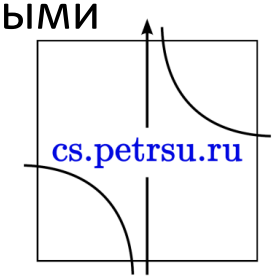


cs.petrsu.ru

Дуги

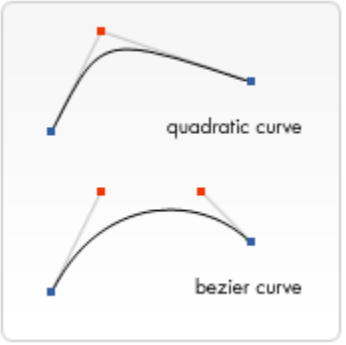
- arc(x, y, radius, startAngle, endAngle, anticlockwise) – рисует дугу с центром в точке (x,y) радиусом radius, начиная с угла startAngle и заканчивая в endAngle в направлении против часовой стрелки anticlockwise (по умолчанию по ходу движения часовой стрелки).
  - параметры: x и y — это координаты центра окружности, в которой должна быть нарисована дуга. radius —радиус окружности. Углы startAngle и endAngle определяют начальную и конечную точки дуги в радианах вдоль кривой окружности. Отсчёт происходит от оси x. Параметр anticlockwise — логическое значение, которое, если true, то рисование дуги совершается против хода часовой стрелки; иначе по ходу часовой стрелки.
  - Углы в функции arc() измеряют в радианах, не в градусах. Для перевода градусов в радианы вы можете использовать JavaScript-выражение: radians = (Math.PI/180)*degrees

- arcTo(x1, y1, x2, y2, radius) – рисует дугу с заданными контрольными точками и радиусом, соединяя эти точки прямой линией.

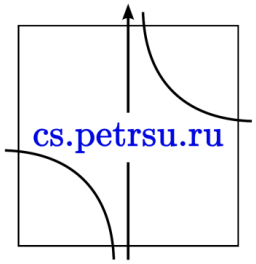


Безье и квадратичные кривые

- quadraticCurveTo(cp1x, cp1y, x, y) – рисуется квадратичная кривая Безье с текущей позиции пера в конечную точку с координатами x и y, используя контрольную точку с координатами cp1x и cp1y.
- bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y) – рисуется кубическая кривая Безье с текущей позиции пера в конечную точку с координатами x и y, используя две контрольные точки с координатами (cp1x, cp1y) и (cp2x, cp2y).
  - Квадратичная кривая Безье имеет стартовую и конечную точки (синие точки) и всего одну контрольную точку (красная точка), в то время как кубическая кривая Безье использует две контрольные точки.
  - Параметры x и y в этих двух методах являются координатами конечной точки;
    cp1x и cp1y — координаты первой контрольной точки;
    cp2x и cp2y — координаты второй контрольной точки.



- clip() – Обрезает область любой формы и размера, находящуюся вне указанного контура.
  - после обрезки области все последующее рисование будет ограничено обрезанной областью (к другим областям холста доступа не будет). Тем не менее, перед использованием метода **clip()** можно сохранить текущую область холста при помощи метода **save()**, а затем в любое время восстановить ее при помощи метода **restore()**.
- isPointInPath(x,y) – Возвращает значение true, если заданная точка находится внутри текущего контура, в обратном случае возвращается значение false

