JavaScript. Уровень 1
Основы веб-программирования

Валентин Гревцов

Темы курса

- Основы программирования
- Управляющие конструкции
- Функции
- Объектные типы
- Объектно-ориентированное программирование
- Дополнительная информация

Модуль 1

Основы программирования

Темы модуля

- Введение в JavaScript (ECMAScript-262)
- Обзор базовых типов
- Операторы
- Выражения и инструкции
- Переменные и константы
- Манипуляции с базовыми типами
- Тривиальные типы

Что мы изучаем на курсе?

- **ECMAScript-262**
  - Интерпретируемый
  - Регистрозависимый
  - ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
- **Диалекты**
  - JavaScript
  - JScript
  - ActionScript
- **Реализации**
  - Node.js
  - Windows Script Host
  - Adobe Flash
- **Редакции (версии)**
  - ES 5
  - ES 6 (ES 2015)
  - …

Инструменты

- Текстовый редактор
  - **Notepad++**, Sublime Text и т.д.

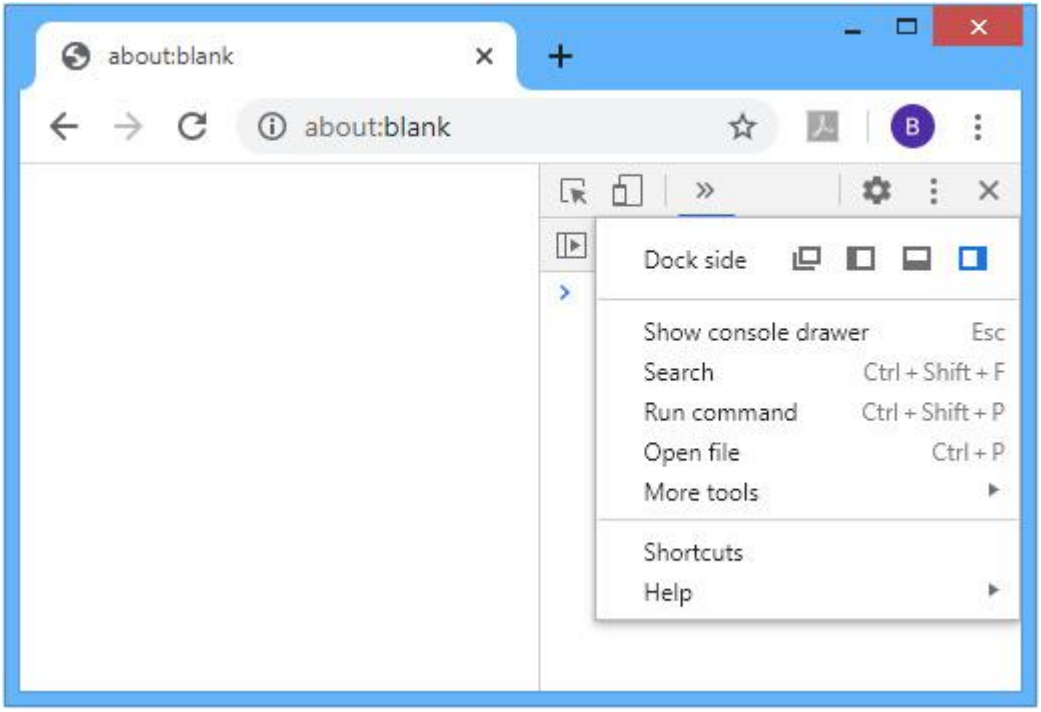
- Консоль JavaScript в браузерах
  - **Google Chrome (F12)**
  - Opera (Ctrl + Shift + I)
  - Mozilla Firefox (Ctrl + Shift + I) + addon FireBug
  - MS Internet Explorer (F12)

- Интерпретатор командной строки
  - **SpiderMonkey**, Rhino, V8

Консоль браузера Google Chrome

- Пустая вкладка браузера, в адресной строке введите **about:blank**

- Консоль вызывается клавишей **F12**



- Для очистки консоли нажмите **Ctrl** + **L**

Базовые типы: числа (Number)

- Целые положительные и отрицательные числа

```
128
1000000
-57
1_000_000  // 1000000 (ES 2015)
```

- Дробные числа

```
7.56
```

- Экспоненциальная форма записи (применяется не часто)

```
1.2e+2    // 120
1.2e2     // 120
1.2e-2    // 0.012
```

Операторы

- Операторы бинарные (два операнда) и унарные (один операнд)
  - 2 **+** 2 // 4
  - 2 **-** 2 // 0
  - 2 ***** 2 // 4
  - 2 **/** 2 // 1
  - 2 ****** 3 // 8

  - **%** (целочисленный остаток от деления)
    - 7 / 3 = 2.3333
      - но
    - 7 **%** 3 = **1**

      т.е. 7 = 3 * n + **1** // где n, сколько раз число 3 входит в число 7

- https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence
  - **( )**
    - 2 + 2 * 2 = 6
    - (2 + 2) * 2 = 8

Базовые типы: строки (String)

- Кавычки и апострофы
  - "Hello, world!" или 'Hello, world!'

- Спецсимволы
  - 'Hello, **\n\t**world!'
  - 'Hello "my" world!'
  - 'Hello ****'my****' world!'
  - 'format disk c:**** on computer'

- Конкатенация (склеивание строк)
  - 'Hello ' + 'my ' + 'world' + '!'

- В строках заданных через одинарные и двойные кавычки нельзя делать переносы на новую строку (разрыв строки)

- Если одинарные кавычки находятся внутри одинарных кавычек (это касается и двойных кавычек), то их надо экранировать обратным слешем. Если нужно использовать сам обратный слеш, то его то же необходимо экранировать

Лабораторная работа – 1.1

- Вывод текста на экран:
  - alert( );
    Привет, Alert!
  - document.write( );
    Привет, document.write
    <h2 style='color: red;'>Привет, Мир!</h2>
  - console.log( );
    Мое первое сообщение в консоли

Базовые типы: булев (Boolean)

- Константы **true** и **false**

```
3 < 4          // true
3 > 4          // false
5e2 == 500  // true
```

- Другие операторы сравнения

  ===

  <= и >=

  != и !==

- Сравнивание строк и чисел (**win** + **r** (выполнить) -> **charmap**)

```
'a' > 'A';              // true
'z' > 'A';              // true
'A' > '1';              // true
'10' > 'n';             // false
'John' == 'John';  // true
'John' == 'john';   // false
```

Логические операторы

- При проверке значения *операндов* преобразуются *в логические*

- https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence
  - ! [логическое NOT (НЕ)]
  - && [логическое AND (И)]
  - || [логическое OR (ИЛИ)]

    - console.log(true && true); // true
    - console.log(true && false); // false

    - console.log(true || false); // true
    - console.log(!true); // false

- Число **0** и **пустая строка** преобразуются в **false**
  - 3 && 4 // 4
  - 0 && 3 // 0
  - 'Hello' && 2 || '' && 5 // 2

Разминка

- Не запуская код вычислите, что вернет следующая конструкция?

- ((5 >= 7) || ('JavaScript' != 'Java')) && !(((11 * 3) == 99) && true)

Выражения и инструкции

- https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements

- Expression (выражение)
  Выражением является любой корректный блок кода, который возвращает значение
  - 5 + 10
  - 'John'
  - !false

- Statement (инструкция)
  Инструкция – это (грубо говоря) команда, действие
  - 5 + 10;
  - 'John';
  - !false;

Переменные

- Объявление переменной
  - `var user;`

- После объявления, можно записать в переменную данные
  - ```
    var user;
    user = 'John';
    ```

- Можно совместить объявление переменной и запись данных
  - `var user = 'John', pass = 123;`

- Использование переменной
  - ```
    var num = 25;
    console.log( num + 10 );      // 35, значение переменной не меняется
    console.log( num );           // 25
    ```
  - `num = num + 10;` // 35
    - или с использованием оператора «прибавить и присвоить»
  - `num += 10;` // 35

Зарезервированные слова

- Лексический синтаксис
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Lexical_grammar

- Зарезервированные ключевые слова в ECMAScript 2015:

  break, case, class, catch, const, continue, debugger, default, delete, do, else, export, extends, finally, for, function, if, import, in, instanceof, let, new, return, super, switch, this, throw, try, typeof, var, void, while, with, yield

Переменные и константы (ES-2015)

- В ES-2015 предусмотрены новые способы объявления переменных через **let** и **const** вместо **var**

- Объявление переменной через **let**
  В отличие от ключевого слова var, переменную объявленную с помощью **let** нельзя редекларировать (объявить повторно)
  - ```
    let a = 5;
    a = 10;
    let a = 15;     // SyntaxError: Identifier 'a' has already been declared
    ```

- Объявление **const** задает константу, т.е. переменную, которую нельзя изменить. Имя константы принято задавать в верхнем регистре
  - ```
    const NUM = 5;
    ```

- Константу нельзя объявить отдельно, а потом присвоить ей значение. Это нужно делать за один раз
  - ```
    const NUM; // Error: Missing initializer in const declaration
    NUM = 5;
    ```
    ```

Манипуляции с типами

- Проблемы:

```
2 + '2'     // '22'
2 + true      // 3
'' + true     // 'true'
'x' * 10      // NaN! Что это?
```

- Оператор определения типа

```
var x = 5, y = 0, z = 'text';
console.log( typeof x );     // 'number'
console.log( typeof( z ) );  // 'string'
```

- Ф-ция **isNaN( );** определяет является ли литерал или переменная нечисловым значением (NaN) или нет:

```
console.log( isNaN( x ) );          // false, т.к. аргумент является числом
console.log( isNaN( x * z ) );      // true, т.к. аргумент является не числом
```

Привидение типов

- В число:

```
var x = '5' * 1;
x = +'5';

x = parseInt( '123.456' );        // 123
x = parseInt( 's123.456' );       // NaN

x = parseFloat( '123.456' );      // 123.456

x = Number( '123.456' );          // 123.456
```

- В строку:

```
x = 5 + '';
x = String( 5 );
```

- В логический тип:

```
x = !!5;          // true
x = Boolean( 5 );
```

Тривиальные типы

- Тип **undefined** имеет переменная, которой не было присвоено значение
  ```
  var x;
  console.log( typeof x ); // undefined
  ```

- Значение **null** является литералом JavaScript, представляющим нулевое или «пустое» значение, то есть, когда нет никакого объектного значения.
  Это **одно из примитивных значений JavaScript**

  Значение null является литералом (а не свойством глобального объекта, как undefined). В API **null часто присутствует в местах, где ожидается объект, но подходящего объекта нет**. При проверке на null или undefined помните о различии между операторами равенства (==) и идентичности (===) (с первым выполняется преобразование типов)
  ```
  var age = null;
  ```

- ```
  null === undefined;      // false
  null == undefined;       // true
  ```
  ```

Запускаем файл в браузере

- <!-- подключаем внешний JavaScript-файл -->
  <script src='lib.js'></script>

- <!-- пишем свой код, можем обращаться к коду из внешнего файла -->
  <script>
  // однострочный комментарий

  /*
  многострочный
  комментарий
  */

  var name = 'John';
  console.log('Hello, ' + name);
  </script>

- В одном теге SCRIPT нельзя одновременно подключить внешний скрипт и разместить код

Шаблонные строки (ES-2015)

- Шаблонные строки (шаблоны) являются строковыми литералами, допускающими **использование выражений** и **переносов**

- Они заключены в обратные кавычки (` `) вместо двойных или одинарных

- Для вставки выражений в строки использовали следующий синтаксис:

```
var a = 5, b = 10;
console.log( "Fifteen is " + ( a + b ) + "." );          // Fifteen is 15.
```

- Сейчас, при помощи шаблонов, вы можете использовать **синтаксический сахар** для повышения читаемости кода:

```
var a = 5, b = 10;
console.log( `Fifteen is ${ a + b }.` );                 // Fifteen is 15.
```

- В шаблонных строках допускаются переносы

```
var txt = `Hello,
    world`;
console.log( txt );
```

Выводы

- Введение в JavaScript (ECMAScript-262)
- Обзор базовых типов
- Операторы
- Выражения и инструкции
- Переменные и константы
- Манипуляции с базовыми типами
- Тривиальные типы

Модуль 2

Управляющие конструкции

Темы модуля

- Цикл while
- Операторы инкремента и декремента
- Цикл for
- Цикл do while
- Управляющие конструкции if – else if – else
- Прерывание и продолжение цикла
- Управляющая конструкция switch

