

План практической работы № 5

для преподавателя

Оглавление

Введение	1
Среда для работы	2
Как добавить сценарий Js на страницу?	3
Первая программа	3
Операторы	5
Комментарии	6
Обработка ошибок	7
Функции	8
Области видимости переменных	10
Формы	12
Циклы for, while	14
Концепция объектов	15
Строки	17
Экранирование	17
Числа	18
Массивы	19

Введение

JavaScript – Язык сценариев для придания интерактивности web-страницам.

Язык программирования JavaScript (JS) придает веб-страницам возможность реагировать на действия пользователя и превращать статичные страницы в динамические, так, чтобы страницы буквально "оживали" на глазах.

Программы на этом языке называются скриптами. В браузере они подключаются напрямую к HTML и, как только загружается страничка — тут же выполняются.

Важный момент: основная теоретическая часть по языку JavaScript (объявление переменных, типы данных, описание условного оператора, циклов, объектов и пр.) рассказаны на лекции, материалы к лекции высланы студентам.

Так как JavaScript является в настоящее время единственным языком сценариев, который поддерживают все основные браузеры Web (Internet Explorer, Firefox, Netscape, Safari, Opera, Camino и т.д.), то он используется очень широко.

Практическое занятие призвано рассмотреть несколько задач с применением теории.

На первой практике по JS предполагается изучение основ, подробнее работа с DOM будет рассмотрена во второй практике.

Для самостоятельного изучения синтаксиса языка рекомендованы следующие ресурсы:

<https://learn.javascript.ru/>

<https://www.w3schools.com/js/default.asp>

Среда для работы

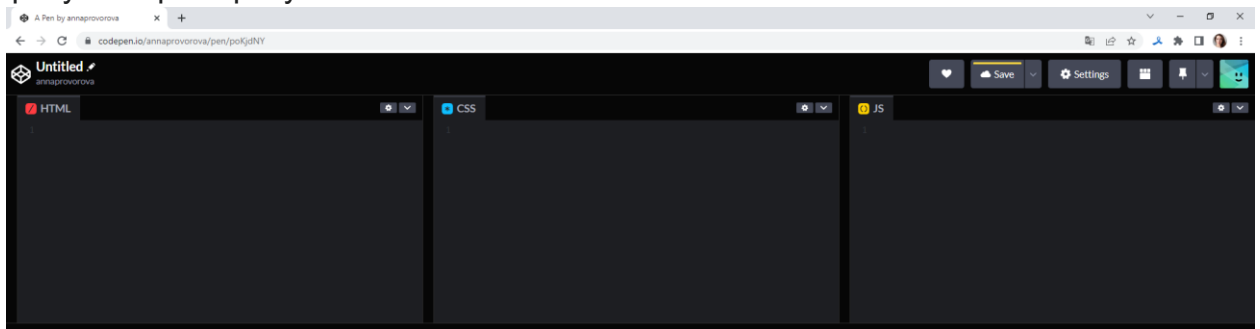
В этой практической работе для быстроты реализации использован сайт <https://codepen.io/>

CodePen — интернет-сообщество, созданное для тестирования и показа сниппетов на основе HTML, CSS и JavaScript.

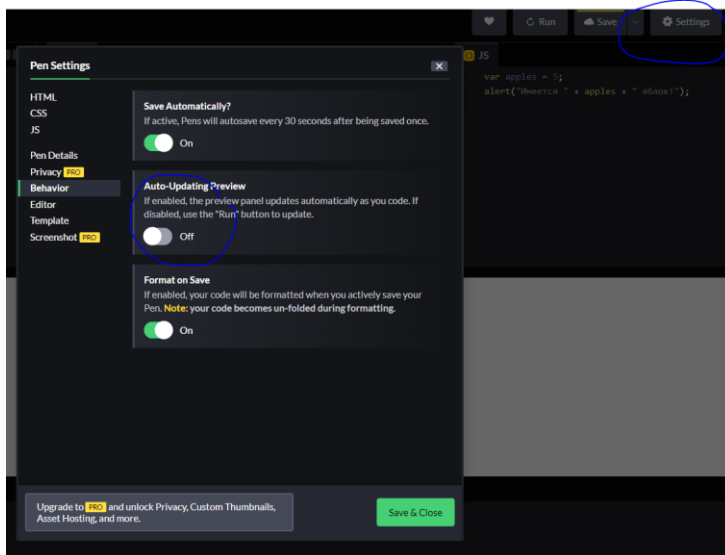
Работает как онлайн-редактор кода, где можно создать свой сниппет (pen) и посмотреть его.

CodePen является одним из самых больших интернет-сообществ для показа своих навыков программирования. Сейчас там насчитывается около 330 000 зарегистрированных пользователей, а в месяц сайт посещает более 16 миллионов человек.

С среде сразу рядом открыты три окна, для того, чтобы добавлять html, css и js и сразу смотреть результат.