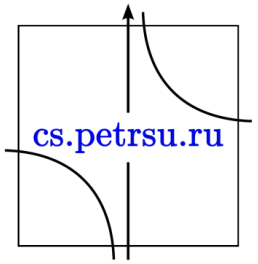


Path2D объекты

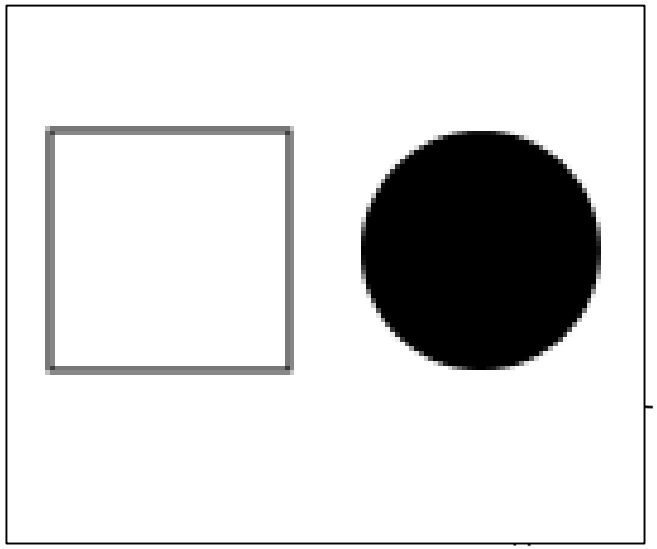
- Path2D() – конструктор возвращает вновь созданный объект Path2D.
  - Все методы path , такие как moveTo, rect, arc, или quadraticCurveTo, и т.п., доступны для объектов Path2D

```
new Path2D(); // пустой path объект
new Path2D(path); // копирование из другого path
new Path2D("M10 10 h 80 v 80 h -80 Z"); // path из SVG
/* Путь перемещается в точку (M10 10), а затем горизонтально перемещается
 * на 80 пунктов вправо (h 80), затем на 80 пунктов вниз (v 80), затем
 * на 80 пунктов влево (h -80), а затем обратно на start (z).
 */
```

- Path2D.addPath(path [, transform]) – добавляет путь к текущему пути с необязательной матрицей преобразования.
  - Это полезно при создании объекта из нескольких компонентов.



```
function draw() {
  let canvas = document.getElementById('canvas');
  if (canvas.getContext){
    let ctx = canvas.getContext('2d');
    let rectangle = new Path2D();
    rectangle.rect(10, 10, 50, 50);

    let circle = new Path2D();
    circle.moveTo(125, 35);
    circle.arc(100, 35, 25, 0, 2 * Math.PI);

    ctx.stroke(rectangle);
    ctx.fill(circle);
  }
}
```

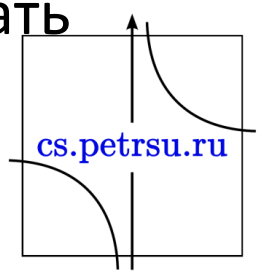


Изображения

- Внешние изображения могут быть использованы в любых поддерживаемых браузером форматах, таких как PNG, GIF, или JPEG.
- Вы можете использовать изображение, произведённое другими canvas элементами на той же странице.

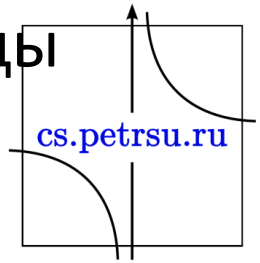
Импортирование изображений в canvas состоит из 2 этапов:

1. Дать ссылку на HTMLImageElement объект, canvas или ссылку на URL.
2. Для рисования изображения на canvas использовать функцию drawImage().



Источники изображений

- HTMLImageElement – изображения созданные, конструктором Image(), такой же как все <img> элементы.
- HTMLVideoElement – используя HTML <video> элемент как источник изображения захватывает текущий кадр из видео и использует его как изображение.
- HTMLCanvasElement – использовать другой <canvas> элемент как источник изображения.
- Использование изображений из той же страницы (document.images, document.getElementById).

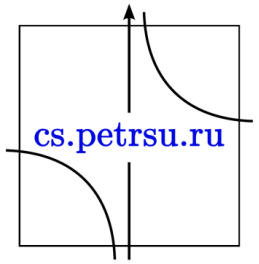


Создание изображений с нуля

- Создать новый HTMLImageElement объект в скрипте с помощью конструктора Image() :

```
var img = new Image();   // Создаёт новый элемент изображения
img.src = 'myImage.png'; // Устанавливает путь
// Когда этот скрипт выполнится, изображение начнёт загружаться.
// Можем получить проблемы если изображение не успеет загрузиться
```

```
var img = new Image(); // Создаёт новое изображение
img.addEventListener("load", function() {
  // здесь выполнить drawImage функцию
}, false);
img.src = 'myImage.png'; // Устанавливает источник файла
```

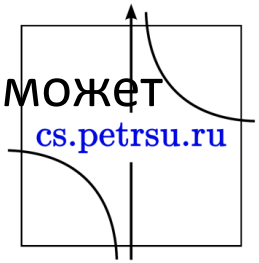


Вложение изображения с помощью Data:URLs

- Data URLs позволяет вам полностью определить изображение как Base64 кодированную строку символов прямо в ваш код:

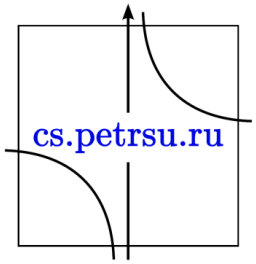
```
var img = new Image(); // Создаёт новый элемент img
img.src = '
AAAALAAsAAAIUhA+hkcuO4lmNVindo7qyrIXiGBYAOw==';
```

+ Полученное изображение доступно сразу без других запросов на сервер.
+ Можно инкапсулировать всё в одном файле.
— Для изображений с большим размером кодирование url может стать очень долгим процессом.



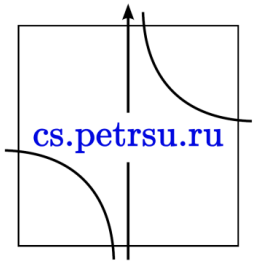
Рисование изображений

- Как только получена ссылка на источник объекта изображения, можно использовать метод drawImage() для включения его в <canvas>.
- drawImage(image, x, y) - рисует изображение, указанное в CanvasImageSource в координатах (x, y).
  - SVG изображения должны указывать ширину и высоту корневого <svg> элемента.



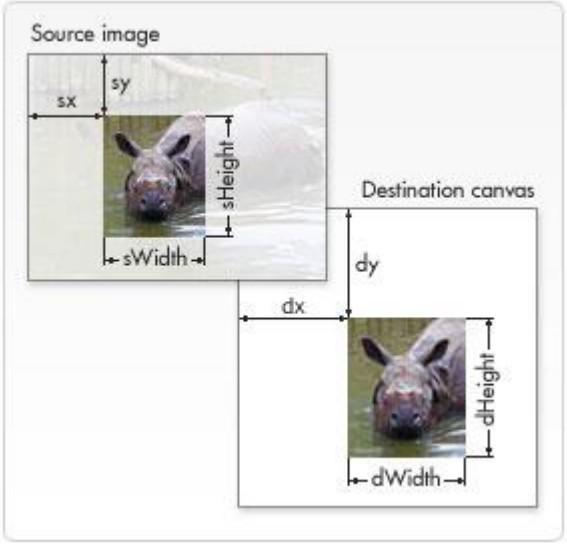
Изменение размеров

- Второй вариант метода drawImage() добавляет два новых параметра и позволяет разместить изображение в canvas с изменёнными размерами:
- drawImage(image, x, y, width, height) – добавились параметра ширины и высоты, которые указывают до какого размера нужно изменить изображение при рисовании его в <canvas>.



Нарезка

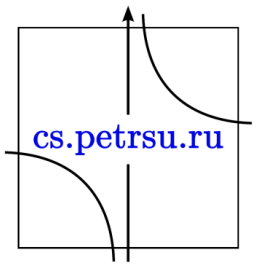
- drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight) – функция берёт фрагмент из изображения, в виде прямоугольника, левый верхний угол которого - (sx, sy), ширина и высота - sWidth и sHeight и рисует в <canvas>, располагая его в точке (dx, dy) и изменяя его размер на указанные величины в dWidth и dHeight.



Контроль изменений размеров изображения

- Изменение размеров изображений может привести к размытости или к шуму в процессе преобразования.
- Вы можете использовать контекст рисования imageSmoothingEnabled свойства, чтобы контролировать использование сглаживающего алгоритма, когда изменяющиеся изображения в вашем контексте.
- Обычно это свойство установлено в true, означая, что изображения будут сглажены во время изменения размеров.
- Вы можете отключить это свойство:

```
ctx.mozImageSmoothingEnabled = false;
ctx.webkitImageSmoothingEnabled = false;
ctx.msImageSmoothingEnabled = false;
ctx.imageSmoothingEnabled = false;
```

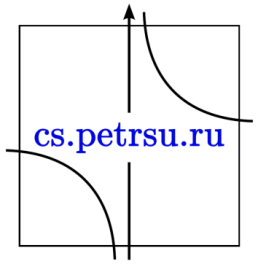


Цвета

Если мы хотим применить цвета к фигуре, то есть два свойства, которые можно использовать: fillStyle и strokeStyle.