Цикл (loop)

- Цикл – это блок кода, позволяющий повторять выполнение какого-нибудь раздела кода несколько раз, возможно изменяя значение некоторых переменных при каждом выполнении кода (итерации). За счет этого, как правило, удается сократить некоторые сценарии до нескольких строк кода. Это не только экономит время, но и позволяет избавиться от опечаток в повторяющихся строках

- Для любого цикла всегда выполняются следующие действия:
  - инициализация "счетчика";
  - проверка условия;
  - выполнение инструкций;
  - изменение "счетчика" (как правило увеличение на единицу)

Цикл while

- ```
  while( условие ) {
      инструкция 1;
      инструкция 2;
      …;
  }
  ```

- ```
  var x = 0;                          // Часть A – создание счетчика

  while( x < 5 ) {                    // Часть B – проверка условия выхода
      console.log( 'x = ' + x );      // Тело цикла
      x += 1;                         // Часть C – изменение счетчика
  }

  console.log( 'Код после цикла' );
  ```

- Фактически цикл "бегает" по треугольнику: **Часть B** – **тело цикла** – **Часть C**, пока в **Части B** не получим **false**. И только после этого выполнится **Код после цикла**

Операторы ++ (инкремент) и -- (декремент)

- Позволяют изменить значение переменной на единицу

- ```
  var i = 1;
  i++;
  console.log( i );   // 2
  ```

- POST
  - ```
    var i = 1;
    var x = i++;   // x == 1, i == 2
    ```
    - или
  - ```
    var x = i; i += 1;
    ```

- PRE
  - ```
    var i = 1;
    var x = ++i;   // x == 2, i == 2
    ```
    - или
  - ```
    i += 1; var x = i;
    ```
    ```

Лабораторная работа – 2.1

- Напишите программу, которая в цикле с помощью метода document.write( ) выводит на экран все 6 заголовков HTML

- document.write('<h1>Заголовок H1</h1>');
  document.write('<h2>Заголовок H2</h2>');
  document.write('<h3>Заголовок H3</h3>');
  document.write('<h4>Заголовок H4</h4>');
  document.write('<h5>Заголовок H5</h5>');
  document.write('<h6>Заголовок H6</h6>');

Лабораторная работа – 2.2

- Нарисуйте треугольник с помощью цикла

- *
  **
  ***
  ****
  *****
  ******
  ****

Цикл for

- for(выражения (Часть A); условие (Часть B); выражения (Часть C))
  ```
  // если инструкция одна, то фигурные скобки можно не использовать
  инструкция;
  ```

- for(выражения (Часть A); условие (Часть B); выражения (Часть C)) {
  ```
  инструкция 1;
  инструкция 2;
  …;
  }
  ```

- ```
  for( var x = 0; x < 5; x++ ) {
      console.log( 'x = ' + x );
  }

  console.log( 'Код после цикла' );
  ```

- **Часть A** выполняются только один раз, **части B** и **C** выполняются при каждой итерации (повторении действия) цикла
  ```

Цикл do – while

- ```
  do {
      инструкция 1;
      инструкция 2;
      …;
  } while ( условие );
  ```

- ```
  var x = 10;

  do {
      console.log( 'x = ' + x );
      x++;
  } while ( x < 5 );

  console.log( 'Код после цикла' );
  ```

- Цикл do – while выполнится минимум один раз, даже если в **Часть В** сразу придет false

Условные операторы if – else if – else

- ```
  if ( условие 1 ) {
      инструкция 1;
      инструкция 2;
  } else if ( условие 2 ) {
      инструкция 3;
      инструкция 4;
  } else {
      инструкция 5;
      инструкция 6;
  }
  ```

- ```
  var check = 1500;

  if ( check >= 1000 ) {
      console.log( 'Скидка 5%' );
  }

  console.log( 'Приходите еще' );
  ```

- 


- ```
  var check = 500;

  if ( check >= 1000 ) {
      console.log( 'Скидка 5%' );
  } else {
      console.log( 'Вам календарик' );
  }

  console.log( 'Приходите еще' );
  ```
  ```

Различия между var и let

- Приведение к булеву типу: любое число отличное от 0 преобразуется в true

- ```
  if ( 1 ) {
      var a = 10;
      console.log( a );   // 10
  }
  console.log( a );       // 10
  ```

- ```
  if ( 1 ) {
      let b = 10;
      console.log( b );   // 10
  }
  console.log( b );       // Uncaught ReferenceError: b is not defined
  ```

- Переменная, объявленная через **var**, видна везде

- Переменная, объявленная через **let**, видна только в рамках блока {…}, в котором объявлена

Лабораторная работа – 2.3

- Написать цикл, выводящий цифры от 10 до 0

- Отобразить только нечетные цифры и подписать их:
  9 нечетное
  7 нечетное
  5 нечетное
  3 нечетное
  1 нечетное

Тернарный оператор ? :

- https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

- ```
  var name = 'John', txt = '';

  if ( name == 'John' ) txt = 'Привет, Джон';
  else txt = 'Привет, незнакомец';

  alert( txt );
  ```

- условие **?** значение1 **:** значение2;
  Если условие верно – вернется **значение1**, если неверно – **значение2**

  ```
  var name = 'John';

  var txt = ( name == 'John' ) ? 'Привет, Джон' : 'Привет, незнакомец';

  alert( txt );    // 'Привет, Джон'
  ```
  ```

Лабораторная работа – 2.4

- Создайте переменную **age** и присвойте ей целочисленное значение

- Используя конструкцию **if** – **else if** – **else**, напишите программу, которая проверяет значение в переменной **age** и:

  - если значение попадает в диапазон 1 – 17 (вкл.)
    - выводит 'Вам работать еще рано – учитесь'

  - если значение попадает в диапазон 18 – 59 (вкл.)
    - выводит 'Вам еще работать и работать'

  - если значение больше 59
    - выводит 'Вам пора на пенсию'

Лабораторная работа – 2.5

- Считаем ворон на ветке

- Создайте переменную **crow** и присвойте ей целочисленное значение

- Скрипт должен вернуть количество ворон на ветке с правильным окончанием. Например:
  - На ветке сидит **2** ворон**ы**
  - На ветке сидит **11** ворон

Прерывание и продолжение

- for(выражения; условие; выражения) {
  инструкция;
  if (условие 1)
  break; // прекращает работу цикла
  инструкция;
  if (условие 2)
  continue; // прекращает только текущую итерацию
  инструкция;
  }

- for(var x = 0; x < 10; x++) {
  if (x == 5) {
  console.log('Завершаем цикл');
  break;
  } else {
  console.log('x = ' + x);
  }
  }

Вложенные циклы

- Используются, например, для создания таблиц (внешний цикл создает строки, вложенный цикл создает и наполняет ячейки)

-
```
for( var i = 0; i < 3; i++ ) {

    console.log( ( i + 1 ) + ". Начало внешнего цикла" );

    // вложенный цикл начало
    for( var j = 0; j < 3; j++ ) {
        console.log( "\t Я вложенный цикл" );
    }
    // вложенный цикл завершение

    console.log( "Завершение внешнего цикла" );

} // внешний цикл

console.log( 'Код после циклов' );
```

Метки

- Инструкция метки (label) используется вместе с break или continue для альтернативного выхода из цикла. Она добавляется перед блочным выражением в качестве ссылки, которая может быть использована в дальнейшем

-
```
outer_loop: for( let i = 0; i<10; i++ ) {
    console.log( 'Внешний цикл' );

    inner_loop: for( let i = 0; i<10; i++ ) {
        console.log( '\tВложенный цикл' );

        if ( i == 5 ) {
            console.log( 'Завершаем оба цикла' );
            // break inner_loop;
            break outer_loop;
        }
    } // вложенный цикл

}
```

switch (переключатель)

- Конструкция switch заменяет собой сразу несколько if

- switch (выражение) {
    case условие 1:
        инструкции;
        break;
    case условие 2:
        инструкции;
        break;
    case условие n:
        инструкции;
        break;
    default: инструкции;
  }

- var num = 10;

  switch (true) {
      case (num < 10):
          console.log('num меньше 10');
          break;
      case (num == 10):
          console.log('num равен 10');
          break;
      case (num > 10):
          console.log('num больше 10');
          break;
      default:
          console.log('Странный num');
  }

Выводы

- Цикл while
- Операторы инкремента и декремента
- Цикл for
- Цикл do while
- Управляющие конструкции if – else if – else
- Прерывание и продолжение цикла
- Управляющая конструкция switch

Модуль 3

Функции

Темы модуля

- Понятие функций
- Декларация функций
- Аргументы функции
- Возврат значений
- Области видимости
- Функция-выражение
- Анонимная функция

Функции

- Ф-ция – это небольшой сценарий внутри сценария. Она предназначена для выполнения отдельной задачи или ряда задач
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Functions

- Встроенные функции:
  ```
  alert( 'Hello world!' );
  var x = parseInt( '123.456' );:
  ```

- Пользовательские функции
  ```
  function sayHello( ) {
      console.log( 'Hello world!' );
  }

  console.log( 'А я текст после функции' );

  sayHello( ); // вызов ф-ции
  ```

Параметры

- При вызове функции ей можно передать данные, которые та использует по своему усмотрению. Если передается несколько параметров, то они разделяются запятыми

- ```
  function sayHello( name ) {
      console.log( 'Hello, ' + name + '!' );
  }

  sayHello( 'John' ); // Hello, John!
  sayHello( 'Mike' ); // Hello, Mike!
  ```
  ```

Параметры по умолчанию

- Можно указывать параметры по умолчанию через равенство **=**

- Параметр по умолчанию используется при **отсутствующем аргументе** или равном **undefined**

- При передаче любого значения, кроме undefined, включая пустую строку, ноль или null, параметр считается переданным, и значение по умолчанию не используется

- ```
  function user( name = "Guest", age = 18, admin = false ) {
      console.log( name + '; ' + age + '; ' + admin );
  }

  user( undefined, 25 ); // Guest; 25; false
  ```
  ```

Оператор spread

- Чтобы получить массив аргументов, можно использовать оператор **...** https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Spread_syntax

- Тут в **arr** попадёт массив (массивы будут рассмотрены позже) всех аргументов, начиная с третьего. Нумерация элементов массива начинаться с 0

- ```
  function user( name, age, ...arr ) {

      console.log( name + '; ' + age ); // John; 25
      console.log( arr ); // [ 'admin', 'Moscow', 'john@jmail.com' ]
      console.log( arr[ 0 ] ); // admin

  }

  user( 'John', 25, 'admin', 'Moscow', 'john@jmail.com' );
  ```
  ```

Возврат значений

- Оператор **return** завершает выполнение текущей функции и возвращает значение из этой функции в место её вызова

```
function sum( x, y ) {
    return x + y;
    console.log( 'А я текст, который не появится' );
}

// в переменную result попадет результат работы функции
var result = sum( 10, 20 );
console.log( result );     // 30
```

- Функция всегда что-то возвращает. Если **return** вызван без значения, или функция завершилась без **return**, то её результат будет равен **undefined**

```
function foo( ) {
}
console.log( foo( ) );     // undefined
```

Пример использования return (забегая вперед…)

```
// имеем несколько переменных, у которых разный регистр и пробелы по бокам
var pole1 = '   one', pole2 = 'Two   ', pole3 = '   THREE   ';

function clearStr( data ) {
      // фильтруем данные (убираем пробелы и приводим к нижнему регистру)
      // используем цепочку вызовов
      return data.trim( ).toLowerCase( );
}

// присваиваем отфильтрованные данные переменным
pole1 = clearStr( pole1 );
pole2 = clearStr( pole2 );
pole3 = clearStr( pole3 );

// печатаем данные. Св-во length – определяет длину строки
console.log( pole1, pole1.length );   // one 3
console.log( pole2, pole2.length );   // two 3
console.log( pole3, pole3.length );   // three 5
```

Лабораторная работа – 3.1

- Создать функцию-оболочку для console.log( );

Конструктор Function

- Конструктор Function создаёт новый объект Function. В конструктор Function параметры передаются в виде строки. Все параметры необязательные кроме последнего, в который передается тело функции