Важный момент: по умолчанию кнопка «Save» подразумевает обновление и запуск содержимого документа. Если вы хотите, чтобы появилась отдельная кнопка «Run» для запуска проекта, зайдите в настройки проекта и отключите автообновление:



Как добавить сценарий Js на страницу?

Прежде всего, необходимо узнать, как добавить сценарий `JavaScript` на страницу `HTML`. Это можно сделать одним из двух способов: поместить теги `Script` на Web-странице и расположить код `JavaScript` внутри этих тегов, или поместить весь код `JavaScript` в отдельный файл и связаться с ним с помощью тега `Script`.

Любой из этих методов вполне допустим, но они имеют разное назначение. Если имеется небольшой код, который будет использоваться только на одной странице, то размещение его между тегами `Script` будет хорошим решением. Если, однако, имеется большой фрагмент кода, который будет использоваться на нескольких страницах, то, наверно, лучше поместить этот код `JavaScript` в отдельный файл и соединиться с ним. Это делается для того, чтобы не нужно было загружать этот код всякий раз при посещении различных страниц. Код загружается один раз, и браузер сохраняет его для последующего использования. Это похоже на то, как используются каскадные таблицы стилей (`CSS`).

Ниже приведены примеры двух способов подключения кода `JavaScript`:

```
<script type="text/javascript"></script>
```

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

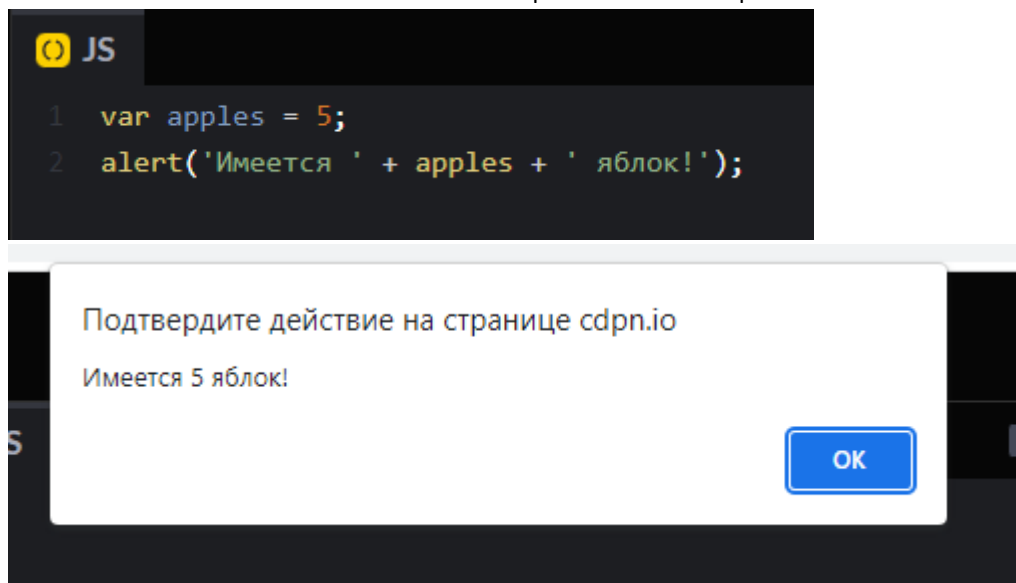
Первая программа

Допустим, перед нами стоит следующая задача:

Сообщить пользователю, что у нас есть определённое количество яблок, затем узнать у него, сколько яблок он желает съесть и уменьшить количество яблок на введённое число.

Давайте решим эту задачу.

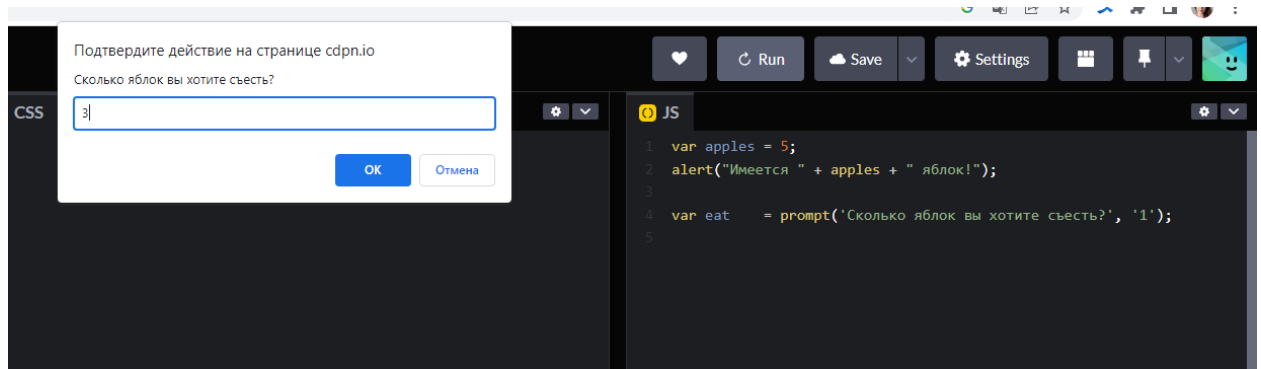
Сначала введём переменную, в которой хранится количество яблок и сообщим об этом количестве яблок пользователю с помощью всплывающего окна.



