

# Лекция 3. JavaScript

# JavaScript

- Язык программирования JavaScript придуман для того, чтобы исполнять сценарии на HTML-страницах, на стороне клиента
- Нужен был, чтобы придать страницам интерактивность, возможность меняться в зависимости от действий пользователя
- Сейчас JavaScript вышел далеко за рамки браузеров – на нем пишется и серверная логика/десктопные приложения (**Node.js**), и мобильные приложения под все платформы (Titanium), и запросы к БД (**Mongo DB**) и т.д.
- Да и на HTML-страницах роль JavaScript заметно выросла – сейчас огромная часть логики веб-приложений перенесена на сторону клиента

# Недостатки JavaScript

- Как язык, JavaScript достаточно прост, но имеет серьёзные недостатки:
  1. **Язык не компилируемый (интерпретируемый).**  
То есть если вы опечатались, то никто об этом не скажет. Либо вы заметите сами, либо среда разработки подскажет, либо увидите ошибку только во время исполнения программы
  2. **Нестрогая типизация.** В JavaScript каждая переменная не имеет определенного типа и может в разные моменты времени хранить данные разных типов

# Недостатки JavaScript

## 3. Неявные преобразования типов.

В JS часто происходят неявные преобразования типов.

Например, `0 == ""` выдаст `true`

Это приводит к сложнообнаруживаемым ошибкам

- Чтобы устранить эти недостатки, создают компилируемые языки, результатом компиляции которых является JavaScript. Например, это **TypeScript** от Microsoft:
- <http://www.typescriptlang.org/>

# Синтаксис JavaScript

# Схожесть синтаксиса

- JavaScript имеет синтаксис языка C, но имеет и отличия как по синтаксису, так и по семантике (смыслу конструкций)
- Сходства:
  - Конструкции `if`, `if-else`, тернарный оператор, `switch` (но `switch` немного отличается)
  - Привычные циклы `while`, `do-while`, `for`
  - Вызов метода и обращение к полю через оператор точка
  - Такие же арифметические, логические операторы и операторы сравнения

# Отличия синтаксиса

- Из отличий синтаксиса:
  - Ключевое слово `var` для объявления переменных
  - Специальный цикл `for in` по полям объекта
  - Синтаксис объявления функций `function`
  - Специальный синтаксис для массивов и объектов
  - Операторы строгого равенства и неравенства `===` и `!==`
  - Др.

# Смысловые отличия языка

- Переменные могут хранить значение любого типа
- Значения `null`, `undefined` и `NaN`
- Ключевое слово `this`
- Область видимости переменных
- Глобальные переменные
- Замыкание
- Строгое и нестрогое равенство
- Неочевидное неявное приведение типов
- Нет целых чисел, только вещественные



# Отсутствие функции main

- В JS отсутствует функция main
- Скрипт просто обрабатывается браузером сверху вниз
- Если встретилось, например, объявление функции или переменной, то она объявляется
- А если встретилась исполняемая команда – вызов функции, присваивание или некоторое выражение, то оно сразу же исполняется
- ```
function f() {  
    alert("OK");  
}  
f();
```

**alert – стандартная функция,  
выводящая окно с сообщением**

# Объявление переменных

- При объявлении переменных не нужно указывать тип, нужно просто указать ключевое слово `var`
- `var x = 3;`
- Переменная может в разное время хранить значения разных типов
- `var x = 3;`  
`x = "Hello!";`
- Можно также объявить переменную без присваивания
- `var x;`  
`x = 3;`

# Глобальные переменные

- Можно объявлять переменные без слова `var`, но это считается очень плохим стилем.
- Без слова `var` объявляется глобальная переменная, которая видна всюду
- `x = 3; // до этого нигде не было var x`

# Привычные конструкции

- `if`, `if-else`, `for`, `while`, `do-while` ведут себя в точности так же, как в Java
- `switch` ведет себя близко к тому, как в Java, но есть отличия
- **Важный момент** – в качестве условий может использоваться любое значение, не только `boolean`
- ```
var x = 3;  
if (x) {  
    alert(1);  
}
```