- Главное отличие от других способов объявления функции, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения

```
var foo = new Function( 'a', 'b', 'return a + b' );
console.log( foo( 2, 6 ) ); // 8

// или

function foo( a, b ) {
    return a + b;
}
console.log( foo( 2, 6 ) ); // 8
```

Лабораторная работа – 3.2

- Используем код Лабораторной работы – 2.2

- Описать функцию **drawLine( )**, которая рисует треугольник

- Функция принимает следующие параметры:
  - **lines** – количество строк;
  - **symbol** – сам символ, например *

Области видимости – 1

- Если переменная отсутствует внутри тела функции, в этом случае функция будет обращаться к внешней переменной. При этом, функция получает текущее значение внешней переменной, т.е., ее последнее значение. Старое значение переменной нигде не сохраняется

- ```
  var first_name = 'Вася', last_name = 'Петров';

  function showUser( city ) {
      var last_name = 'Иванов';
      var message = 'Привет, я ' + first_name + ' ' + last_name + ', ' + city;
      return message;
  }

  first_name = 'Сережа';

  console.log( showUser( 'Москва' ) ); // Привет, я Сережа Иванов, Москва
  ```
  ```

Области видимости – 2

- ```
  var x = 'x-global', y = 'y-global';

  function foo( ) {
      x = 'x-local'; // нет ключевого слова var
      var y = 'y-local';
      var z = 'z-local';

      console.log( 'Внутри ф-ции foo переменная x = ' + x ); // x-local
      console.log( 'Внутри ф-ции foo переменная y = ' + y ); // y-local
      console.log( 'Внутри ф-ции foo переменная z = ' + z ); // z-local
  }

  foo( );

  console.log( 'Снаружи ф-ции foo переменная x = ' + x ); // x-local
  console.log( 'Снаружи ф-ции foo переменная y = ' + y ); // y-global
  console.log( 'Снаружи ф-ции foo переменная z = ' + z ); // Error: z is not defined
  ```
  ```

Функция как выражение

- ```
  function sayHi( name ) {
      console.log( "Привет, " + name );
  }

  sayHi( 'John' ); //  выполнит ф-цию (есть круглые скобки после имени ф-ции)

  console.log( sayHi ); // выведет код ф-ции, т.к. это ссылка на ф-цию

  var f = sayHi; // переменной присвоили код ф-ции

  f( 'John' ); // выполнит ф-цию
  ```

- Пример использования (изменение цвета фона страницы):

  ```
  function bg_color( ) {
      document.body.style.backgroundColor = '#fcc';
  }
  onclick = bg_color;
  ```
  ```

Анонимная функция

- Существует ещё один синтаксис создания функций, который называется **Function Expression** (Функциональное Выражение). При этом функция создаётся и явно присваивается переменной, как любое другое значение
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/function

```
const SUM = function( val ) {
    return val + val;
};  // тут точка с запятой обязательна, т.к. идет присваивание

console.log( SUM( 2 ) );  // 4
```

- Самовызывающаяся анонимная функция или IIFE (Immediately Invoked Function Expression) это JavaScript функция, которая выполняется сразу же после того, как она была определена

```
;( function( ) {
    console.log( 'Hello world!' );
} )( );
```

Разъяснение

- ```
  ;( function( ) {
      console.log( 'Hello world!' );
  } )( );
  ```

- Это выражение известно как **Self-Executing Anonymous Function**. Оно состоит из двух частей:
  - **первая** – сама анонимная функция с лексической областью видимости, заключённая внутри **Оператора группировки ( )**. Благодаря этому переменные **IIFE** замыкаются в ее пределах, и глобальная область видимости ими не засоряется;
  - **вторая** часть создаёт мгновенно выполняющееся функциональное выражение **( )**, благодаря которому JavaScript выполняет функцию

- В примере видна **;** перед анонимной функцией. Дело в том, что существующий механизм автоподстановки точек с запятой Automatic Semicolon Insertion (ASI) срабатывает лишь в определённых случаях. Однако строка, начинающаяся с **(** не входит в перечень этих случаев. Поэтому разработчики добавляют **;** в тех случаях, когда код может быть скопирован и добавлен в другой
  ```

И еще о функциях

- У нас имеются следующие способы объявления функции:

- Конструктор Function

- **Function Declaration** (создаются интерпретатором до выполнения кода)

```
sayHi( 'Вася' );    // функция исполнится, но так делать плохо

function sayHi( name ) {
    console.log( 'Привет, ' + name );
}
```

- **Function Expression**

```
sayHi( 'Вася' );    // Error: sayHi is not a function, т.к. сейчас это переменная

var sayHi = function( name ) {
    console.log( 'Привет, ' + name );
};
```

Функции через => (стрелочные функции)

- https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions

```
var sum1 = function( a ) {
    return a + 1;
};
console.log( sum1( 2 ) );    // 3
```

или

```
var sum2 = a => a + 1;
console.log( sum2( 2 ) );    // 3
```

Как видно, **a => a + 1** – это уже готовая функция. Слева от **=>** находится аргумент, а справа – выражение, которое нужно вернуть

- Если аргументов несколько, то нужно обернуть их в скобки

```
var sum3 = ( a, b ) => a + b;
console.log( sum3( 1, 2 ) ); // 3
```

- Если нет аргументов, указываются пустые круглые скобки

```
var sayHi = ( ) => console.log( 'Hello' );    // Hello
sayHi( );
```

Многострочные стрелочные функции

- Часто нам нужно выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать **return** внутри них, как в обычной ф-ции

- ```
  let sum = ( a, b ) => {  // фигурная скобка, открывающая тело ф-ции
      let result = a + b;
      // при фигурных скобках для возврата значения нужно явно вызвать return
      return result;
  };

  console.log( sum( 1, 2 ) );   // 3
  ```
  ```

Замыкание

- Замыкание – это комбинация функции и лексического окружения, в котором эта функция была определена. Другими словами, замыкание даёт вам доступ к области внешней функции из внутренней функции. В JavaScript замыкания создаются каждый раз при создании функции, во время её создания https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures

-
```
function outer( arg1, arg2 ) {

    function inner( ) {
        return arg1 + arg2;
    }

    return inner( );

}

var result = outer( 2, 4 );
console.log( result ); // 6
```

Передача аргументов во вложенную функцию

- Для этого потребуется, чтобы внешняя функция возвращала ссылку на вложенную функцию, т.е. имя функции без круглых скобок

- ```
  function outer( x ) {
      function inner( y ) {
          return x + y;
      }
      // ссылка на вложенную функцию
      return inner;
  }

  // передаем значение в переменную x
  var add5 = outer( 5 );
  var add10 = outer( 10 );

  // передаем значение в переменную y
  console.log( add5( 2 ) );    // 7
  console.log( add10( 2 ) );   // 12
  ```
  ```

Лабораторная работа – 3.3

- Напишите функцию **compare( )**, которая принимает целочисленное значение **x** и возвращает анонимную функцию

- Анонимная функция должна принимать целочисленное значение **y** и возвращать результат сравнения значений **y** и **x** как:
  - если **y** больше, чем **x**, то возвращается **true**;
  - если **y** меньше, чем **x**, то возвращается **false**;
  - если значения равны, то возвращается **null**

Лабораторная работа – 3.4

- Напишите функцию **showTables( )**, которая рисует таблицу вида:
  dog dog dog cat cat dog
  dog dog cat cat dog dog
  …

- Последовательность: три собаки, два кота. Число зверей в строке: 6. Число строк произвольное. Разделителем является символ табуляции **\t**

- Вывод осуществить в консоль

Рекурсия

- Рекурсивная функция, это функция, которая вызывает саму себя. Используется, например, при создании анимации.
  Не используйте рекурсию там, где можно применить цикл

- ```
  function foo( count ) {
      count++;

      if ( count < 5 ) {
          console.log( "count = " + count );
          // ф-ция вызывает саму себя
          foo( count );
      } else {
          return;
      }

  }

  foo( 0 );
  ```
  ```

Лабораторная работа – 3.5

- Напишите функцию **printNumbers( )**, которая принимает два целочисленных значения:
  num – конечное число;
  cols – количество колонок

- Функция выводит числа от 0 до **num** в следующем порядке:
  0 4 8
  1 5 9
  2 6 10
  3 7 11

- Вывод осуществить в консоль

Функция обратного вызова

- Функция обратного вызова (callback) – это функция, переданная в другую функцию в качестве аргумента, которая затем вызывается по завершению какого-либо действия

- ```
  function greeting( name ) {
      console.log( 'Hello, ' + name );
  }

  function processUserInput( callback ) {
      var name = prompt( 'Please enter your name' ); // модальное окно для ввода

      // если пользователь ничего не ввел – завершаем работу функции
      if ( name == null || name == '' )  return;

      callback( name );
  }

  processUserInput( greeting );
  ```
  ```

Выводы

- Понятие функций
- Декларация функций
- Аргументы функции
- Возврат значений
- Области видимости
- Функция-выражение
- Анонимная функция

Модуль 4

Объектные типы

Темы модуля

- Объектный тип: Объект (Object)
- Свойства объекта
- Методы объекта
- Объектный тип: Массив (Array)
- Свойства и методы функций
- Встроенный объект Math

Объектный тип: Объект (Object)

- Создаем пустой объект
  ```
  var user = { };

  if ( user ) { } // тут в if всегда будет true, даже если объект пустой
  ```

- Добавляем объекту свойства
  ```
  user.name = 'John';
  user.age = 25;
  user.passHash = 'gsfd8hfgfn9ngs';
  user.admin = true;

  console.log( user.name ); // 'John'
  ```

- Объект представляет собой неупорядоченный набор пар – **ключ**: **значение**. Ключами могут быть только строки (тип String). Следовательно, ключи можно не брать в ' '. Значениями может быть любой тип

Свойства объекта

- В момент создания объекта ему можно добавить свойства

```
var user = {
    name: 'John',
    age: 25,
    passHash: 'gsfd8hfgfn9ngs',
    admin: true
};
```

- Свойства объекта не являются константой и их можно изменить

```
user.name = 'Guest';
user.age = 18;

console.log( user.name ); // 'Guest'
```

Короткий синтаксис (ES-2015)

- Если имеются переменные с одинаковым именем, что и у предполагаемых имен свойств объекта в одной области видимости, можно опустить двоеточие и значение при создании литерала объекта.

  Литералы используются для представления значений в JavaScript. Они являются фиксированными значениями, а не переменными
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Grammar_and_types

- ```
  var name = 'John', age = 25, passHash = 'gsfd8hfgfn9ngs', admin = true;

  var user = {
      name,
      age,
      passHash,
      admin,
  };

  console.log( user.name, user.age ); // John 25
  ```
  ```

Property accessors

- В имени переменной нельзя использовать пробел. И именем переменной не может быть зарезервированное слово. Однако, в имени свойства объекта их можно использовать.
  В этом случае имя свойства объекта заключается в ' ', а обращение к свойству происходит через **[ ]**. Хотя, через **[ ]** можно обратиться к любому свойству

- ```
  var user = {
      'user name': 'Guest',
      age: 18,
      'var': false
  };

  user[ 'user name' ] = 'John';

  console.log( user[ 'user name' ] );  // 'John'
  console.log( user.age );             // 18
  console.log( user[ 'age' ] );        // 18
  console.log( user[ 'var' ] );        // false
  ```
  ```

Обращение к свойствам объекта

- ```
  var user = {
      name: 'Guest',
      2: 0
  };

  console.log( user[ '2' ] );        // 0
  console.log( user[ 1+1 ] );        // 0

  var x = 'name';
  console.log( user[ x ] );          // 'Guest'
  ```
  ```

Проверка наличия свойств

- Оператор **in** возвращает **true**, если свойство (ключ) содержится в указанном объекте. Иначе вернется **false**
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/in

- ```
  var user = {
      x: 25
  };

  console.log( 'x' in user ); // true
  console.log( 'y' in user ); // false

  // оператор delete удаляет свойство из объекта
  delete user.x;
  console.log( 'x' in user ); // false
  ```
  ```

Перебор свойств объекта

- Если именами св-в объекта (ключами) являются числа идущие по порядку, подойдет цикл **for**

-
```
var obj1 = {
    0: 'Guest',
    1: 25,
    2: false,
};

for ( var i = 0; i in obj1; i++ )
    //            ключ  :  значение
    console.log( i + ' : ' + obj1[ i ] );
```

-
```
var obj2 = {
    name: 'Guest',
    age: 25,
    admin: false
};

for ( var i in obj2 )
    //            ключ  :  значение
    console.log( i + ' : ' + obj2[ i ] );
```