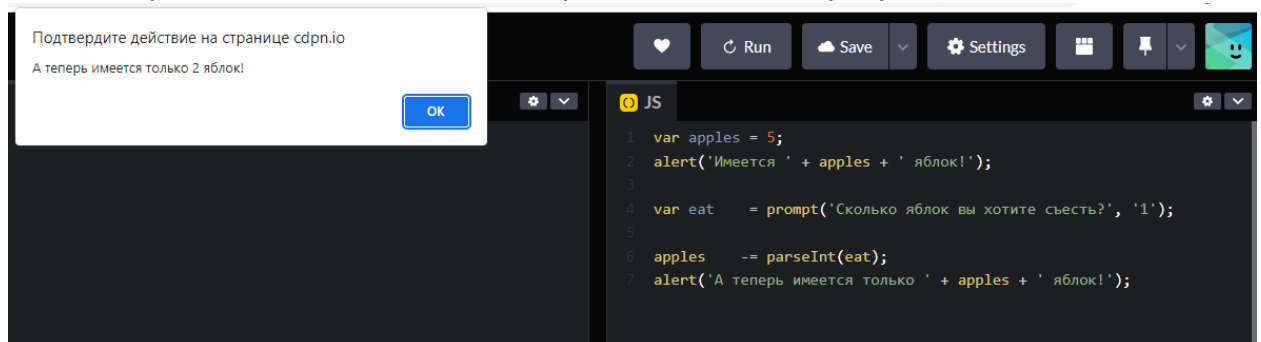
Что если мы хотим предложить пользователю съесть яблоко? Можно, например, спросить, сколько яблок он хотел бы съесть:

`prompt` является другой встроенной функцией, аналогичной `alert`. Однако вместо простого вывода информации она также получает ввод от пользователя. В

данном случае мы спрашиваем у пользователя, сколько яблок он хотел бы съесть. '1' в коде сообщает функции `prompt`, что значением по умолчанию для количества яблок будет 1, так как люди обычно едят только одно яблоко за раз. Однако пользователь может изменить это значение на любое другое. Когда пользователь щелкнет на кнопке `OK`, переменной `eat` будет задано значение этого ввода. Поэтому если пользователь введет 2, то `eat` будет равно 2.



Если пользователь съел 3 яблока, то останется 2, так? Поэтому выполним несколько простых математических операций и покажем результат.



Здесь мы видим два новых элемента. Прежде всего, обращение к функции `parseInt`, которая получает строку и возвращает число. Так как для выполнения математических операций требуются числа, то это гарантирует, что мы имеем число. Если пользователь введет в поле 2, то `parseInt` превратит это в число 2.

Затем, оператор `-=`, который означает вычитание из левой части оператора значения правой части. Поэтому значение переменной `eat` вычитается из переменной `apples`. Можно также записать эту строку следующим образом:

```
apples = apples - parseInt(eat);
```

Это означает в точности то же самое и может быть немного легче для понимания. Теперь, когда известно, сколько осталось яблок, мы еще раз сообщаем пользователю эту информацию.

Существуют другие операторы, подобные `-=`, которые делают похожие вещи. (`+=`, `-=`, `/=`, `*=`)