В JS есть понятие истинности значения – некоторые значения считаются за `true` в условиях, а некоторые – за `false`

Например, число 0, `null` и некоторые другие значения считаются за `false` в условиях

# Истинность выражений

- **Важный момент** – в качестве условий может использоваться любое значение, не только `boolean`
- ```
var x = 3;  
if (x) {  
    alert(1);  
}
```
- Рекомендую никогда не пользоваться этой особенностью языка, а всё же явно указывать условие
- Это ведёт к труднообнаруживаемым ошибкам

# Примеры конструкций

- ```
if (x == null) {  
    alert(1);  
} else {  
    alert(2);  
}
```

В JS так же есть значение `null`, операторы инкремента и декремента, операции с присваиванием вроде `+=`, так же выглядят логические и арифметические операторы

- ```
while (y > 0) {  
    ++y;  
}
```

Комментарии в JS такие же, как в Java

- ```
for (var i = 0; i < a.length; ++i) {  
    // ...  
}
```

# Имена переменных

- В целом правила для имен переменных те же самые, что и в Java
- Имена регистрозависимы
- Но дополнительно, символ \$ также считается допустимым в имени переменной
- Этим часто пользуются разработчики библиотек, например, jQuery, и дают такое имя своей переменной: \$
- Либо начинают имя переменной с \$  
Например: \$score

# Точка с запятой

- В JS не обязательно завершать каждую команду точкой с запятой, тогда команды разделяются по переводу строк
- Но рекомендуется всегда использовать точку с запятой в конце команд, чтобы были ясны намерения программиста, и чтобы избежать ошибок
- Обычно, если не поставить точку с запятой, то интерпретатор ориентируется на переводы строк
- `var x = 3`  
`var y = 4` // тут интерпретатор все поймет правильно
- Точка с запятой нужна, если хотим иметь несколько команд в одной строке  
`var x = 3; var y = 4` // но это плохой стиль



# Точка с запятой

- Не все переводы строк понимаются интерпретатором как конец команды
- Если следующая строка может быть понята как продолжение текущей команды, то интерпретатор так и делает

- `var a`

`a`

`=`

`3`

`console.log(a)`

**`var a` – корректная инструкция объявления переменной. Следующая строка – `a`, не может продолжить эту инструкцию**

**После `a` идет `=` и `3`, они могут быть поняты как продолжение инструкции**

- Выполнится как:

`var a; a = 3; console.log(a);`

# Точка с запятой

- Еще пример:

```
var y = x + f
```

```
(a + b).toString()
```

- Круглые скобки могут быть поняты как попытка вызвать `f(a + b)`

- В итоге получится так:

```
var y = x + f(a + b).toString();
```

Уже видно, что лучше ставить ;

- Но есть 2 исключения

# Точка с запятой

- Но есть 2 исключения

1. Инструкции `return`, `break`, `continue`

- Если после `return` есть код, но он на следующей строке, то этот код не будет выполнен, произойдёт `return` без аргументов
- `return`  
`true`
- Будет понято как `return; true;`

2. Операторы `++` и `--`

`x`

`++`

`y`

будет понято как `x; ++y;` а не как `x++; y;`

Чтобы никогда не думать об этих вещах, всегда ставьте ;

И никогда не ставьте `enter` после `return`

# Типы данных в JS

- Типы данных:
  - Числа (нет разделения на целые и вещественные)
  - Строки (в JS это отдельный тип, а не объект)
  - `boolean`
  - Объекты
  - Массивы
  - Функции (да, тут это тип данных)
  - `undefined`
- Типы данных также делятся на ссылочные и `value`-типы