- Оператор **for…in** проходит через перечисляемые свойства объекта. Он пройдёт по каждому отдельному элементу

Лабораторная работа – 4.1

- Создайте пустой объект **book1**
- Добавьте объекту свойства: **title**, **pubYear** и **price** с произвольными значениями
- Создайте объект **book2** и установите свойства **title**, **pubYear** и **price** с произвольными значениями
- Выведите свойства объектов в цикле

Метод Object.freeze( )

- Замораживает объект. Это значит, что он предотвращает добавление новых свойств к объекту, удаление старых свойств из объекта и изменение существующих свойств. **В строгом режиме** такие попытки вызовут выброс исключения TypeError

- 'use strict';

```
var user = {
    name: 'John',
    age: 25,
    passHash: 'gsfd8hfgfn9ngs',
    admin: true,
};

Object.freeze( user );

user.name = 'Вася'; // Error: Cannot assign to read only property 'name' of object…
```

Метод Object.defineProperty( )

- Определяет новое или изменяет существующее свойство непосредственно на объекте, возвращая этот объект

- ```
  var user = {
      name: 'John',
      age: 25,
      passHash: 'gsfd8hfgfn9ngs', // плохо, если кто-то может это увидеть
      admin: true
  };

  Object.defineProperty( user, 'passHash', {
      enumerable: false,      // возможность видеть свойство в цикле
      writable: true,         // возможность перезаписать значение свойства
      configurable: false     // возможность удалять/переназначать свойство
  } );

  for ( var i in user )
      console.log( i + ' : ' + user[ i ] ); // покажет св-ва name, age и admin
  ```
  ```

Объект JSON

- JSON является синтаксисом для сериализации (преобразования сложной структуры в простую и обратно) объектов и массивов. Содержит два метода:
  JSON.stringify( ); возвращает строку JSON;
  JSON.parse( ); разбирает строку JSON

- ```
  var user = { name: 'John', age: 25 }, a, b;

  // объект выводится в строку
  a = JSON.stringify( user );
  console.log( a, typeof a ); // { "name":"John","age":25 } string

  // добавление параметра space, каждое св-во объекта на своей строке
  a = JSON.stringify( user, null, ' ' );
  console.log( a, typeof a );

  b = JSON.parse( a );
  console.log( b, typeof b ); // { name: "John", age: 25 } "object"
  ```
  ```

Сравнение и передача значений

- У переменных

- ```
  var x1 = 10;
  var x2 = x1;
  var x3 = 10;
  ```

- ```
  x1 == x2;        // true
  x1 == x3;        // true
  ```

- ```
  x1 = 20;
  ```
- ```
  console.log( x2 ); // 10
  ```

- У объектов

- ```
  var obj1 = { x: 10 };
  var obj2 = obj1;
  var obj3 = { x: 10 };
  ```

- ```
  obj1 == obj2;        // true
  obj1 == obj3;        // false
  obj1.x == obj3.x;    // true
  ```

- ```
  obj1.x = 20;
  ```
- ```
  console.log( obj2.x );  // 20
  ```

- Один объект будет равен другому объекту только в одном случае – если это один и тот же объект. В каком случае Вася будет равен Васе?

Методы объекта

- Метод – это функция, ассоциированная с объектом или, проще говоря, метод – это свойство объекта, являющееся функцией. Методы определяются так же, как и обычные функции, за тем исключением, что они присваиваются свойству объекта

- ```
  var user = {
      name: 'Василий',
      method_1: function( ) {
          alert( 'Привет!' );
      }
  };

  user.method_2 = function( x ) {
      alert( 'Hello, ' + x );
  };

  user.method_1( );              // Привет!
  user.method_2( 'John' );       // Hello, John
  ```
  ```

"use strict";

- Режим strict (строгий режим), введённый в ECMAScript 5, позволяет использовать более строгий вариант JavaScript. Это не просто подмножество языка: в нем сознательно используется семантика, отличающаяся от обычно принятой https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Strict_mode

- Например, строгий режим делает невозможным случайное создание глобальных переменных (т.е. без ключевого слова var или let). В обычном JavaScript опечатка в имени переменной во время присваивания приводит к созданию нового свойства глобального объекта, и выполнение продолжается

```
x = 10;         // отработает

// с использованием strict mode

"use strict";
var x = 10;   // отработает
y = 20;         // Uncaught ReferenceError: y is not defined
```

Объект Window

- Объект **window** – самый большой глобальный объект. Создается для каждого окна браузера, появляющегося на экране. Любое свойство или метод должны относиться к определенному объекту, если не относится к объекту **window**

- ```
  console.log( 'Hello' );
  window.console.log( 'Hello' );

  var user = { name: 'John' };

  console.log( user.name ); // 'John'
  console.log( name );      // ???

  // создаем три переменные
  var x = 10;
  window.y = 10;
  z = 10; // если не 'use strict';
  ```

```
if ( x == y && x == z )
    console.log( 'Значения равны' );

function foo( ) {
    // создаем две переменные
    window.a = 'Global a';   // глобальная
    var b = 'Local c';       // локальная
}
foo( );

console.log( a ); // 'Global a'
console.log( b ); // Error: b is not defined
```
```

Контекст вызова функции или что такое this?

- **this** – это ключевое слово в JavaScript которое содержит в себе объект (контекст) выполняемого кода. Контекстом всегда является какой-то объект, из под которого был вызван метод (функция).

  Так же можно сказать, что **this** это то, что находится слева от имени функции, т.е. **this**.my_function( )
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this

- **this** имеет различные значения в зависимости от того, где используется:
  - сама по себе **this** относится к глобальному объекту (window);
  - в методе **this** относится к родительскому объекту;
  - в функции **this** относится к глобальному объекту;
  - в функции в 'strict mode' **this** = undefined;

this в глобальном контексте выполнения

- В глобальном контексте выполнения (за пределами каких-либо функций) **this** ссылается на **Глобальный объект** (в браузере это **window**) вне зависимости от режима (строгий или нестрогий)

- "use strict";

```
// создадим 3 глобальные переменные
var a = 10;
window.b = 10;
this.c = 10;

if ( a == b && a == c && b == c )
    console.log( this == window ); // true
```

this в методе объекта

- Если функция хранится в объекте – это метод этого объекта. В этом случае значением переменной **this** является этот объект.
  Неудобно каждый раз писать в коде название родительского объекта, чтобы вызвать его метод (функцию), а можно написать **this**.my_function( ) вместо **window.my_object**.my_function( )

- ```
  var person = {
      name: 'John',
      say: function( ) {
          console.log( 'Hello, ' + person.name ); // Hello, John
          // или
          console.log( 'Hello, ' + this.name );      // Hello, John
          console.log( this == person );             // true
      }
  };

  // вызываем метод say
  person.say( );
  ```
  ```

this при вызове функции

- При вызове функции, значением **this** может быть либо **глобальный объект**, либо **undefined** при использовании 'use strict'

- ```
  // 'use strict';
  function foo( ) {
      console.log( this == window ); // true
  }

  foo( );
  window.foo( );
  ```
  ```

Ловушка: this во внутренней функции

- Ошибкой при работе с вызовом функции является уверенность в том, что **this** во внутренней функции такой же, как и во внешней. Вообще-то, контекст внутренней функции зависит только от вызова, а не от контекста внешней функции

- ```
  // 'use strict';
  function outer( ) {

      console.log( this == window );        // true

      function inner( ) {
          console.log( this == window );    // true
      }
      inner( );

  }
  outer( );
  ```
  ```

Лабораторная работа – 4.2

- Используйте код *Лабораторной работы 4.1*
- Добавьте объекту **book1** свойство **show**
- Свойство должно содержать анонимную функцию выводящую название и цену книги
- Опишите функцию **showBook( )**, которая выводит название и цену книги
- Добавьте объекту **book2** свойство **show**
- Свойство должно содержать функцию **showBook( )**
- Вызовите метод **show( )** у обоих объектов

Геттеры и сеттеры

- get/set – функции, которые возвращают/записывают значение свойства

- ```
  var user = {
      name: 'Guest',
      get userName( ) {
          return this.name;
      },
      set userName( value ) {
          this.name = value;
      }
  };

  // вызываем сеттер, т.е. присваиваем значение свойству
  user.userName = "John";

  // вызываем геттер, т.е. получаем значение свойства
  console.log( "Вас зовут: " + user.userName ); // Вас зовут: John
  ```
  ```

Метод функции call (вызов)

- Позволяет (временно, на момент исполнения метода) связать функцию с объектом, т.е. сделать ее методом.

  Первым аргументом в метод **call** передается **this** (имя объекта), после которого, можно передать список аргументов (необходимых для работы связанной функции), разделенных запятыми.

  Метод **call** самостоятельно вызывает функцию, к которой применяется.

  Метод **call** может быть полезен, если нужно связать объект и встроенную функцию. Так же он может пригодиться, если нужно к нескольким объектам добавить метод (функцию)

  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call

- Синтаксис:
  function_name.**call(** thisArg [, arg1 [, arg2 [, ...]]] **)**;

Метод call. Пример использования

```
var person = {
    name: "John",
    foo: function( ) {
        console.log( this == person ); // true
    },
};
person.foo( ); // вызываем метод объекта

function test( ) {
    console.log( this == person, this.name ); // true, John
}

test.call( person ); // связываем функция test с объектом person

// но ф-ция test не является методом объекта person
person.test( ); // TypeError: person.test is not a function
```

Метод функции apply (применять)

- Как и метод **call**, позволяет (временно, на момент исполнения метода) связать функцию с объектом, т.е. сделать ее методом.

  Первым аргументом в метод **apply** передается **this** (имя объекта), после которого, можно передать аргументы (необходимых для работы связанной функции) в виде массива.

  Метод **apply** самостоятельно вызывает функцию, к которой применяется.

  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

- Синтаксис:
  function_name.**apply(** thisArg, [argsArray] **)**;

Метод apply. Пример использования

```
var person = {
    name: "John",
    foo: function( ) {
        console.log( this == person ); // true
    },
};
person.foo( ); // вызываем метод объекта

function test( ) {
    console.log( this == person, this.name ); // true, John
}

test.apply( person ); // связываем функция test с объектом person

// но ф-ция test не является методом объекта person
person.test( ); // TypeError: person.test is not a function
```

И еще о функциях

- Объект **arguments** – это локальная переменная, которая доступная внутри любой (нестрелочной) функции. Объект **arguments** позволяет ссылаться на аргументы функции внутри неё. Он состоит из переданных в функцию аргументов, индексация начинается с 0

- ```
  function foo( x, y ) {
      console.log( foo.length );              // 2
      console.log( arguments.length );        // 3
      console.log( arguments[ 0 ] );          // John
      console.log( arguments.callee );        // foo( x, y ) { ... }
  }

  // смотрим кол-во аргументов ф-ции
  console.log( foo.length );                  // 2

  // вызываем ф-цию
  foo( 'John', 25, true );
  ```
  ```

Объектный тип: Массив(Array)

- Создаем пустой массив
```
var a = [ ];

if ( a ) { } // тут в if всегда будет true, даже если массив пустой
```

- Добавление элементов в массив
```
a[ 0 ] = 'Банан';
a[ 2 ] = 'Лимон';
a[ 1 ] = 'Апельсин';

console.log( a );   // (3) [ "Банан", "Апельсин", "Лимон" ]
console.log( a[ 1 ] );    // Апельсин
```

- Создаем массив и наполняем его
```
var a = [ 'Банан', 'Апельсин', 'Лимон' ];
```

- Массив представляет собой упорядоченный набор пар – **ключ**: **значение**

Добавление и замена элементов массива

- var a = ['Банан', 'Апельсин', 'Лимон'];

- Добавление элементов в массив

```
a[ ] = 'Яблоко';   // Error: Unexpected token ']', т.к. нет индекса элемента

a[ 3 ] = 'Яблоко';
a[ 4 ] = 100;
a[ 5 ] = true;
a[ 6 ] = function( ) {
    console.log( 'Hello' );
};

a[ 6 ]( );
```

- Замена элемента массива

```
a[ 1 ] = 'Груша';
console.log( a[ 1 ] );   // Груша
```

Длина массива

- В массиве возможно появление 'дыр'