Ссылка на получившуюся программу <https://codepen.io/annaprovorova/pen/poKjdNY>

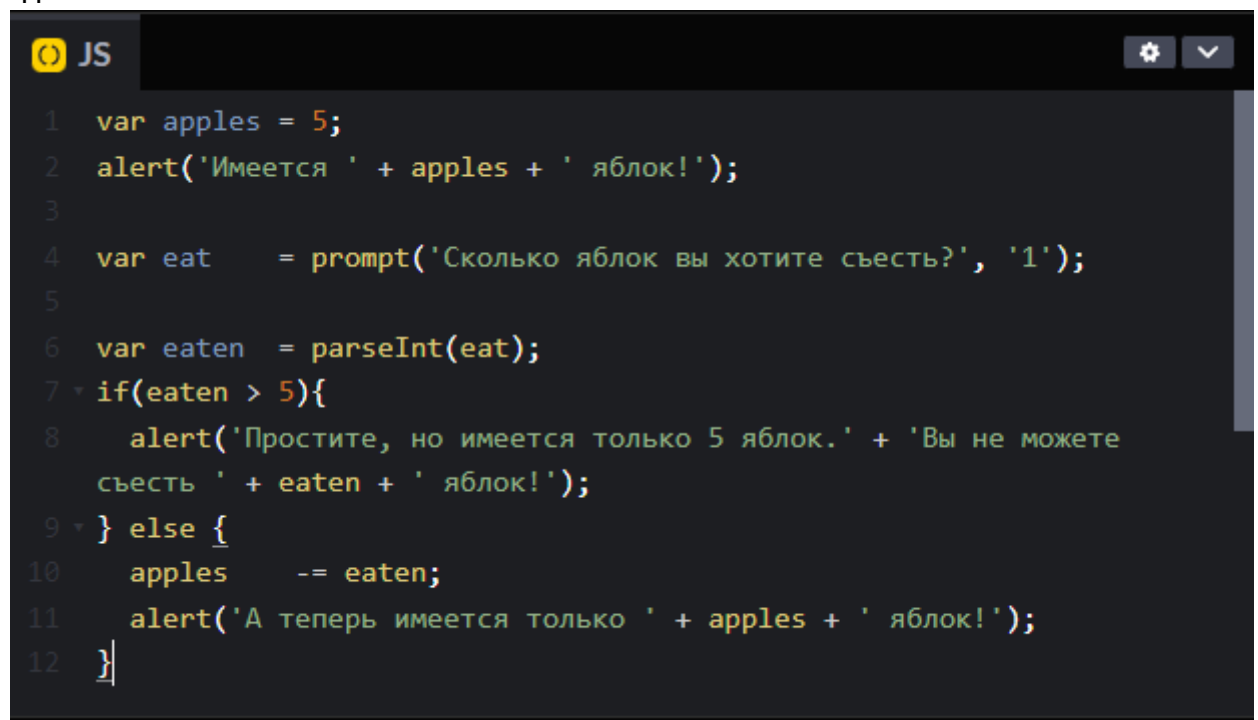
Кроме вывода результата работы программы во всплывающем окне, есть другой вариант работы, вывести результат в консоль. Для этого используется операция

```
console.log(...)  
console.log('Hello, World!')
```

Операторы

При тестировании сценария, написанного в предыдущей задаче, можно заметить, что результат, получаемый из `prompt`, требует некоторой проверки. Когда сценарий спрашивает, сколько яблок желает съесть пользователь, то пользователь может ввести число больше 5, меньше 0 или что-то, что вообще не является числом. В каждом из таких случаев желательно информировать пользователей, что введено недопустимое значение.

Так как в этом сценарии имеется только 5 яблок, то это максимальное количество яблок, которое может получить пользователь. Поэтому начнем с проверки, что введенное число не больше 5.

A screenshot of a code editor with a dark theme. The editor has a tab labeled 'JS' and icons for settings and a dropdown menu. The code is as follows:

```
1  var apples = 5;  
2  alert('Имеется ' + apples + ' яблок!');  
3  
4  var eat    = prompt('Сколько яблок вы хотите съесть?', '1');  
5  
6  var eaten  = parseInt(eat);  
7  if(eaten > 5){  
8      alert('Простите, но имеется только 5 яблок.' + 'Вы не можете  
    съесть ' + eaten + ' яблок!');  
9  } else {  
10     apples    -= eaten;  
11     alert('А теперь имеется только ' + apples + ' яблок!');  
12 }
```

Основными новыми понятиями здесь являются операторы `if` и `else`. Операторы `if` и `else` достаточно легко понять. Приведенный выше код дает возможность сказать: "Если пользователь выбрал для еды более 5 яблок, то сообщите ему, что такого количества яблок нет. Иначе позвольте ему съесть столько яблок, сколько он попросит."

Основной синтаксис оператора `if` / `else` следующий:

```
if(условие){  
    // код, который выполняется, когда справедливо условие if  
} else {  
    // код, который выполняется, когда условие if ложно  
}
```

Необходимо отметить открывающую и закрывающую скобки, `{` и `}`, в приведенном выше коде. Открывающая скобка сообщает коду, где начинается блок

кода, а закрывающая скобка указывает коду, где блок заканчивается. Поэтому все между { и } выполняется как часть оператора `if`. Необходимо отметить, что закрывающая скобка оператора `if` размещается непосредственно перед ключевым словом `else`. Оператор `else` имеет свой собственный набор скобок и свой собственный блок для выполнения.

Комментарии

Две косые черты `//` в приведенном примере говорят коду, что здесь находится комментарий. Комментарий является частью кода, который не выполняется.

`//` - однострочный комментарий

`/* ... */` - многострочный комментарий.

Пример:

```
// это однострочный комментарий
```

```
/*  
если требуется более длинный комментарий, то  
лучше использовать "блочный комментарий".
```

```
Этот комментарий является блочным комментарием,  
и полностью игнорируется при выполнении кода  
*/
```

Вернёмся к нашей программе. Пока мы обработали только случай, когда пользователь захотел яблок больше, чем есть. Но что, если пользователь введёт отрицательное количество яблок?

Или если введёт значение, которое не является числом?