# Числа в JS

- `var x = 5;`  
`var y = 2;`  
`alert(5 / 2);`
- Чему будет равен результат?
- 2.5 – в JS нет понятия целых чисел
- Поэтому, чтобы получить целое число, нужно округлить результат вниз:
- `alert(Math.floor(5 / 2))`

**Math в JS отличается, но в целом он очень похож на Java**

# Бесконечности и значение NaN

- При операциях с числами никогда не происходят ошибки, но могут получаться специальные значения
- Например:
  - $1 / 0$     `Number.POSITIVE_INFINITY`
  - $-1 / 0$     `Number.NEGATIVE_INFINITY`
  - $1 / \text{"123"}$     `NaN` (Not-a-number, не число)
- Любая операция с NaN дает NaN
- Причем NaN не равен сам себе: `NaN == NaN` *// false*
- Чтобы проверить, что значение равно NaN, нужно пользоваться функцией `isNaN`:
- `isNaN(NaN)` *// true*

# Преобразование строки в число

- Можно воспользоваться приведением типов:
- `var str = "2";`  
`var number = Number(str); // 2`
- Если строка не является числом, то результат NaN
- Еще есть 2 функции: `parseInt` и `parseFloat`
- При использовании `parseInt`, можно передать ей 1 или 2 аргумента – строку и систему счисления
- Лучше всегда передавать второй аргумент, потому что `parseInt` не использует 10 как систему по умолчанию, а пытается угадать систему счисления:
- `parseInt("010") == 8`

# Строки в JS

- `var x = "Hello world";`  
`var y = 'Hello world';`
- Литералы строк заключаются в одинарные или двойные кавычки
- Разницы нет, но я рекомендую двойные, как во многих других языках
- Есть специальный синтаксис для многострочных строк:  
`var htmlString = "<div>\n This is a string.\n</div>";`

**В конце каждой строки ставится обратный слэш, чтобы указать на перевод строки**



# Обертки над примитивными типами

- В JS есть обертки над примитивными типами, но их использовать не рекомендуется:
  - `var b = new Boolean(true);`
- Что интересно, есть обертка над строкой
- `"123"` и `new String("123")` – не одно и то же в JS

# Как подключить скрипт к странице

- Есть 2 варианта:
  - Писать JS код внутри тега `script`, т.е. JS будет в теле страницы
  - Сделать отдельный подключаемый JS файл и подключать его при помощи тега `script`
- Второй вариант более гибкий, он позволяет:
  - Подключать скрипт ко многим страницам
  - Менять файл скрипта независимо от страницы

# Пишем скрипт в теге script

- `<script type="text/javascript">`  
    `var x = 3;`  
    `alert(x);`  
    `</script>`

Атрибут type не обязателен

- Тег `<script>` можно вставлять в любом месте страницы - как в `head`, так и в `body`
- Когда при разборе страницы браузер дойдет до тега `<script>`, то его содержимое сразу же исполняется
- Рекомендация – вставлять скрипты в самый конец `body`

# Подключаем js файл

- `<script src="script.js"></script>`  
`<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.2.0/jquery.min.js">`  
`</script>`
- В атрибуте `src` указываем путь к JS-скрипту
- Если файл лежит на том же сайте, что и веб-страница, то можно использовать относительные пути:
  - `script.js`
  - `../js/script.js`
- Путь может быть и адресом на любом сайте – скрипт загружается, а потом сразу же исполняется
- Если у тега `script` указан атрибут `src`, то код внутри тега `script` игнорируется

# Массивы

- В JS есть массивы, но по смыслу это списки
- Литералы массивов очень удобны
- `var list = [1, 2, 3, 4, 5];` // объявили массив и заполнили его
- `var emptyList = [];` // пустой массив
- Доступ по индексу (отсчитываются от 0):
- `var x = list[3];` // 4  
`list[1] = 5;` // [1, 5, 3, 4, 5]

# Нетипизированность массивов

- Массивы могут хранить значения разных типов одновременно
- `var list = [1, "Hi", null, { title: "123" }, [1, 2]];`
- `var length = list.length; // длина массива: 5`