- ```
  var a = [ 'one', 'two' ];
  console.log( a.length ); // 2
  ```

- ```
  a[ 9 ] = 'three';
  console.log( a.length ); // 10
  ```

- ```
  var a = [ 'one', 'two' ];
  a.length = 3;
  console.log( a );          // (3) [ 'one', 'two', empty ]
  ```

- Удаление всех элементов из массива
  ```
  a.length = 0;
  console.log( a ) ;          // [ ]
  ```

- Длина массива всегда на единицу больше индекса последнего элемента
  ```

Перебор элементов массива

- ```
  var a = [ 'one', 'two', 'three' ];
  a[ 9 ] = 'four';
  ```

- Цикл for (рекомендуется выносить определение длины массива из цикла)

  ```
  var count = a.length;

  for ( var i = 0; i < count; i++) {
      console.log( a[ i ] );      // получим элементы массива и 6 undefined
  }
  ```

- Цикл for…in

  ```
  for ( var i in a ) {
      console.log( a[ i ] );      // получим только элементы массива
  }
  ```
  ```

Лабораторная работа – 4.3

- Числа Фибоначчи

- var fibs = [1, 1], n = 10;

- Заполните массив **fibs** до 10 элементов так, чтобы каждый последующий элемент был равен сумме двух предыдущих

- Результат: (10) [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

Удаление элементов массива

- ```
  var a = [ 'one', 'two', 'three' ];

  a[ 1 ] = undefined;

  for ( var i in a ) {
      console.log( a[ i ] ); // one, undefined, three
  }
  console.log( 'Длина массива: ' + a.length ); // 3
  ```

- ```
  var a = [ 'one', 'two', 'three' ];

  delete a[ 1 ];

  for ( var i in a ) {
      console.log( a[ i ] ); // one, three
  }
  console.log( 'Длина массива: ' + a.length ); // 3
  ```

Получение строки из массива

- `var a = [ 'one', 'two', 'three' ], s;`

- Метод **toString( );** преобразует массив в строку, разделяя элементы массива запятыми

- ```
  s = a.toString( );
  console.log( s );                    // one,two,three
  ```

- Метод **join(** [separator] **);** объединяет все элементы массива в строку. Необязательный параметр **separator** определяет строку, разделяющую элементы массива

- ```
  s = a.join( );
  console.log( s );                    // one,two,three
  ```

- ```
  s = a.join( '---' );
  console.log( s, typeof a );     // one---two---three object
  ```
  ```

Сложение массивов

- ```
  var a = [ 'one', 'two' ];
  var b = [ 'four', 'five' ];
  ```

- Попробуем сложить массивы как строки:
  ```
  var s = a + b;
  console.log( s );   // one,twofour,five ???
  console.log( typeof s );     // string
  ```

- Метод **concat( );** возвращает новый массив, состоящий из массива, на котором он был вызван, соединённого с другими массивами и/или значениями, переданными в качестве аргументов

  ```
  var arr = a.concat( 3, b );
  console.log( arr, typeof arr );   // [ "one", "two", 3, "four", "five" ] "object"
  ```
  ```

Получение части массива

- var a = ['one', 'two', 'three', 'four', 'five'], arr;

- Метод **slice(** [begin [, end]] **);** возвращает новый массив, содержащий копию части исходного массива. **begin** – индекс первого извлекаемого элемента (входит). **end** – индекс последнего извлекаемого элемента (не входит)

- arr = a.slice(2), ; // ["three", "four", "five"], т.е. от **begin** и до конца

- arr = a.slice(1, 3); // ["two", "three"], т.е. от **begin** и до **end-1**

- arr = a.slice(0, a.length); // **arr** – это копия массива **a**
  - или
- arr = a.slice( );

```
arr[ 0 ] = 'Один';
console.log( a[ 0 ] );          // one, исходный массив не изменился
console.log( arr );             // [ "Один", "two", "three", "four", "five" ]
```

Сортировка массива

- Метод **array.reverse( );** обращает порядок следования элементов массива. Первый элемент массива становится последним, а последний – первым. Изменяет исходный массив

```
var a = [ 14, 51, 7, 2 ];
a.reverse( );
console.log( a ); // (4) [2, 7, 51, 14]
```

- Метод **array.sort(** [compareFunction] **);** сортирует элементы массива и возвращает отсортированный массив. Порядок сортировки по умолчанию соответствует порядку символов Unicode. Изменяет исходный массив. Необязательный параметр **compareFunction** указывает функцию, определяющую порядок сортировки

```
var a = [ 14, 51, 7, 2 ];
a.sort( );
console.log( a ); // (4) [14, 2, 51, 7]
```

Пользовательская сортировка массива

- Для числового сравнения, вместо строкового, функция сравнения может просто вычитать **b** из **a**. Следующая функция будет сортировать массив по возрастанию:
  - если mySort(a, b) **меньше 0**, сортировка поставит a перед b;
  - если mySort(a, b) **вернёт 0**, сортировка оставит a и b неизменными по отношению друг к другу, но отсортирует их по отношению ко всем другим элементам;
  - если mySort(a, b) **больше 0**, сортировка поставит b перед a

- ```
  var a = [ 14, 51, 7, 2 ];

  function mySort( a, b ) {
      return a - b;
  }

  var res = a.sort( mySort );
  console.log( res );   // (4) [2, 7, 14, 51]
  ```
  ```

Работа с концом массива

- `var a = [ 'one', 'two', 'three' ], v;`

- Извлечение элемента
  ```
  v = a.pop( );                    // [ "one", "two" ], v = "three"
  ```

- Добавление элементов
  ```
  v = a.push( 'four', 'five' );    // [ "one", "two", "four", "five" ], v = 4;
  ```

- Или, перед добавлением нового элемента определяем длину массива, т.е. индекс добавляемого элемента. И добавляем элементы по одному

  ```
  var a = [ 'one', 'two' ], index;
  index = a.length;                // 2
  a[ index ] = 'three';            // добавляем третий элемент, т.к. у него индекс 2
  index = a.length;                // 3
  a[ index ] = 'four';             // добавляем четвертый элемент, т.к. у него индекс 3
  console.log( a );                // ["one", "two", "three", "four"]
  ```

Работа с началом массива

- `var a = [ 'one', 'two', 'three' ], v;`

- Извлечение элемента
  `v = a.shift( );` `// [ "two", "three" ], v = "one"`

- Добавление элементов
  `v = a.unshift( 'zero', 'one' );` `// [ "zero", "one", "two", "three" ], v = 4`

Вставка и удаление в любом месте

- var a = ['one', 'two', 'three', 'four', 'five'], arr;

- Метод **splice(** start [, deleteCount [, item1[, item2 [, ...]]]] **);** изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые. Возвращает массив, содержащий удалённые элементы. Параметры:
  - **start** – индекс, по которому начинает изменять массив;
  - **deleteCount** – (необязательный) целое число, показывающее количество старых удаляемых из массива элементов. Если **deleteCount** равен **0**, элементы не удаляются

- ```
  arr = a.splice( 1, 2 );
  console.log( a );          // [ "one", "four", "five" ]
  console.log( arr );        // ["two", "three"]
  ```

- ```
  arr = a.splice( 1, 0, 'Новый элемент' );
  console.log( a );          // [ "one", "Новый элемент", "four", "five" ]
  console.log( arr );        // [ ]
  ```

Методы массива (ES-2015)

- Метод **forEach( );** выполняет указанную функцию один раз для каждого элемента в массиве:

```
var nums = [ 1, 2, 3, , 5 ];
nums.forEach( function( v ) {
    console.log( v * 10 );      // выведет 10, 20, 30, 50
} );
```

- Метод **map( );** позволяет вызвать переданную функцию один раз для каждого элемента массива, формируя новый массив из результатов вызова этой функции:

```
var nums = [ 1, 4, 9 ];
var result = nums.map( function( v ) {
    return v * 10;
} );
console.log( result );          // (3) [10, 40, 90]
```

Цикл for…of (ES-2015)

- Цикл **for...in** обходит имена свойств объекта, но **for...of** выполняет обход значений свойств. Однако, с помощью **for...in** можно обращаться и к значениям

- ```
  var arr = [ 'Банан', 'Апельсин', 'Лимон' ];

  for ( var i in arr ) console.log( i );        // 0, 1, 2

  for ( var i of arr ) console.log( i );        // 'Банан', 'Апельсин', 'Лимон'
  ```

- По умолчанию объекты { } не являются итерабельными

  ```
  var user = { name: 'John', age: 25, admin: true };

  for ( var i in user ) console.log( i );       // name, age, admin

  for ( var i of user ) console.log( i );       // TypeError: user is not iterable
  ```
  ```

Лабораторная работа – 4.4

- В классе '5Б' учится 10 учеников, из которых 6 девочек и 4 мальчика.
  Задача:
  – обеспечить целостность хранения данных;
  – заполнить данные всех учеников класса;
  – вывести в консоли отдельно список мальчиков и список девочек;
  – сортировкой по фамилиям учеников пренебречь;
  – само собой, в школе не один класс;
  – предусмотрите возможность добавления дополнительных сведений ученику, таких как: увлечения спортом, дата рождения и т.д.

  'Аня К.', 'Маша Б.', 'Оля С.', 'Таня О.', 'Катя А.', 'Юля М.', 'Никита А.', 'Рома Б.', 'Петя В.', 'Костя Н.'

Деструктуризация (ES-2015)

- Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

- ```
  var user = [ "Иван", "Иванов" ];

  var [ firstName, lastName ] = user;

  console.log( firstName );    // Иван
  console.log( lastName );    // Иванов
  ```
  ```

Метод includes (ES-2015)

- Определяет, содержит ли массив определённый элемент, возвращая в зависимости от этого **true** или **false**
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/includes

- ```
  var pets = [ 'cat', 'dog', 'bat' ], str = 'cat';

  if ( pets.includes( str ) )
     console.log( `Элемент ${ str } находится в массиве` );
  else
     console.log( `Элемент ${ str } отсутствует в массиве` );
  ```
  ```

Метод indexOf

- Метод **arr.indexOf(** searchElement [, fromIndex = 0] **);** возвращает первый индекс, по которому данный элемент может быть найден в массиве или -1, если такого индекса нет. Параметр **searchElement** – искомый элемент в массиве. Параметр **fromIndex** – индекс, с которого начинать поиск

- ```
  var pets = [ 'cat', 'dog', 'bat' ];

  console.log( pets.indexOf( 'dog' ) );       // 1

  console.log( pets.indexOf( 'squirrel' ) );  // -1
  ```
  ```

Встроенный объект Math

- Объект **Math** является встроенным объектом, хранящим в своих свойствах и методах различные математические константы и функции

- Свойства
  E, LN2, SQRT2, PI…

- Методы
  cos( ), acos( ), sin( ), asin( )…
  pow( ), sqrt( )…
  floor( ), round( ), ceil( ), random( )
  max( ), min( )

Встроенный объект Math: примеры использования – 1

- Получение числа *π*

```
Math.PI;      // 3.141592653589793
```

- Возведение числа 3 в степень 2

```
Math.pow( 3, 2 );                    // 9
```

- Нахождение наибольшего числа

```
Math.max( 5, 0, -10 );               // 5
```

- Нахождение наибольшего элемента массива. Метод apply

```
var arr = [ 5, 0, -10, 15, 1 ];
var res1 = Math.max.apply( null, arr );

// или используя новый синтаксис – оператор spread
var res2 = Math.max( ...arr );
console.log( res1, res2 );           // 15 15
```

Встроенный объект Math: примеры использования – 2

- Метод **Math.floor( );** округляет аргумент до ближайшего меньшего целого
```
Math.floor( 3.5 ); // 3
```

- Метод **Math.ceil( );** округляет аргумент до ближайшего большего целого
```
Math.ceil( 3.5 );   // 4
```

- Метод **Math.round( );** округляет аргумент до ближайшего целого
```
Math.round( 3.5 );                    // 4
Math.round( 3.49 );                   // 3
```

- Метод **Math.random( );** возвращает случайное число с плавающей запятой из диапазона [0, 1), то есть, от 0 (включительно) до 1 (но не включая 1)

- Генерация случайного числа в диапазоне от n1 до n2
```
var n1 = 0, n2 = 10, res;
res =  Math.random( ) * ( n2 - n1 ) + n1;
console.log( Math.round( res );
```

Лабораторная работа – 4.5

- Гороскоп
- Создайте массив **arr** с несколькими строковыми элементами, которые описывают сегодняшний день. Например, 'Сегодня прекрасный день', Сегодня хорошая погода'…
- Выведите на экран случайное описание дня

Выводы

- Объектный тип: Объект (Object)
- Свойства объекта
- Методы объекта
- Объектный тип: Массив (Array)
- Свойства и методы функций
- Встроенный объект Math

Модуль 5

Объектно-ориентированное программирование

Темы модуля

- Конструкторы объектов
- Прототипы
- Классы
- Объект Date

Что мы имеем?