- fillStyle = color – устанавливает стиль для фона фигур.
- strokeStyle = color – устанавливает стиль контура фигуры.
  - Где color может быть цветом, (строка, представленная в CSS <color>), градиентом или паттерном. По умолчанию цвет фона и контура — чёрный (значение CSS цвета #000000).
  - Когда устанавливается значения strokeStyle / fillStyle, то это значение становится стандартным для всех следующих фигур. Если далее нужен другой цвет, необходимо перезаписать значение в fillStyle или в strokeStyle.

```
ctx.fillStyle = "orange";
ctx.fillStyle = "#FFA500";
ctx.fillStyle = "rgb(255,165,0)";
ctx.fillStyle = "rgba(255,165,0,1)";
```

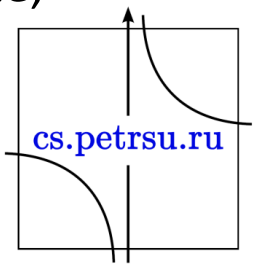


Прозрачность

Рисование прозрачные (полупрозрачные) фигуры реализовано через установку свойства globalAlpha или задачи полупрозрачного цвета фона или контура.

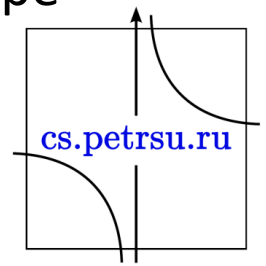
- globalAlpha = transparencyValue – указывается значения прозрачности для всех будущих фигур, что будут нарисованы на canvas.
  - Значение полупрозрачности могут быть между 0.0 (полная прозрачность) и 1.0 (полная непрозрачность). Значение 1.0 (полная непрозрачность) установлено по умолчанию.
  - Свойство globalAlpha может быть использовано, если вы хотите рисовать формы с одинаковой прозрачностью, но обычно устанавливают прозрачность индивидуально к каждой форме, когда указывают их цвет.

```
ctx.strokeStyle = "rgba(255,0,0,0.5)";
ctx.fillStyle = "rgba(255,0,0,0.5)";
```



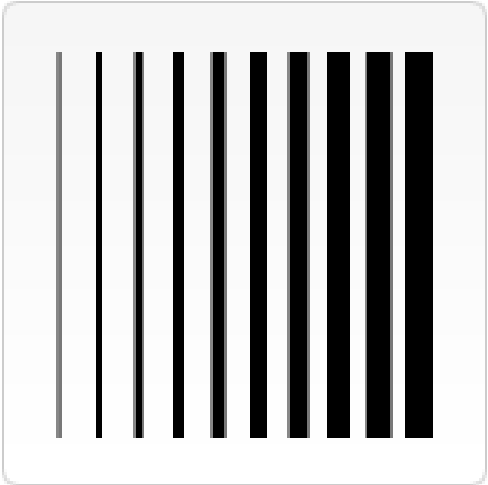
Стили линий

- lineWidth = value – устанавливает ширину линий, рисуемых в будущем.
- lineCap = type – устанавливает внешний вид концов линий.
- lineJoin = type – устанавливает внешний вид «углов», где встречаются линии.
- miterLimit = value – устанавливает ограничение на митру, когда две линии соединяются под острым углом, чтобы вы могли контролировать её толщину.
- getLineDash() – возвращает текущий массив тире штриховки, содержащий чётное число неотрицательных чисел.
- setLineDash(segments) – устанавливает текущий пунктир линии.
- lineDashOffset = value – указывает, где следует начинать тире массива в строке.



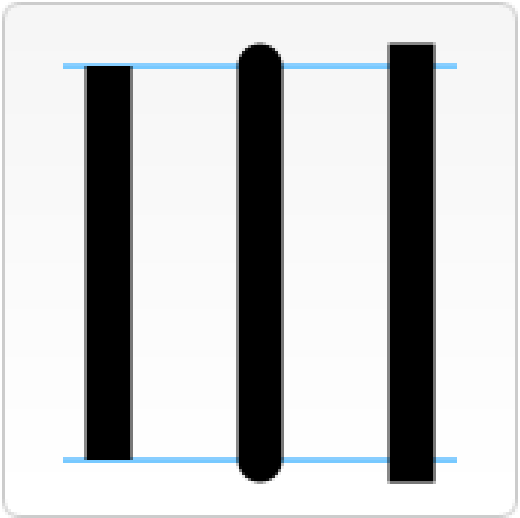
lineWidth

```
function draw() {
  let ctx = document.getElementById('canvas').getContext('2d');
  for (let i = 0; i < 10; i++){
     ctx.lineWidth = 1+i;
     ctx.beginPath();
     ctx.moveTo(5+i*14,5);
     ctx.lineTo(5+i*14,140);
     ctx.stroke();
  }
}
```