# Работа с массивами

- Есть большое количество стандартных функций для работы с массивами:
- `var list = [1, 2, 3, 4, 5];`  
`list.push(6);`      `// добавление в конец массива`  
`list.push(7, 8);`      `// можно вставлять сколько угодно за раз`
- `// [1, 2, 3, 4, 5, 6, 7, 8]`
- `var list = [1, 2, 3, 4, 5];`  
`var el = list.shift();`      `// удаление первого элемента`  
                                 `// метод выдает удаленный элемент`  
`// [2, 3, 4, 5]`

# Работа с массивами

- `var list = [1, 2, 3, 4, 5];`  
`console.log(list.join(", "));`  
`// 1, 2, 3, 4, 5`

Формирует строку из элементов, соединяя их переданным разделителем

- `var list = [1, 2, 3, 4, 5];`  
`list.reverse();`  
`// разворачивает массив – [5, 4, 3, 2, 1]`

- `var list = [1, 2, 3, 4, 5];`  
`list.unshift(0);`      `// добавление в начало массива`  
`list.unshift(-2, -1);` `// можно вставлять сколько угодно за раз`  
`// [-2, -1, 0, 1, 2, 3, 4, 5]`



# Работа с массивами

- `var list1 = [1, 2, 3];`  
`var list2 = [4, 5];`  
`var list3 = list1.concat(list2);`  
`// создает новый массив, в котором сначала будут идти`  
`// элементы из list1, а дальше – из list2`  
`[1, 2, 3, 4, 5]`
- В аргументах в `concat` можно передать много списков
- `var list = [1, 2, 3, 4, 5];`  
`var list1 = list.slice(1, 3);`  
`// [2, 3] – подмассив от начального индекса до конечного`  
`var list2 = list.slice(3);`  
`// [4, 5] – подмассив от индекса до конца массива`

# Работа с массивами

- ```
var list = [3, 1, 2, 6, 5];  
list.sort(function(e1, e2) {  
    return e1 - e2;  
});  
// [1, 2, 3, 5, 6]
```
- Еще есть метод `splice`, который позволяет удалить из массива подмассив
- <http://javascript.ru/Array/splice>

# Стандарты JavaScript

- Язык JavaScript стандартизован, стандарт называется **ECMAScript** (иногда сокращают до **ES**)
- Старая версия, которая поддерживается везде – **ES3**, там много чего нет
- Версия **ES5** вышла в 2009 году, в ней много чего добавилось, эта версия поддерживается во всех современных браузерах
- Дальше вышел **ES6** (или **ES2015**) – в нем много фич, которые поддерживаются не всеми браузерами. И каждый год выходят новые версии с новыми фичами
- Поэтому пока используйте **ES5**
- <https://tproger.ru/translations/wtf-is-ecmascript/>

# Функции массивов в ES5

- В ES5 у массивов добавилось много полезных функций:
  - **arr.every(callback)** - проверка, что все элементы удовлетворяют предикату
  - **arr.some(callback)** - проверка, что хотя бы 1 элемент удовлетворяет предикату
  - **arr.filter(callback)** – создает массив с элементами, прошедшими проверку переданным предикатом
  - **arr.forEach(callback)** – выполняет по очереди функцию для каждого элемента. Нет возможности прервать цикл

# Функции массивов в ES5

- **arr.lastIndexOf(element)** – выдает индекс последнего вхождения элемента или -1
- **arr.map(callback)** – создает новый массив из результатов вызова callback для каждого элемента массива
- **arr.reduce(callback, initialValue)** – вызов функции для каждого элемента массива слева направо с накоплением результата
- **arr.reduceRight(callback, initialValue)** – вызов функции для каждого элемента массива справа налево с накоплением результата