- Объектно-ориентированный JavaScript простыми слова https://habr.com/ru/post/522380/

- ```
  var work_car = {
      seats: 'cloth',
      engine: 'V-6',
      carinfo: function( ) {
          console.log( 'Салон: ' + this.seats + '; двигатель: ' + this.engine );
      }
  };

  var fun_car = {
      seats: 'leather',
      engine: 'V-8',
      carinfo: function( ) {
          console.log( 'Салон: ' + this.seats + '; двигатель: ' + this.engine );
      }
  };
  ```
  ```

Конструкторы

- ```
  function Car( seats, engine ) {
      this.seats = seats;
      this.engine = engine;
      this.carinfo = function( ) {
          console.log( 'Салон: ' + this.seats + '; двигатель: ' + this.engine );
      }
  }
  ```

- ```
  var work_car = new Car( 'cloth', 'V-6' );
  var fun_car = new Car( 'leather', 'V-8' );
  ```

- ```
  var seat_type = fun_car.seats;  // присвоили переменной свойство объекта
  ```

- ```
  work_car.engine = 'V-4';          // изменили свойство объекта
  work_car.carinfo( );              // Салон: cloth; двигатель: V-4
  ```

- ```
  console.log( fun_car.constructor ); // function Car( seats, engine ) { ... };
  console.log( fun_car.constructor.name ); // Car
  ```
  ```

Пример использования объектов (забегая вперед…)

- Vue – это фреймворк для создания пользовательских интерфейсов

- В этом руководстве мы предполагаем, что вы уже знакомы с HTML, CSS и JavaScript на базовом уровне. Если же вы во фронтенд-разработке совсем новичок, начинать сразу с изучения фреймворка может быть не лучшей идеей

  https://ru.vuejs.org/v2/guide/ (Декларативная отрисовка)

Фабричные функции

- Они дают возможность создавать экземпляры объектов без погружения в запутанность классов и ключевого слова **new**

- ```
  function car( seats, engine ) {
      return {
          seats,
          engine,
          carinfo( ) {
              console.log( 'Салон: ' + seats + '; двигатель: ' + engine );
              console.log( work_car == this ); // true
          },
      }
  }

  var work_car = car( 'cloth', 'V-6' );
  console.log( typeof car, typeof work_car ); // function object

  work_car.carinfo( ); // Салон: cloth; двигатель: V-6
  ```
  ```

Новый взгляд на привычное

- var n = new Number(5), n1 = 5;
  console.log(n); // Number{5}
  console.log(n1); // 5

  n == n1; // true
  n === n1; // false

- var s = new String('John');
  console.log(s); // String {"John"}, 0: "J" 1: "o" 2: "h" 3: "n"

- var b = new Boolean(true);

- var a = new Array(5);
- var o = new Object( );

- var re = new RegExp('[a-z+]','i');

Проверка конструктора

- ```
  var arr = [ ];
  console.log( arr instanceof Array ); // true
  ```

- Оператор **instanceof** позволяет определить был ли данный конструктор задействован в создании объекта. Возвращает либо true, либо false

- ```
  function Car( seats, engine ) {
      this.seats = seats;
      this.engine = engine;
      this.carinfo = function( ) {
          console.log( 'Салон: ' + this.seats + '; двигатель: ' + this.engine );
      }
  }

  // создаем экземпляр объекта Car
  var work_car = new Car( 'cloth', 'V-6' );

  console.log( work_car instanceof Car ); // true
  ```

Прототипы

- Чтобы добавить св-во к ранее определённому типу объекта – используется специальное св-во **prototype**. Оно создаёт св-во, единое для всех объектов данного типа, а не одного экземпляра этого типа объекта

- ```
  function Car( seats, engine ) {
      this.seats = seats;
      this.engine = engine;
      this.carinfo = function( ) {
          console.log( 'Салон: ' + this.seats + '; двигатель: ' + this.engine );
      }
  }

  Car.prototype.car_color = null;

  var work_car = new Car( 'cloth', 'V-6' );
  work_car.car_color = 'Blue';

  console.log( 'Цвет машины: ' + work_car.car_color ); // Blue
  ```
  ```

Прототипы, пример. Добавление своего метода в Array

- Массив (Array) в JavaScript является глобальным объектом, который используется для создания массивов, которые представляют собой высокоуровневые спископодобные объекты. Добавим в него (в прототип) метод **myFunc**, который считает сумму элементов массива

- ```
  var arr = [ 1, 2, 3, 4, 5 ];

  Array.prototype.myFunc = function( ) {
      console.log( this == arr ); // true

      var count = this.length, result = 0;
      for( var i = 0; i < count; i++ )
          result += this[ i ];

      return result;
  }

  console.log( arr.myFunc( ) ); // 15
  ```
  ```

Методы объекта

- ```
  var n = new Number( 5 );
  console.log( n ); // Number {5} ???

  var s = new String( 'John' );
  console.log( s ); // String {"John"} ???
  ```

- Метод **valueOf( );** возвращает примитивное значение указанного объекта
  ```
  console.log( n.valueOf( ) ); // 5
  console.log( s.valueOf( ) ); // John
  ```

- Метод **hasOwnProperty( );** возвращает логическое значение, указывающее, содержит ли объект указанное свойство
  ```
  var obj = { x: 10 };
  console.log( obj.hasOwnProperty( 'x' ) ); // true
  ```

- Использовался ли прототип при создании нашего объекта
  ```
  console.log( Object.prototype.isPrototypeOf( obj ) ); // true
  ```
  ```

Лабораторная работа – 5.1

- Создайте конструктор **book** со свойствами:
  - title
  - pubYear
  - price

- Создайте объект типа **book** передав в конструктор произвольные значение

- Добавьте в **прототип** конструктора **book** метод **show**, который выводит значения свойств **title** и **price**

- Вызовите метод **show** созданного объекта

Классы (ES-2015)

- Классы в JavaScript были введены в ECMAScript 2015 и представляют собой синтаксический сахар над существующим в JavaScript механизмом прототипного наследования. Синтаксис классов не вводит новую объектно-ориентированную модель, а предоставляет более простой способ создания объектов

- На самом деле классы – это "специальные функции", поэтому точно также, как вы определяете функции (function expressions и function declarations), вы можете определять и классы с помощью: class declarations и class expressions

- Разница между объявлением функции (function declaration) и объявлением класса (class declaration) в том, что объявление функции совершает подъём (hoisted), в то время как объявление класса – нет

```
foo( );
function foo( ) { }

var p = new Rectangle( ); // ReferenceError
class Rectangle { };
```

class declaration

- Метод **constructor( )**; специальный метод, необходимый для создания и инициализации объектов, созданных, с помощью класса. В классе может быть только один метод с именем constructor

- ```
  class Car {
      constructor( seats, engine ) {
          this.seats = seats;
          this.engine = engine;
      }

      carinfo( ) {
          return `Салон: ${this.seats}; мотор: ${this.engine}`;
      }
  }

  var work_car = new Car( 'cloth', 'V-6' );

  console.log( work_car.carinfo( ) ); // Салон: cloth; мотор: V-6
  ```
  ```

Статические свойства и методы

- Статическое методы (и свойства) вызываются через имя класса. Вызывать статические методы через имя объекта запрещено. Статические методы часто используются для создания вспомогательных функций приложения

- ```
  class Car {
      constructor( seats, engine ) {
          this.seats = seats;
          this.engine = engine;
      }

      // статическое свойство
      static num = 1;

      // статический метод
      static car_count( ) {
          return Car.num++;
      }

      carinfo( ) {
          return `Салон: ${this.seats};
              мотор: ${this.engine};
              номер: ${Car.car_count( )}`;
      }

  } // завершение класса Car

  var work_car = new Car( 'cloth', 'V-6' );
  var fun_car = new Car( 'leather', 'V-8' );

  console.log( fun_car.carinfo( ) );
  console.log( work_car.carinfo( ) );
  ```
  ```

Наследование классов

- Ключевое слово **extends** может быть использовано для создания дочернего класса для уже существующего класса

- Ключевое слово **super** используется для вызова функций, принадлежащих родителю объекта

- ```
  class Car {
      constructor( seats, engine ) {
          this.seats = seats;
          this.engine = engine;
      }

      carinfo( ) {
          return `Салон: ${this.seats}; мотор: ${this.engine}`;
      }
  }
  ```
  ```

```
// создаем class Truck, являющийся наследником class Car

class Truck extends Car {
    constructor( seats, engine, tonnage ) {
        super( seats, engine );
        this.tonnage = tonnage;
    }

    trucinfo( ) {
        return `Салон: ${this.seats}; мотор: ${this.engine}; тоннаж: ${this.tonnage}`;
    }
}

var truck_car = new Truck( 'cloth', 'V-12', '10 ton' );

console.log( truck_car.carinfo( ) );

console.log( truck_car.trucinfo( ) );
```

class expression

- безымянный

- ```
  var Car = class {
      constructor( seats, engine ) {
          this.seats = seats;
          this.engine = engine;
      }
  };
  ```

- именованный

- ```
  var Car = class Car {
      constructor( seats, engine ) {
          this.seats = seats;
          this.engine = engine;
      }
  };
  ```

Лабораторная работа – 5.2

- Создайте класс **Book** со свойствами:
  - title
  - pubYear
  - price

- Создайте экземпляры класса **book1** и **book2** со свойствами: title, pubYear и price с произвольными значениями

- Опишите для класса **Book** метод **showBook( )**, который выводит:
  - название;
  - год издания;
  - цену книги

- Вызовите метод **showBook( )** у обоих экземпляров класса **Book**

Объект Date

- Время хранится в миллисекундах

- Точка отсчета 01 января 1970 года (00:00:00)

- **Greenwich MeanTime (GMT)**
  - Дата и время указываются в соответствии с местным часовым поясом
  - Указывается смещение относительно Гринвического меридиана
  - Смещение зависит от перехода на летнее и зимнее время

- **Universal TimeCoordinated (UTC)**
  - Дата и время в любой точке планеты одинаково
  - Точка отсчета совпадает с точностью до долей секунды с точкой отсчета GMT
  - Никаких переходов на летнее и зимнее время в UTC нет

Создание объекта Date

- var d = new Date( );
  console.log(d); // выведет текущее время

- var d = new Date('Jan 01 2018 01:00:00');
  console.log(d); // Mon Jan 01 2018 01:00:00 GMT+0300 (Москва, ...)

- var d = new Date(1234567890000);
- var d = new Date(2018, 2);
  - **год**, **номер месяца**, дата, часы, минуты, секунды, миллисекунды

- var d = new Date(2018, 2, 28);

- Передавать дату можно с ошибками
  var d = new Date(2018, 2, 40);
  var d = new Date(2018, -2);

Методы объекта Date

- getFullYear( )
- getMonth( ) **0 – 11**
- getDate( ) 1 – 31
- getHours( ) 0 – 24
- getMinutes( ) 0 – 59
- getSeconds( ) 0 – 59
- getMilliseconds( ) 0 – 999
- getDay( ) 0 – 6
  - 0 – Воскресенье!

- Получаем правильный индекс дня (по остатку от деления):

```
/* П В С Ч П С Вс
   1 2 3 4 5 6 0 – имеем
   0 1 2 3 4 5 6 – надо */

var day = new Date( ).getDay( );
( day + 6 ) % 7;
```

Дополнительные методы

- **getTimezoneOffset( );** возвращает смещение часового пояса относительно часового пояса UTC в минутах для текущей локали
  - ```
    var d = new Date( );
    console.log( d.getTimezoneOffset( ) ); // -180
    ```

- **getTime( );** возвращает кол-во миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC, либо до указанной даты
  - ```
    var d = new Date( 'Feb 28 2018 03:45:15' );
    console.log( d.getTime( ) ); // 1519778715000
    var d = new Date( );
    console.log( d.getTime( ) ); // 1558520678894
    ```

- **Date.parse( );** разбирает строковое представление даты и возвращает количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 по UTC
  - ```
    var n = Date.parse( 'Feb 28 2018 03:45:15' );
    console.log( n );  // 1519778715000
    var d = new Date( n );
    console.log( d );  // Wed Feb 28 2018 03:45:15 GMT+0300 (Москва, ...)
    ```
    ```

Строчное представление даты

- var d = new Date( );

- **toString( );** возвращает строковое представление объекта Date
  console.log(d.toString( )); // Wed May 22 2019 13:07:29 GMT+03…

- **toUTCString( );** преобразует дату в строку, используя часовой пояс UTC
  console.log(d.toUTCString( )); // Wed, 22 May 2019 10:07:29 GMT

- **toTimeString( );** возвращает часть, содержащую только время объекта Date в виде человеко-читаемой строки на американском английском
  console.log(d.toTimeString( )); // 13:07:29 GMT+0300 (Москва, ...)