Первый случай нетрудно обработать по аналогии. Для обработки второго случая нам потребуется ещё одна встроенная функция `isNaN`. При попытке преобразовать что-нибудь в число с помощью функции `parseInt`, возвращается значение `NaN`, если функция не может выполнить операцию. `NaN` означает `Not a Number` (Не число). С её помощью решим проблему:

```
JS
1 var apples = 5;
2 alert('Имеется ' + apples + ' яблок!');
3
4 var eat = prompt('Сколько яблок вы хотите съесть?', '1');
5
6 var eaten = parseInt(eat);
7 if(isNaN(eaten)){
8     alert('Вы должны ввести допустимое число яблок!');
9 } else if(eaten > apples){
10     alert('Простите, но имеется только ' + apples + ' яблок. Вы не можете съесть ' + eaten + ' яблок!');
11 } else if(eaten < 0){
12     alert('Простите, но вы не можете съесть отрицательное количество яблок!');
13 } else {
14     apples -= eaten;
15     alert('А теперь имеется только ' + apples + ' яблок!');
16 }
```

Ссылка на готовый скрипт <https://codepen.io/annaprovorova/pen/GRGpQmo>

Обработка ошибок

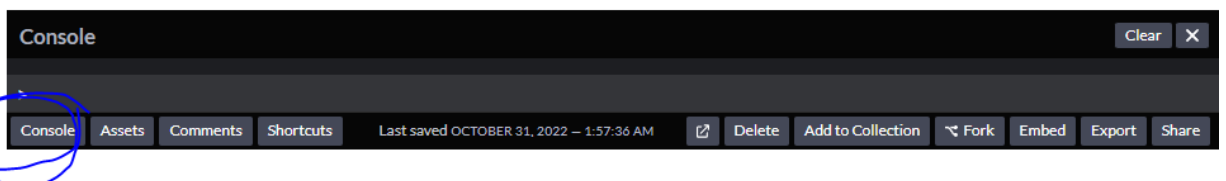
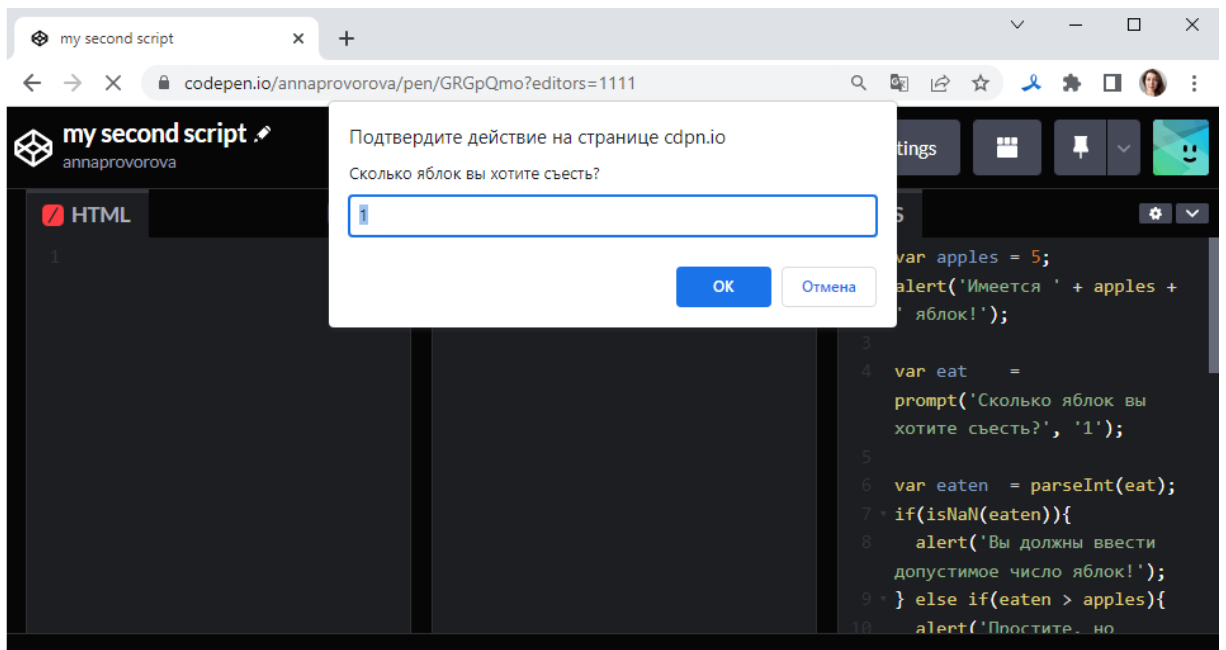
Уже на этом моменте хочется обратить внимание на обработку ошибок, которые могут возникать даже в маленьких скриптах.