# Операторы сравнения === и !==

- В JavaScript помимо привычных операторов == и != есть операторы === и !==
- Операторы === и !== используют строгое сравнение
- А операторы == и != используют нестрогое сравнение и пытаются выполнить приведение типов
- Например,  
2 === "2" // false  
2 == "2" // true
- То есть === и !== в JavaScript соответствуют == и != в Java
- Рекомендуется всегда использовать === и !==, чтобы избежать ошибок, связанным с неявным приведением

# Значения `null` и `undefined`

- В JavaScript есть аж два специальных значения
  - `null` обозначает пустую ссылку
  - `undefined` обозначает что-то несуществующее
- В чем разница:
  - `null` всегда появляется из-за нас — мы сами должны присвоить `null` какой-то переменной: `var x = null;`
  - `undefined` обозначает отсутствие чего-то:
    - Обращение к несуществующему полю объекта
    - Обращение к не переданному аргументу функции
    - Обращение к массиву по индексу, по которому ничего не лежит
    - Результат функции, которая ничего не выдает

# Значения null и undefined

- `var x = { a: 3 };`  
`var y = x.b; // undefined`
- Кстати, `null == undefined`, но `null !== undefined`



# Функции

- Функции в JS – это тип данных
- Функции бывают именованные и анонимные

- Именованная функция:

- `function f(a, b) {  
 return a + b;  
}`

- Анонимная функция:

- `var f = function(a, b) {  
 return a + b;  
};`

`// функция присвоена переменной`

Для функции не указывается тип результата и типы аргументов

В начале идет ключевое слово `function`, потом может идти имя функции, а потом – перечень аргументов в скобках, затем – тело функции

# Вложенные функции

- В JS можно объявлять функции внутри функций
- ```
function f1() {  
    var a = 0;  
  
    function f2(b) {  
        a += b;  
    }  
  
    f2(10);  
    return a;  
}
```
- Вложенность может быть любой. Вложенная функция видна везде в функции, внутри которой она объявлена
- Невложенная функция видна везде

# Место объявления функции

- Именованная функция доступна во всей области видимости, а не с места объявления. Это называется **поднятием**
- ```
var x = f();           // все хорошо  
function f() {  
    return 5;  
}
```
- Если функция невложенная, то она видна во всем скрипте, будто бы была объявлена в самом верху
- Если функция вложенная в другую, то она видна всем внутри функции, будто бы была объявлена в самом верху

# Место объявления функции

- Если функция присвоена переменной, то она видна только с момента присваивания
- `var x = f();` // упадет с ошибкой  
`var f = function() {`  
    `return 5;`  
`};`

# Область видимости внутри функций

- Переменные, объявленные внутри функции при помощи слова `var`, видны только внутри этой функции и во вложенных в нее функциях
- ```
function calculateTime(distance, speed) {  
    var result = distance / speed;  
    return result;  
}
```

# Область видимости внутри функций

- Область видимости в JS – не блочная, а функциональная
- То есть переменная, объявленная в любом месте функции, видна во всей функции
- ```
function f(x) {  
  if (x >= 0) {  
    var result = 1;  
  } else {  
    var result = -1;  
    // переменная объявлена дважды, плохо  
    // но JS такое позволяет и не падает  
  }  
  return result;  
}
```

# Область видимости внутри функций

- Переменные ведут себя так, будто они объявлены в самом начале функции. Но при этом присваиваются они только в местах присваиваний
- ```
function f(x) {  
    var result = 0;  
    if (x >= 0) {  
        var temp = 1;  
        result += temp;  
    }  
    return result;  
}
```
- ```
function f(x) {  
    var result = 0;  
    var temp;  
    if (x >= 0) {  
        temp = 1;  
        result += temp;  
    }  
    return result;  
}
```

# Аргументы функции

- Любую функцию можно вызвать с любым количеством аргументов, причем любых типов
- Неважно, сколько аргументов указано при объявлении функции

- ```
function f(a, b) {  
    // код  
}
```