- **toDateString( );** возвращает часть, содержащую только дату объекта Date в виде человеко-читаемой строки на американском английском
  console.log(d.toDateString( )); // Wed May 22 2019

Запись информации

- setFullYear( )
  - setUTCFullYear( )
- setMonth( )
- setDate( )
- setDate( )
- setHours( )
- setMinutes( )
- setSeconds( )
- setMilliseconds( )
- setTime( ) не имеет UTC

- Установка значения года. Остальные значения времени берутся текущие

```
var d = new Date( );
d.setFullYear( 1997 );
console.log( d ); // Fri May 23 1997 11:53:04 GMT+0400 (Москва, …)
```

Лабораторная работа – 5.3

- Вывести на экран текущую дату в формате '28 февраля 2018 года'

Лабораторная работа – 5.4

- Вывести на экран количество дней оставшихся до Нового года

Лабораторная работа – 5.5

- Написать программу, определяющую время работы цикла

- ```
  for (var i = 0; i < 100000; i++) {
      var res = i * i * i;
  }
  ```
  ```

Выводы

- Конструкторы объектов
- Прототипы
- Классы
- Объект Date

Модуль 6

Дополнительная информация

Темы модуля

- Дополнительная информация по функционалу JavaScript
- Дополнительные встроенные объекты

Объект Number. Статические св-ва

- Значение отрицательной бесконечности
  - Number.NEGATIVE_INFINITY

- Значение положительной бесконечности
  - Number.POSITIVE_INFINITY

- Значение «не число». Эквивалентно глобальному объекту NaN
  - Number.NaN

- Минимальное положительное числовое значение, примерно равное 5e-324
  - Number.MIN_VALUE

- Максимальное числовое значение, примерно равное 1.79E+308
  - Number.MAX_VALUE

Преобразование числа в строку

- // метод toString( ); возвращает строковое представление указанного объекта
  var x = 10;
  var y = x.toString( );
  console.log(typeof y); // string

- var n = 12345.6789;

- // метод toFixed( ); форматирует число, используя запись с фиксированной запятой
  n.toFixed(2); // 12345.68

- // метод toExponential( ); возвращает строку, представляющую объект Number в экспоненциальной записи
  n.toExponential(2); // 1.23e+4

- // метод toPrecision( ); возвращает строку, представляющую объект Number с указанной точностью
  n.toPrecision(6); // 12345.7

Свойства и методы глобального объекта

- Свойство **Infinity** является числовым значением, представляющим бесконечность

- Свойство **NaN** является значением, представляющим не-число (Not-A-Number)

- **NaN** != **NaN**

- // функция isNaN( ); определяет является ли литерал или переменная нечисловым значением (NaN) или нет
  isNaN(3 * 'a'); // true

- // глобальная isFinite( ); функция определяет, является ли переданное значение конечным числом
  isFinite(3 * 'a'); // false
  isFinite(3 / 0); // false

Использование систем счисления

- Метод **toString(** [radix] **);** преобразует число в строку. Необязательный параметр **radix** указывает, в какую систему счисления (от 2 до 36) следует преобразовывать число

- ```
  var num = 255, str;
  str = num.toString( 16 );
  console.log( str ); // ff
  ```

- Функция **parseInt(** string, radix **);** принимает строку в качестве аргумента и возвращает целое число в соответствии с указанным основанием системы счисления. Параметр **string**, это значение, которое необходимо проинтерпретировать. Параметр **radix**, это целое число в диапазоне между 2 и 36, представляющее собой основание системы счисления числовой строки **string**, описанной выше

- ```
  var str = 'ff', num;
  num = parseInt( str, 16 );
  console.log( num ); // 255
  ```

Длина строки

```
var str = 'Это \u2014\u0020просто\x20пример';
```

- https://unicode-table.com/ru/2014/

- ```
  console.log( str );          // Это – просто пример
  console.log( str.length );   // 19
  ```

- В JavaScript длина строки измеряется в символах. Однако, в некоторых языках (например, php) длина строки определяется в байтах

- В кодировке utf-8, 1 буква английского языка занимает 1 байт.
  Однако, 1 кириллическая буква занимает 2 байта
  ```

Сложение строк

- ```
  var s1 = 'Это';
  var s2 = ' \u2014\x20';
  var s3 = 'просто пример';
  ```

- ```
  var s = s1 + s2 + s3;
  ```

- ```
  var s = s1.concat( s2, s3 );
  ```

- ```
  console.log( s ); // Это — просто пример
  ```

Регистр символов и лишние пробелы

- `var s = '   Test   ';`

- Метод **toLowerCase( );** возвращает значение строки, преобразованное в нижний регистр
```
var x = s.toLowerCase( );
console.log( x, x.length );   // test, 6
```

- Метод **toUpperCase( );** возвращает значение строки, преобразованное в верхний регистр
```
var x = s.toUpperCase( );
console.log( x, x.length );   // TEST, 6
```

- Метод **trim( );** удаляет пробельные символы с начала и конца строки
```
var x = s.trim( );
console.log( x, x.length );   // Test, 4
```

- `var x = s.toLowerCase( ).trim( );`
```
console.log( x, x.length );   // test, 4
```

Получение символа из строки

- var s = 'просто пример';

- Метод **charAt( );** возвращает указанный символ из строки

```
var x = s.charAt( 4 );          // 'т'
var x = s.charAt( 40 );         // ''
```

- Метод **charCodeAt( );** возвращает числовое значение Юникода для символа по указанному индексу

```
var x = s.charCodeAt( 4 );      //1090 (U+0442)
var x = s.charCodeAt( 40 );     // NaN
```

Создание строки из кодов символов

- Статический метод **String.fromCharCode( );** возвращает строку, созданную из указанной последовательности значений единиц кода UTF-16.
  Поскольку метод **fromCharCode( );** является статическим методом объекта String, вы всегда должны использовать его как **String.fromCharCode( );** а не как метод созданного вами экземпляра String

```
var x = String.fromCharCode( 1051, 1091, 1085, 1072 ); // Луна
```

- Передача массива в качестве параметра

```
var arr = [ 116, 101, 115, 116, 64, 109, 97, 105, 108, 46, 114, 117 ];

// применяем метод apply
var res1 = String.fromCharCode.apply( null, arr );

// или используя новый синтаксис – оператор spread
var res2 = String.fromCharCode( ...arr );

console.log( res1, res2 ); // test@mail.ru test@mail.ru
```

Лабораторная работа – 6.1

- Описать функцию **isPalindrom( )**, которая проверяет строку, является та палиндромом или нет

Лабораторная работа – 6.2

- Последовательно преобразовать адрес электронной почты **info@my-syte.ru** в коды символов

- Вывести на экран адрес электронной почты из кодов символов, используя метод **String.fromCharCode( );**

Получение части строки

- `var s = 'просто пример';`

- Метод **slice( );** возвращает новый массив, содержащий копию части исходного массива

- Метод **substring( );** возвращает подстроку строки между двумя индексами, или от одного индекса и до конца строки

- ```
  var x = s.slice( 3, 6 );        // сто
  var x = s.substring( 3, 6 );    // сто
  ```

- ```
  var x = s.slice( 6 );           // пример
  var x = s.substring( 6 );       // пример
  ```

- ```
  var x = s.slice( );             // просто пример
  var x = s.substring( );         // просто пример
  ```
  ```

Поиск по строке

- var x = s.indexOf('пр'); // 6
  var x = s.indexOf('пр', 7); // 13

- var x = s.lastIndexOf('пр'); // 13
  var x = s.lastIndexOf('пр', 9); // 6

Лабораторная работа – 6.3

- Исходные данные:
  - var s = 'Мы знаем, что монохромный цвет – это градации серого';
  - var txt = 'хром';
  - var word;

- В значении переменной **s** найдите слово, содержащее подстроку, которая является значением переменной **txt** и присвойте его переменной word

Замена в строке

- `var s = 'Это просто пример';`
- `var x = s.replace( 'просто', 'сложный' );`
- `// 'Это сложный пример'`

Разбиение строк

- Метод **str.split(** [separator [, limit]] **);** разбивает объект String на массив строк путём разделения строки указанной подстрокой. Параметр **separator** указывает символы, используемые в качестве разделителя внутри строки. Параметр **limit** – это целое число, определяющее ограничение на количество найденных подстрок

- ```
  var monthString = 'Янв,Фев,Мар,Апр,Май,Июн,Июл,Авг,Сен,Окт,Ноя,Дек';

  var arr = monthString.split( ',' );
  console.log( arr.length );    // 12

  arr = monthString.split( ',', 6 );
  console.log( arr );           // (6) [ "Янв", "Фев", "Мар", "Апр", "Май", "Июн" ]
  ```
  ```

Регулярные выражения

- Это мощное средство поиска и замены в строке. В JavaScript регулярные выражения реализованы отдельным объектом **RegExp** и интегрированы в методы строк
  https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions

- Конструктор RegExp (позволяет использовать переменные) создаёт объект регулярного выражения для сопоставления текста с шаблоном:

```
var ptrn = new RegExp( "шаблон", "флаги" );
```

- Как правило, используют более короткую запись (шаблон внутри слешей "/"):

```
var ptrn = /шаблон/;              // без флагов
var ptrn = /шаблон/gmi;           // с флагами gmi (изучим их дальше)
```

- Слеши "/" говорят JavaScript о том, что это регулярное выражение. Они играют здесь ту же роль, что и кавычки для обозначения строк

Базовое использование

- Метод **test(** str **);** объекта RegExp проверяет, есть ли совпадение в строке относительно шаблона.
  Возвращает true если совпадение обнаружено, иначе, вернет false

- ```
  var str = 'Это просто пример';
  var ptrn = /просто/;
  var result = ptrn.test( str );
  console.log( result );   // true
  ```

- Основа регулярного выражения – **паттерн**. Это строка, которую можно расширить специальными символами, которые делают поиск намного мощнее

- Регулярные выражения обычно содержат символьные (алфавитные) данные, цифровые данные и специальные символы **^ $. * + ? = ! : | \ / ( ) [ ] { }**

- Эти символы используются при выборке из текста каких-то данных. Идея регулярного выражения заключается в написании правила, по которому будет что-то находится или что-то проверяться
  ```

Методы объекта String

- **replace(** ptrn, str **);** используется для замены информации в строке
- **split(** ptrn **);** используется для разбиения строки на массив подстрок
- **search(** ptrn **);** возвращает позицию первого символа найденного вхождения. Если совпадений не найдено, вернет -1
- **match(** ptrn **);** сравнивает регулярное выражение со строкой, что бы определить их соответствие. Метод возвращает массив, состоящий из одного или нескольких соответствий. Если совпадений не найдено, вернет -1

Метод replace( )

- ```
  var str = 'Привет, привет, новый день';
  var ptrn = /привет/;
  var result = str.replace( ptrn, 'Ура' );
  console.log( result );   // Привет, Ура, новый день
  ```

- Флаги:
  - i (ignoreCase) – игнорирование регистра при сопоставлении;
  - g (global) – глобальное сопоставление;
  - m (multiline) – поиск осуществляется в множественной строке

- ```
  var str = 'Привет, привет, новый день';
  var ptrn = /привет/ig;
  var result = str.replace( ptrn, 'Ура' );
  console.log( result );   // Ура, Ура, новый день
  ```