Самыми классическими для новичков являются синтаксические ошибки. Помните важные правила:

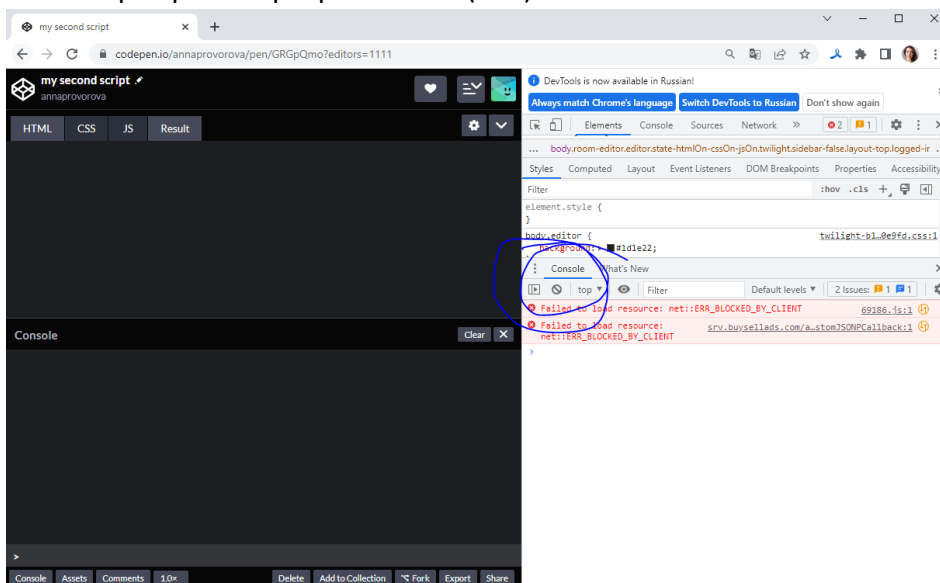
1. JS чувствителен к регистру. Переменные `apples` и `Apples` – разные!
2. Строчные значения нельзя разделять Enter. На скриншоте выше длинные предложения действительно переносятся со строки на строку, но это делает автоматический редактор. В противном же случае возникает синтаксическая ошибка.
3. В конце каждой строки обязательно нужно ставить “;”.

Если вы написали скрипт, а он по какой-то причине не выполняется, проверьте консоль, скорее всего в ней вы увидите сообщение об ошибке.

Чтобы проверить консоль в codepen нужно открыть её, кликнув по кнопке в левом нижнем углу окна:



Аналогично, если вы отлаживаете скрипт просто в окне браузера, вы можете проверить консоль через режим разработчика (F12):



Функции

Продолжим развивать нашу задачу:

Если пользователь ввел каким-либо образом недопустимое значение, то можно попросить его повторно ввести количество яблок, которое он хочет съесть. Одним из способов сделать это было бы копирование всего кода несколько раз. Однако

обычно это не самое лучшее решение. Что, если пользователь вводит недопустимое значение снова и снова? Можно продолжить копирование кода, но легко видеть, что это крайне неэффективно и очень трудно поддерживать код в рабочем состоянии.

В этом случае лучшим решением будет использование так называемой функции. Функция содержит код, который выполняет определенную задачу. Мы уже видели использование функций `alert`, `prompt`, `parseInt` и `isNaN`, которые встроены в язык `JavaScript`. Преимущество использования функций состоит в том, что можно выполнять один и тот же блок кода снова и снова, не копируя этот код. Для выполнения функции необходимо написать ее имя, за которым следуют скобки `()`, а все значения, передаваемые в функцию, записываются между скобками. Попробуем создать функцию самостоятельно.

Порядок объявления функции следующий:

- Имя функции.
- Список параметров (принимаемых функцией) заключённых в круглые скобки `()` и разделённых запятыми.
- Инструкции, которые будут выполнены после вызова функции, заключают в фигурные скобки `{ }`.