Если передали меньше аргументов, то они будут undefined

- ```
f(1, 3);      // a = 1, b = 3  
f("1");      // a = "1", b = undefined  
f();          // a = undefined, b = undefined  
f(1, 2, 3);   // a = 1, b = 2
```



# Аргументы функции

- Допустим, в функцию передали больше аргументов, чем в ней объявлено. Как к ним обратиться?
- Внутри функций можно обращаться к особой переменной **arguments**, в которой хранится массив аргументов (на самом деле – объект, подобный массиву)
- ```
function f(a, b) {  
    for (var i = 0; i < arguments.length; ++i) {  
        console.log(arguments[i]);  
    }  
}
```
- Это также позволяет работать с неопределенным числом параметров

# Перегрузка функций

- В JS нельзя перегружать функции – нельзя иметь несколько функций с одинаковым именем в одной области видимости

# Функции в качестве аргументов

- Функции удобно и полезно использовать в качестве аргументов
- Например, напишем универсальную функцию для фильтрации списка объектов:
- ```
function filter(list, f) {  
    var result = [];  
    for (var i = 0; i < list.length; ++i) {  
        var el = list[i];  
        if (f(el)) {  
            result.push(el);  
        }  
    }  
    return result;  
}
```

**f – функция, которая  
выдает boolean: true если  
надо оставить элемент,  
false - иначе**

# Функции в качестве аргументов

- ```
function filter(list, f) {  
    var result = [];  
    for (var i = 0; i < list.length; ++i) {  
        var el = list[i];  
        if (f(el)) {  
            result.push(el);  
        }  
    }  
    return result;  
}
```
- ```
var list = [1, 2, 3, 4, 5];  
var result = filter(list, function(e) {  
    return e % 2 === 0;  
});
```

# Замыкания

- Смысл замыкания – функция может использовать и менять все переменные из того контекста, где она была объявлена

- Пример:

- ```
var x = 1;  
function f() {  
    ++x;  
}
```

`f();`

`f();`

`console.log(x);`    *// 3*

Замыкания широко  
используются

# Частая ошибка с замыканием

- Навешиваем обработчики клика на кнопки
- ```
for (var i = 0; i < buttons.length; ++i) {  
    buttons[i].onclick = function() {  
        alert(i);  
    };  
}
```
- Казалось бы, делаем, чтобы при нажатии на каждую кнопку выводилось свое число, но это не так
- Все замыкания будут ссылаться на одну и ту же переменную `i`
- А оно после цикла будет равно `buttons.length`

# Исправление ошибки

- ```
for (var i = 0; i < buttons.length; ++i) {  
    buttons[i].onclick = function(x) {  
        return function() {  
            alert(x);  
        };  
    }(i);  
}
```

# Объекты

- Объекты в JS – это просто коллекции пар ключ-значение (хэш-таблица)
- Ключ – это имя поля или метода, значение – это значение поля или функция-метод
- `var obj = {};`      *// пустой объект – без свойств и методов*
- `var person = {  
 firstName: "Ivan",  
 lastName: "Ivanov",  
 getFullName: function() {  
 return this.firstName + this.lastName;  
 }  
};`



# Объекты

- `var person = {  
 firstName: "Ivan",  
 lastName: "Ivanov",  
 getFullName: function() {  
 return this.firstName + this.lastName;  
 }  
};`
- Для объектов не обязательно создавать класс
- Можно просто пользоваться литералами объектов, как в примере выше
- Часто так и делают

# Добавление свойств на лету

- В отличие от Java, объектам можно добавлять новые свойства на лету

- ```
var x = {};  
x.name = "Pavel";  
x.getName = function() {  
    return this.name;  
};
```

По сути, объект является хэш-таблицей, ключ которой – всегда строка

- Можно и удалять свойства на лету:  

```
delete x.name;  
console.log(x.name); // undefined
```