- ```
  var ptrn1 = /привет/ig, ptrn2 = /новый/ig,
  var result = str.replace( ptrn1, 'Ура' ).replace( ptrn2, 'отличный' );
  console.log( result );   // Ура, Ура, отличный день
  ```
  ```

Лабораторная работа – 6.4

- Замените в предложенном тексте все теги **** на теги ****
- Добавьте тегу **** атрибут **class='red'**

Метод split( )

- ```
  var str = 'John; Paul; George; Ringo';
  var ptrn = /;\s/;
  var result = str.split( ptrn );
  console.log( result.length ); // 4
  ```

- Коды регулярных выражений:
  - \s – соответствует одиночному пробелу, символу (например, пробелу или символу разделителя строк)
  - \S – соответствует отдельному символу, если он не является пробельным
  ```

Метод search( )

- ```
  var str = 'Это просто пример';
  var ptrn = /пр/;
  var result = str.search( ptrn );
  console.log( result ); // 4
  ```
  ```

Метод match( )

- ```
  var str = 'Купил за 100, а продал за 200';
  // var ptrn = /\d\d\d/g;
  var ptrn = /\d{3}/g;
  var result = str.match( ptrn );
  console.log( result ); // (2) ["100", "200"]
  ```

- ***index*** – псевдосвойство массива, созданного в результате сравнения на соответствие регулярному выражению. Для такого массива данное свойство содержит индекс найденного соответствия в исследуемой строке

- Коды регулярных выражений:
  - \d – соответствует одиночному цифровому символу
  - {3} – соответствует числу появлений предшествующего ему символа
  ```

Методы объекта RegExp

- **test(** str **);**

- **exec(** ptrn **);** возвращает массив из единственного (первого) вхождения, но предоставляет о нем полную информацию. Если совпадений нет, вернет null

Метод test( )

- ```
  var str = 'Купил за 99, а продал за 100', i = 0;
  var ptrn = /\d{2,3}/g;

  while( ptrn.test( str ) ) {
      console.log( ptrn.lastIndex ); // Выведет 11, 28
      i++;
  };

  console.log( 'Количество совпадений = ' + i ); // Количество совпадений = 2
  ```

- ***lastIndex*** – целочисленное свойство (доступно для записи), которое указывает индекс, с которого начать следующий поиск методом test( ) или exec( ). Это свойство работает только, если установлен флаг **g**
  ```

Метод exec( )

- var str = 'Корпоративная электронная почта info2@my-site.ru, электронная почта технической поддержки support@my-site.ru', result = [ ];

  ```
  var ptrn = /([a-z\d]+)@([a-z-]+)\.([a-z]{2,})/ig;

  while( result = ptrn.exec( str ) ) {
      console.log( result[ 0 ] + ' : ' + result[ 'index' ] );
  };

  // Выведет info@mysite.ru : 32, support@mysite.ru : 90
  ```

- Коды регулярных выражений:
  - + – соответствует появлению предшествующего ему символа от одного и более раз
  - {2,} – соответствует числу появлений предшествующего ему символа не менее 2-х раз

Классы символов

- var ptrn = /[abc]/;
  var ptrn = /[^abc]/;
  var ptrn = /[a-f]/;

- . любой символ, кроме перевода строки
  \w любой символ ASCII [a-zA-Z0-9_]
  \W противоположное \w [^a-zA-Z0-9_]
  \d любая цифра ASCII [0-9]
  \D противоположное \d [^0-9]
  \s любой символ-разделитель в Unicode
  \S противоположное \s

Повторения (квантификаторы)

- { n, m }
  - шаблон повторяется не менее n, но не более m раз

- { n, } { ,n }
  - шаблон повторяется не менее или не более n раз

- { n }
  - шаблон повторяется точно n раз

- ? эквивалентно { 0, 1 }
  + эквивалентно { 1, }
  * эквивалентно { 0, }

Лабораторная работа – 6.5

- Составить регулярное выражение, которое соответствует следующим форматам даты

  - 25-02-2012
  - 25-2-2012
  - 01-12-2011
  - 2-12-1978

- Для проверки использовать метод test( )

Позиции соответствия

- ^ поиск с начала строки
  $ поиск с конца строки
  \b позиция между символами ASCII и не символом ASCII (граница слова)
  \B позиция между двумя символами ASCII (не граница слова)

- var ptrn = /^abc/;
  var ptrn = /abc$/;
  var ptrn = /\bjava\b/;
  var ptrn = /\bjava\B/;

Группировка и ссылки

- Часть шаблона можно заключить в скобки (...). Это называется «скобочная группа». У такого выделения есть два эффекта:
  - позволяет поместить часть совпадения в отдельный массив;
  - если установить квантификатор после скобок, то он будет применяться ко всему содержимому скобки, а не к одному символу

- https://learn.javascript.ru/regexp-groups

- ```
  var str = '1abcde';
  var ptrn = /ab(cd)?/;
  var result = ptrn.exec( str );
  console.log( result ); // (2) ["abcd", "cd", index: 1, input: "1abcde]
  ```
  ```

Ищем содержимое в одинаковых кавычках

- ```
  var str = "Привет, \"мир'!";

  var ptrn = /[""][^""]*[""]/;   // проблема
  var result = ptrn.exec( str );
  console.log( result );     // [""мир"", index: 8, input: "Привет, "мир'!"]
  ```

- Часть шаблона, заключенная в круглые скобки, помещается RegExp-ую переменную, к которой (точнее, к тому, что он нашел) мы можем обращаться по ее индексу. Нумерация начинается с 1

  ```
  var ptrn = /([""])[^""]*\1/;  // ссылка \1
  var result = ptrn.exec( str );
  console.log( result );     // null
  ```
  ```

Внешние ссылки (скобочные группы при замене)

- Метод **str.replace(** regexp, replacement **);** осуществляющий замену совпадений с **regexp** в строке **str**, позволяет использовать в строке замены содержимое скобок. Это делается при помощи обозначений вида **$n**, где **n** – номер скобочной группы

- ```
  var str = '123 John';
  var ptrn = /^(\d+)\s([a-z]+)$/i;
  var result = str.replace(ptrn, '$2 – $1');
  console.log( result ); // John – 123
  ```
  ```

Переменная в регулярном выражении

- Используйте конструктор для создания объекта RegExp из переменной

- ```
  var str = 'We have cats and Dogs'; // где ищем

  var txt = 'dog'; // что ищем

  var ptrn = new RegExp( '\\s'+txt+'s', 'i' );

  // выводим получившийся шаблон регулярного выражения
  console.log( ptrn ); // /\sdogs/i

  var result = ptrn.test( str );

  if ( result )
      console.log( 'Совпадение найдено' );
  else
      console.log( 'Совпадений нет' );
  ```
  ```

Объект Error

- Конструкция **try**…**catch** помечает блок инструкций как **try**, и в зависимости от того, произошла ошибка или нет, вызывает дополнительный блок инструкций **catch**

- ```
  try {
      var x = 10;
      console.log( y );
      console.log( x );
  } catch( e ) {
      console.log( e.name );       // ReferenceError
      console.log( e.message );  // y is not defined
      console.log( 'Ошибка есть, но идем дальше' );
  };

  console.log( 'x = ' + x );
  ```
  ```

Конструктор объекта Error

- Конструктор **Error** создаёт объект ошибки. Экземпляры объекта **Error** выбрасываются при возникновении ошибок во время выполнения. Объект **Error** также может использоваться в качестве базового для пользовательских исключений.

  Обычно, вы создаёте объект **Error** с намерением возбудить ошибку с помощью ключевого слова **throw**. Вы можете обработать ошибку с помощью конструкции **try...catch**

- ```
  try {
      var a = 1, b = 0, x;
      if ( b == 0 )
          throw new Error( 'Ошибка!' );
      x = a / b;
  } catch( e ) {
      console.log( e.message ); // Ошибка!
  };
  ```
  ```

Babel.JS

- babeljs.io
- Babel.JS – это транспайлер, переписывающий код на ES-2015 в код на предыдущем стандарте ES5
- Ссылка: unpkg.com/babel-standalone@6/babel.min.js

- Следующий код сработает начиная с **IE11**

```
<script>
let x = 10;
alert( x );
</script>
```

- Однако, следующий код сработает начиная с **IE9**

```
<script src='https://unpkg.com/babel-standalone@6.26.0/babel.min.js'></script>
<script type="text/babel">
let x = 10;
alert( x );
</script>
```

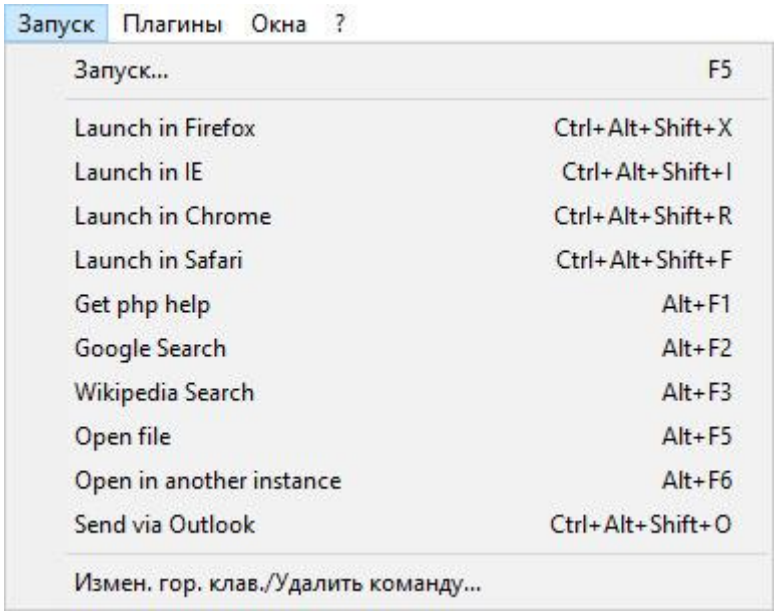
Выводы

- Дополнительная информация по функционалу JavaScript
- Дополнительные встроенные объекты

Благодарю за внимание

Добавление браузера (команды) в Notepad++

Источник: https://youtu.be/N_eVFp7_pc0

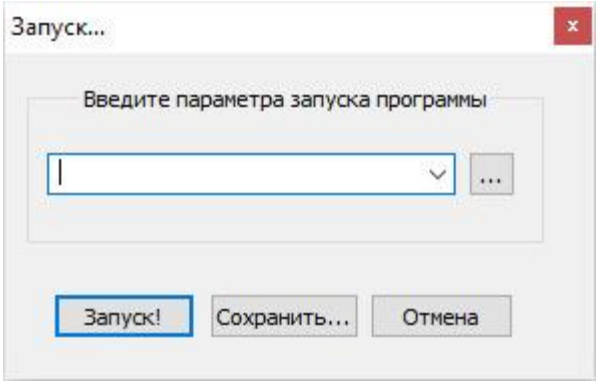


Хочу добавить в меню 'Запуск' текстового редактора Notepad++ браузер, например, Vivaldi

Нажимаю правой кнопкой на ярлык браузера Vivaldi на рабочем столе, что бы найти путь к исполняемому файлу.

Получаю, например,
"C:\Users\Valentin\AppData\Local\Vivaldi\Application\vivaldi.exe"

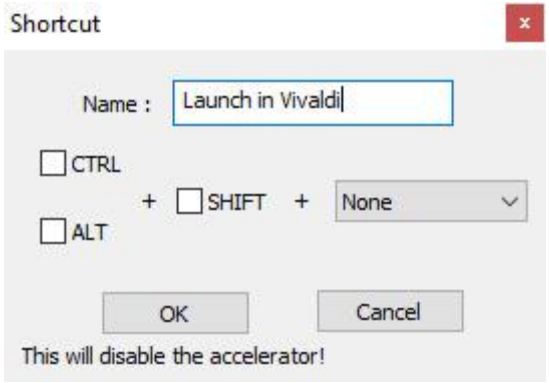
Вызываю окно 'Запуск...'



В окно вставляю путь к исполняемому файла в двойных кавычках, потом пробел, потом в двойных кавычках "$(FULL_CURRENT_PATH)".

Получается так:
"C:\Users\Valentin\AppData\Local\Vivaldi\Application\vivaldi.exe" "$(FULL_CUR-RENT_PATH)"

Нажимаю "Сохранить…" и ввожу название нового пункта меню в меню Запуск, который будет запускать браузер Vivaldi. Например, "Launch in Vivaldi".



Все, готово)