Перепишем наш код в функцию `eatApples()`

```
var apples = 5;

function eatApples(){
    alert('Имеется ' + apples + ' яблок!');

    var eat = prompt('Сколько яблок вы хотите съесть?', '1');

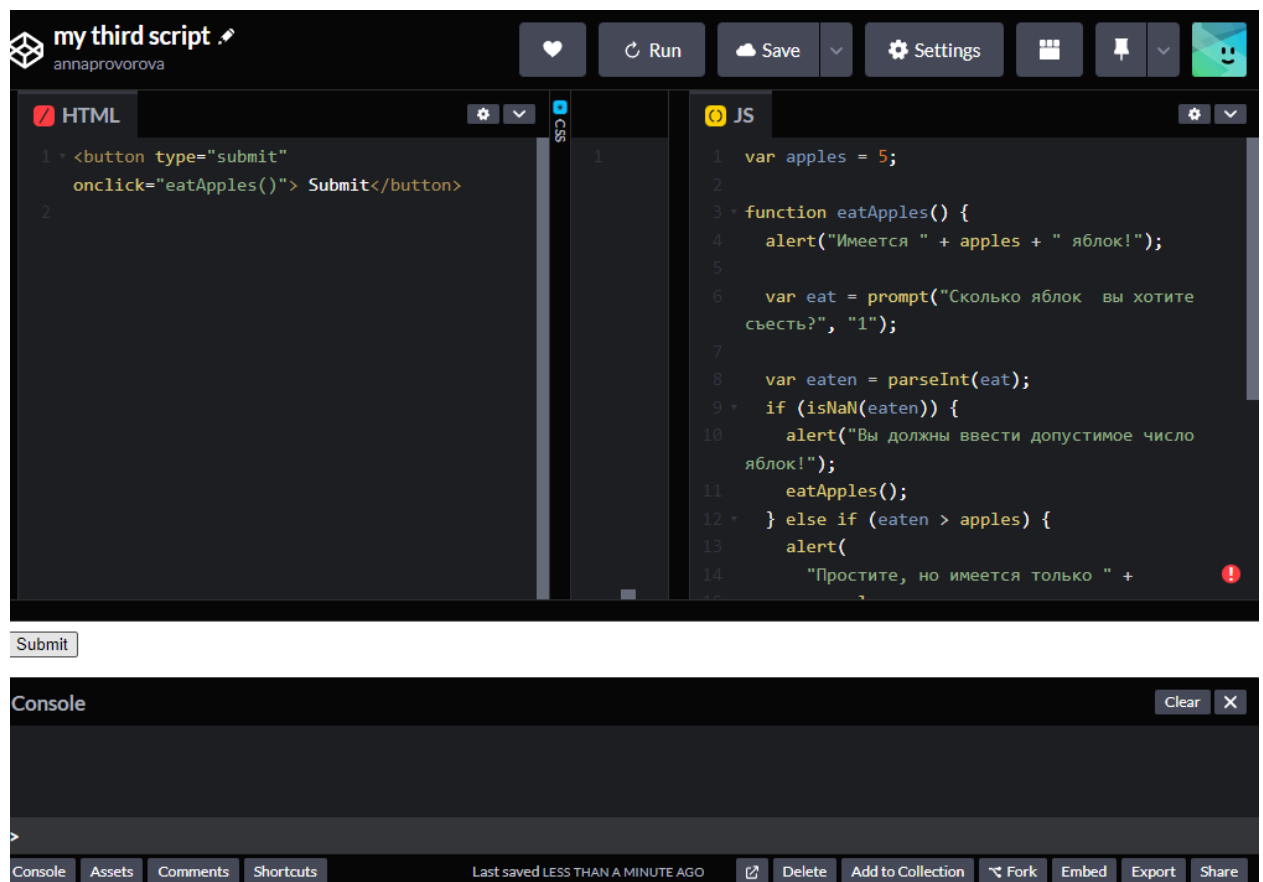
    var eaten = parseInt(eat);
    if(isNaN(eaten)){
        alert('Вы должны ввести допустимое число яблок!');
        eatApples();
    } else if(eaten > apples){
        alert('Простите, но имеется только ' + apples + ' яблок. Вы не можете съесть ' + eaten + ' яблок!');
        eatApples();
    } else if(eaten < 0){
        alert('Простите, но вы не можете съесть отрицательное количество яблок!');
        eatApples();
    } else {
        apples -= eaten;
        alert('А теперь имеется только ' + apples + ' яблок!');
        if(apples > 0){
            if(confirm('Не хотите съесть еще яблочко?')){
                eatApples();
            }
        } else {
            alert('Яблоко больше нет!');
        }
    }
}
```

Каждый раз, когда пользователь вводит неверное значение, снова вызывается функция `eatApples()`, чтобы пользователь мог ввести новое значение. Когда пользователь вводит допустимое значение, то он либо может еще есть яблоки, либо, если все яблоки закончились, он получит соответствующее сообщение. Здесь используется также одна новая функция, `confirm`. Функция `confirm` просто выводит пользователю приглашение `OK or Cancel` ("Да или Отмена"). Если пользователь нажмет кнопку "OK", то функция возвращает значение `true` (да). Если пользователь нажмет кнопку `Cancel` или просто закроет окно, то функция `confirm` возвращает значение `false` (нет). Поэтому в нашем примере функция `eatApples` вызывается снова только в том случае, когда пользователь щелкнет на кнопке `OK`.

Чтобы запустить функцию, её нужно вызвать. Для этого добавим вызов функции из HTML по кнопке. Подробнее о действиях по кнопке поговорим дальше.

ВАЖНО: в codepen не нужно писать `head` для страницы `html`, достаточно указывать содержимое `body`. Также не нужно отдельно прописывать подключение `js` файла, он подключается автоматически.

<https://codepen.io/annaprovorova/pen/GRGpQdj>



Области видимости переменных

Как можно видеть, в последнем примере переменная `apples` находится вне функции `eatApples`. Это делает переменную `apples` "глобальной переменной",

т.е. она будет доступна из любой функции. Переменная `eat`, с другой стороны, является локальной переменной и существует только внутри функции `eatApples`. Кроме того, каждый раз при вызове функции `eatApples` переменная `eat` не будет существовать, пока не будет снова определена функцией `prompt`.

Чтобы увидеть эту концепцию в действии, напишем две простые функции счета:

```
function counting1(){
  var count = 0;
  count++;

  alert(count);
}

var count = 0;
function counting2(){
  count++;

  alert(count);
}
```

Если запустить этот пример в браузере и щелкнуть на каждой кнопке несколько раз, то можно заметить, что `counting1` всегда выдает одно и то же значение, 1. `counting2`, с другой стороны, выдает увеличивающееся число. Почему?