# Обращение к свойствам

- Кроме точки, можно обращаться к свойствам при помощи []
- Эквивалентно:
  - `x.name = "123";`  
`x["name"] = "123";`
  - `var y = x["name"];`  
`var z = x.name;`
- Причем синтаксис со скобками более мощный – туда можно передавать и переменные:
- `var propertyName = "name";`  
`var z = x[propertyName];`

Заметим, что с квадратными скобками мы передаем именно строку

# Обращение к свойствам

- Синтаксис с квадратными скобками позволяет использовать в качестве имен любые строки:
- `var propertyName = "*";`  
`x[propertyName] = 123;`
- А если `x.*`, то будет ошибка при выполнении скрипта
- Кстати, в литералах объектов тоже можно явно указывать, что имена свойств – строки:
- `var x = {`  
    `"name": 123     // это эквивалентно просто name: 123`  
`};`

# Итерирование по свойствам

- Есть специальная версия цикла `for`, чтобы пройти по всем свойствам объекта:
- ```
var obj = {  
    name: "Ivan",  
    age: 13,  
    married: false  
};  
for (var propName in obj) {  
    console.log(obj[propName]);  
}
```

# Слово `this` для функций

- В JS ключевое слово `this` очень коварно и совсем не похоже на это же слово в Java
- Посмотрим, чему оно равно в функциях

- ```
var x = {  
  name: "Pavel",  
  getName: function() {  
    return this.name;  
  }  
};
```

```
console.log(x.getName()); // Pavel
```

```
// при вызове метода от объекта, он становится this
```

# Слово `this` для функций

- ```
function f() {  
    alert(this.surname);  
}  
var x = {  
    surname: "Ivanov",  
    sayName: function() {  
        console.log(this.surname);  
        f(); // выдаст undefined  
    }  
};  
x.sayName();
```
- Когда функцию вызывают не от объекта, то `this` в ней будет `window` – глобальный объект

# Вызов при помощи call и apply

- Любую функцию в JS можно вызвать с любым объектом в качестве `this`
- ```
var x = {  
  name: "Ivan",  
  sayName: function() {  
    alert(this.name);  
  }  
};  
var y = {  
  name: "Petr"  
};
```
- ```
x.sayName.call(y);
```

 или  

```
x.sayName.apply(y);
```

`// Petr`



# Call и apply

- Разница только в способе передачи аргументов
- Первым аргументом у обеих идет то, что будет использовано в качестве `this`
- `call` принимает бесконечное число аргументов через запятую
- `apply` требует массив аргументов
- ```
f.call(y, arg1, arg2, arg3);  
f.apply(y, [arg1, arg2, arg3]);
```

# Namespaces

- При помощи объектов удобно организовывать namespace'ы:

- ```
ETR = ETR || {};  
ETR.S7 = ETR.S7 || {};
```

```
ETR.S7.initPage = function() {  
    // код  
};  
ETR.S7.user = "Pavel";
```

**||** обладает интересным свойством – оно выдает первый объект, который конвертируется в true

За счет этого если ETR уже существует, то он не перезапишется. А если нет – то станет пустым объектом

- Выгода: пространства имен, как в Java и других языках  
И меньше глобальных переменных – только ETR

# Существование глобальной переменной

- Если проверить так, то программа упадет:

```
if (globalVar !== undefined) {  
    // ...  
}
```

- Используемое решение:

```
if (typeof globalVar !== "undefined") {  
    // работает  
}
```

- Оператор `typeof` выдает тип переменной в виде строки

# Самовывзывающаяся функция

- Чтобы не засорять глобальную область видимости, часто отдельный модуль делают самовывзывающейся функцией:

- `(function() {`  
    `var x = 3; // не будет видна снаружи, удобно`

```
ETR = ETR || {};
```

```
ETR.S7 = ETR.S7 || {};
```

```
ETR.S7.initPage = function() {
```

```
};
```

```
// изменения будут доступны снаружи
```

```
})();
```