lineCap

- butt – концы линий соответствуют крайним точкам.
- round – концы линий округлены.
- square – концы линий описаны квадратом с равной шириной и половиной высоты толщины линии.

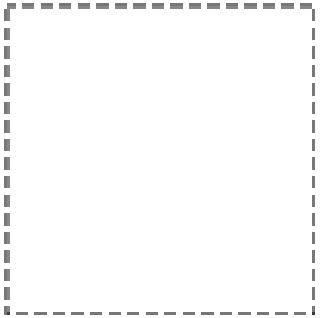


lineJoin

- round - радиус заполняемой части для скруглённых углов равен половине ширины линии, центр этого радиуса совпадает с концами соединяемых сегментов.
- bevel – заполняет дополнительную треугольную область между общей конечной точкой соединяемых сегментов и отдельными внешними прямоугольными углами каждого сегмента.
- miter – соединяемых сегменты соединяются путём расширения их внешних краёв для соединения в одной точке с эффектом заполнения дополнительной области в форме пастилки.
  - Эта настройка выполняется с помощью свойства miterLimit.

Использование штрихов

- Метод setLineDash и свойство lineDashOffset задают шаблон штрихов для линий.
- Метод setLineDash принимает список чисел, который определяет расстояния для попеременного рисования линии и разрыва, а свойство lineDashOffset устанавливает смещение, с которого начинается шаблон.



Градиенты

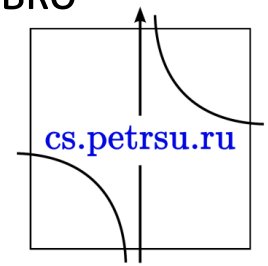
- createLinearGradient(x1, y1, x2, y2) - создает объект линейного градиента. Градиент может использоваться для заливки прямоугольников, окружностей, линий, текста и т.д.

- createRadialGradient(x1, y1, r1, x2, y2, r2) - создает объект радиального/кругового градиента.

- gradient.addColorStop(position, color) - определяет цвет и позицию остановки в объекте градиента.

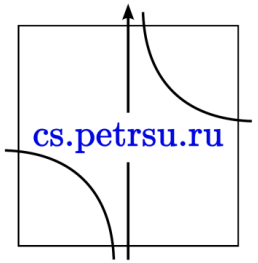
Шаблоны

- createPattern(image, type) - создаёт и возвращает новый canvas объект - шаблон (pattern).
  - image - CanvasImageSource (то есть HTMLImageElement, другой холст, элемент <video> или подобный объект).
  - type - строка, указывающая, как использовать image.
    - repeat - повторяет изображение в вертикальном и горизонтальном направлениях.
    - repeat-x - повторяет изображение по горизонтали, но не по вертикали.
    - repeat-y - повторяет изображение по вертикали, но не по горизонтали.
    - no-repeat - не повторяет изображение. Используется только один раз.



Тени

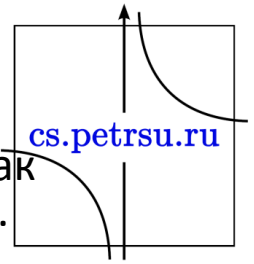
- shadowBlur – устанавливает/возвращает уровень размытости для теней
- shadowColor – устанавливает/возвращает цвет для теней
- shadowOffsetX – устанавливает/возвращает горизонтальное расстояние тени от фигуры
- shadowOffsetY – устанавливает/возвращает вертикальное расстояние тени от фигуры



```
function draw() {
  let ctx = document.getElementById('canvas').getContext('2d');
  ctx.shadowOffsetX = 2;
  ctx.shadowOffsetY = 2;
  ctx.shadowBlur = 2;
  ctx.shadowColor = "rgba(0, 0, 0, 0.5)";
  ctx.font = "20px Times New Roman";
  ctx.fillStyle = "Black";
  ctx.fillText("Sample String", 5, 30);
}
```