# Что не упомянуто

- **Прототипы и наследование** – достаточно глубокая тема, ее следует изучить. Рекомендуется создавать объекты через конструкторы, а методы добавлять через прототипы, производительность будет выше
- **strict mode** – хороший стиль использовать его
- **console.log** и другие методы для работы с консолью – удобно для логирования и отладки
- **setTimeout, setInterval, clearTimeout, clearInterval** – выполнение кода с задержкой, либо периодически. Часто используют setTimeout с задержкой 0 для асинхронности

# Работа с DOM

# Понятие DOM

- Помним, что HTML документ представляет из себя дерево элементов
- Это дерево называют **DOM (Document Object Model)**
- Каждый элемент страницы соответствует некоторому объекту в JS, через который можно взаимодействовать с этим элементом

# Получение DOM элемента

- Получить коллекцию DOM элементов с указанным тегом:
  - `var paragraphs = document.getElementsByTagName("p");`
- Получить элемент с заданным id:
  - `var el = document.getElementById("my-id");`
- Получить первый элемент по селектору:
  - `var link = document.body.querySelector(".myClass > a");`
- Получить коллекцию элементов по селектору:
  - `var links = document.body.querySelectorAll(".myClass > a");`
- Все эти методы могут вызываться не обязательно от `document.body`, а от любого элемента
- Тогда поиск будет производиться внутри элемента



# Полезные свойства DOM элемента

- ```
<div class="my-div">  
    
  <span class="text">Текст</span>  
</div>
```
- `var myElement = document.getElementById("my-div");`
- `myElement.outerHTML` – это будет текст всей этой разметки
- `myElement.innerHTML` – внутреннее содержимое элемента, т.е. это только **img** и **span**
- Эти свойства можно не только читать, но и присваивать

# Работы с атрибутами

- У элемента есть методы:
  - **getAttribute(attrName)** – получение значение атрибута по имени
  - **setAttribute(attrName, value)** – добавление или изменение значения атрибута по имени
  - **removeAttribute(attrName)** – удаление атрибута по имени
  - **hasAttribute(attrName)** – выдает boolean – есть ли указанный атрибут

# Создание DOM элемента

- Создаем элемент h1:
  - `var header = document.createElement("h1");`
- Задаем текст:
  - `header.innerText = "Заголовок";`
- Вставляем элемент в конец body:
  - `document.body.appendChild(header);`
  - Естественно, можно вставлять не обязательно в body
- Есть метод `insertBefore` – вставить элемент перед указанным элементом

# Материалы для изучения

- <http://learn.javascript.ru>
- В первую очередь эти разделы:
  - <http://joxi.ru/DmBEeoDiwN49wr>
- Также есть старая версия сайта:
  - <http://javascript.ru>

# Материалы для изучения

- Книги:
- Флэнаган Д. JavaScript. Подробное руководство (6-е издание, 2012)
- Рекомендую приступать к книге потом
- Книга довольно скучна, но это полное описание языка, которое нужно знать



# Демонстрация

- Отладка скриптов в Chrome

# Задача «Квадратное уравнение»

- Решить задачу про квадратное уравнение на JS
- Сделайте поля ввода для коэффициентов, и выводите результат на страницу

# Задача «Квадратное уравнение»

- Решить задачу про квадратное уравнение на JS
- Сделайте поля ввода для коэффициентов, и выводите результат на страницу



# Задача «Работа с массивами»

- Создайте массив чисел
  - Отсортируйте его по убыванию
  - Получите подмассив из первых 5 элементов и подмассив из последних 5 элементов
  - Найдите сумму элементов массива, которые являются четными числами
- Создайте массив чисел от 1 до 100, в таком порядке
  - Получите список квадратов четных чисел из этого массива

# Задача «Работа с DOM»

- Создать простую программу TODO List
- Есть список записей (поначалу пустой), поле ввода и кнопка для добавления записи
- По нажатию на кнопку в конец списка должна добавиться запись с текстом из поля ввода. При этом поле ввода очищается
- Рядом с каждым элементом списка есть кнопка удаления