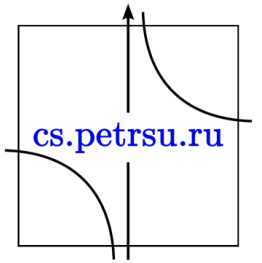
Трансформации

- rotate() – поворачивает текущий графический объект
  - Чтобы графический объект был повернут, метод rotate() необходимо применить до того, как объект будет нарисован.
- scale() – изменяет масштаб текущего графического объекта
  - Если изменяется масштаб какого-нибудь графического объекта, то будущие графические объекты также изменят масштаб. Позиционирование также будет масштабироваться.
- setTransform() – сбрасывает текущую матрицу трансформации в начальное состояние, а затем вызывает метод transform() с теми же параметрами
  - Трансформация будет применяться только на те графические объекты, которые будут нарисованы после вызова метода
- transform() – применяет заданную матрицу трансформации
  - Метод позволяет масштабировать, поворачивать, двигать и скручивать текущий контекст. Воздействует пропорционально на другие трансформации - rotate(), scale(), translate() или transform().
- translate() – ретранслирует позицию (0,0) в новое место
  - Если после метода translate() вызывается, например, такой метод как fillRect(), то его значение добавляется к значениям x- и y-координат.



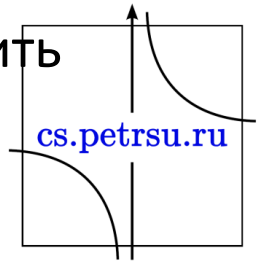
Текст

- font – устанавливает/возвращает свойства шрифта для текстового содержимого
- textAlign – устанавливает/возвращает выравнивание для текстового содержимого
- textBaseline – устанавливает/возвращает базовую линию, используемую при выводе текста
- fillText() – рисует текст с заливкой
- measureText() – возвращает объект, содержащий ширину заданного текста
- strokeText() – рисует текст без заливки



Основные шаги анимации

1. Очистить canvas
   - Если фигура не занимает всю площадь canvas, то всё что было нарисовано ранее необходимо стереть.
   - Проще всего это сделать при помощи метода clearRect().
2. Сохранить изначальное состояние canvas
   - Если изменяются любые настройки (такие как стили, трансформации и т.п.), которые затрагивают состояние canvas, то необходимо сохранить это состояние.
3. Нарисовать анимированные фигуры
   - Шаг на котором происходит отрисовка кадра.
4. Восстановить состояние canvas
   - Если сохранено первоначальное состояние, восстановить его, прежде чем отрисовывать новый кадр.



Управление анимацией

- Фигуры отрисовываются на canvas либо напрямую — при помощи методов canvas, либо с помощью сторонних функций.
- В нормальной ситуации результат станет виден на canvas после окончания выполнения скрипта.
  - К примеру, цикл for использовать для анимации нельзя.

- Это значит, нужен способ выполнения функций отрисовки через интервалы времени.
- Есть два способа для управления такой анимацией.

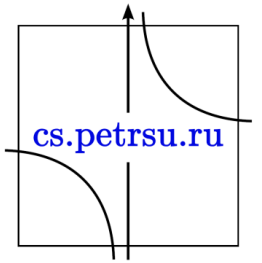
Два варианта анимации

- setInterval(function, delay)
  - Начинает периодически исполнять функцию function каждые delay миллисекунд.
- setTimeout(function, delay)
  - Запускает выполнение указанной функции function через delay миллисекунд.

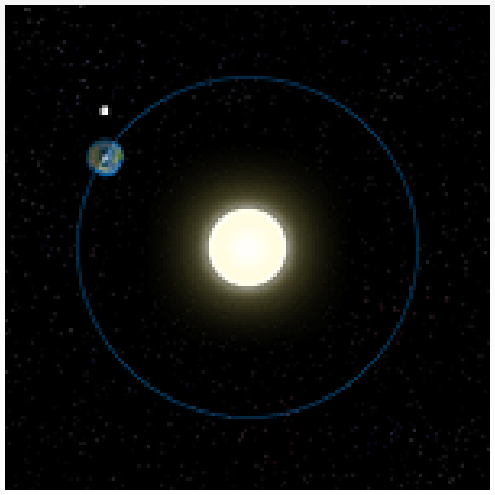
- requestAnimationFrame(callback)
  - Сообщает браузеру, что необходимо выполнить анимацию, и вызывает указанную функцию callback для обновления анимации перед следующей перерисовкой.

Что выбрать?

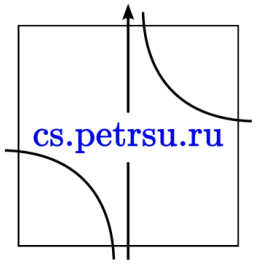
- Если не планируется взаимодействия с пользователем, можно использовать функцию setInterval() , которая многократно выполняет, предоставленный ей код.
- Иначе необходимо использовать setTimeout().
- Установив EventListener, можно перехватывать любые действия пользователя и запускать соответствующие функции анимации.
- Функция requestAnimationFrame является более эффективной для создания анимации, так как новая итерация вызывается, когда система готова к отрисовке нового кадра.
  - Количество вызовов в секунду примерно равно 60 и уменьшается, когда вкладка неактивна.



```
let sun = new Image();
function init(){
  sun.src = 'Canvas_sun.png';
  window.requestAnimationFrame(draw);
}

function draw() {
  let ctx = document.getElementById('canvas').getContext('2d');
  ctx.clearRect(0,0,300,300); // clear canvas
  ctx.fillStyle = 'rgba(0,0,0,0.4)';
  ctx.strokeStyle = 'rgba(0,153,255,0.4)';
  ctx.save();
  ctx.translate(150,150);

  // …

  ctx.drawImage(sun,0,0,300,300);
  window.requestAnimationFrame(draw);
}

init();
```

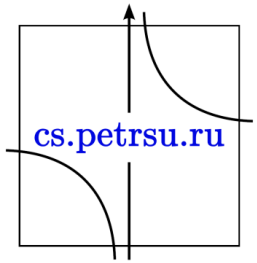


```
let pos_x = 50;
let pos_y = 50;

window.onload = function() {
  setTimeout("drawFrame()",200);
}

function drawFrame() {
  let ctx = document.getElementById('canvas').getContext('2d');
  ctx.clearRect(0,0,300,300);
  ctx.beginPath();
  ctx.rect(pos_x,pos_y,50,50);
  ctx.lineStyle = '#109bfc';
  ctx.lineWidth = 1;
  ctx.stroke();

  pos_x += 1;
  setTimeout("drawFrame()",200);
}
```



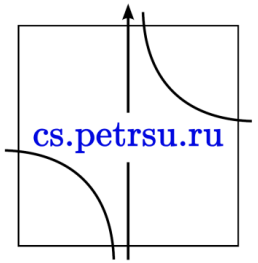
Доп.материал

JAVASCRIPT. JQUERY

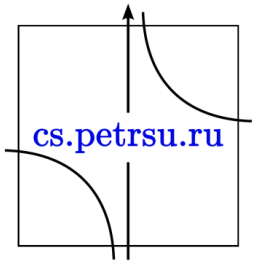


JQuery

- https://jquery.com/
- **jQuery** — библиотека JavaScript, содержащая в себе готовые функции языка JavaScript, все операции jQuery выполняются из кода JavaScript.
- Библиотека производит манипуляции с html-элементами, управляя их поведением и используя DOM для изменения структуры страницы.
  - При этом исходные файлы HTML и CSS не меняются, изменения вносятся лишь в отображение страницы для пользователя.

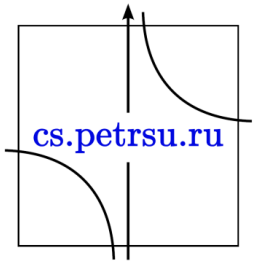


- Для выбора элементов используются селекторы CSS.
- Осуществляется с помощью функции $().
  - При вызове функция $() возвращает новый экземпляр объекта JQuery, который оборачивает ноль или более элементов DOM и позволяет взаимодействовать с ними различными способами.



- Выполнение различных сценариев возможно только после окончания загрузки структуры документа document, когда браузер преобразует html-код страницы в дерево DOM.
- Управление процессом загрузки обеспечивает конструкция:

```
jQuery(document).ready(function() {
...
});
```



Подключение

```
<script
src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>
```