Оператор `++` пока еще не встречался. `count++` просто увеличивает `count` на 1. Другими словами, это в точности то же самое, что написать `count += 1` или `count = count + 1`.

Аналогично оператор `--` вычитает 1 из переменной: `count--`.

Каждый раз, когда функция `counting1` выполняет `alert(count)`, она сообщает значение новой переменной `count`, которое только что было определено как `0+1`.

Теперь посмотрим на `counting2`. Можно видеть, что переменная `count` в этом случае находится вне функции. Даже до вызова этой функции значение `count` задано как 0. При вызове `counting2` прежде всего происходит увеличение переменной `count` на 1. Так как мы не восстанавливаем значение `count` в 0, как в случае `counting1`, то переменная `count` продолжает сохранять свое значение, и все происходит, как и предполагалось.

Соответственно, в функции `counting1` переменная `count` объявлена как локальная, а в функции `counting2` - как глобальная.

Формы

Мы знаем теперь, как проверять данные, но при создании кода `JavaScript` обычно требуется проверять не оставшееся количество воображаемых яблочек. Одной из наиболее общих областей применения `JavaScript` являются поля формы. Предположим, например, что имеется простая контактная форма. Иногда требуется убедиться, что пользователь ввел в форму свое имя или что он выбрал как минимум одну радио-кнопку для вопроса. Вот пример такой формы:

Имя:	<input type="text"/>	Ваш любимый цвет:	<input type="radio"/> Синий	<input type="radio"/> Желтый
Фамилия:	<input type="text"/>		<input type="radio"/> Красный	<input type="radio"/> Черный
Адрес Email:	<input type="text"/>		<input type="radio"/> Зеленый	<input type="radio"/> Другой
<input type="button" value="Отправить форму"/>		<input type="button" value="Очистить форму"/>	<input type="button" value="Зафиксировать форму на месте"/>	

Вот кусочек кода с этой формой:

<https://codepen.io/annaprovorova/pen/vYrNdwX>

Прежде всего необходимо узнать, как создать объект `JavaScript`, который ссылается на форму. Любую форму на странице можно указать с помощью конструкции `document.forms`. Если имеется форма с именем `firstform`, то к ней можно обратиться следующим образом: `document.forms.firstform`

На любые элементы внутри формы (поля ввода, поля выбора, флажки и т.д.) можно ссылаться с помощью конструкции `elements: document.forms.ИмяФормы.elements`. Если на форме имеется поле ввода с именем `firstname`, то значение этого поля можно вывести следующим образом:

```
alert('Имя: ' +  
document.forms.tutform.elements.firstname.value);
```

Когда форма посылается на сервер, Web-браузер ищет код `onsubmit`. Если этот код существует, то форма выполняет его перед отправкой:

```
<SCRIPT TYPE="text/javascript">  
function validateForm(){  
// код проверки формы находится здесь  
}  
</SCRIPT>  
  
<FORM ONSUBMIT="return validateForm();">  
<!-- элементы формы находятся здесь -->  
</FORM>
```

Выполним некоторые основные проверки. Распространенной задачей является проверка, что именно пользователь ввел в поле ввода. Например, надо проверить, что пользователь ввел свое имя.

Как видно из предыдущего фрагмента кода, свойство `".value"` объекта формы можно использовать для получения его значения. Это работает для объектов формы любого типа. Попробуем теперь проверить, что пользователь ввел на форме свои имя и фамилию:

```
JS
1 * function validateForm(){
2   var form_object = document.forms.firstform;
3   if(form_object.elements.firstname.value == ''){
4     alert('Вы должны ввести свое имя!');
5     return false;
6   } else if(form_object.elements.lastname.value == ''){
7     alert('Вы должны ввести свою фамилию!');
8     return false;
9   }
10  return true;
11 }
```

Важными моментами, которые необходимо отметить в этой функции, являются строки `return false;` и `return true;`. Если функция проверки возвращает значение `true`, то форма будет отправлена как обычно. Если, однако, функция вернет значение `false`, то форма отправлена не будет. Необходимо сообщить пользователю, почему форма не была отправлена, поэтому в функцию вставлены уведомления (`alert`).

Другим важным полем для проверки в демонстрационной форме будет набор радио-кнопок "Любимый цвет". Если щелкнуть несколько раз на этих кнопках, то можно видеть, что в данный момент времени может быть выбрана только одна из них. Но желательно знать, что пользователь выбрал хотя бы одну из этих кнопок.

Радио-кнопки и флажки на форме представляют специальную ситуацию. Часто имеется несколько радио-кнопок с одним и тем же именем, что почти всегда исключено для полей ввода, полей выбора и т.д.:

```
<input type="radio" name="color" value="blue">Синий
<input type="radio" name="color" value="red">Красный
<input type="radio" name="color" value="green">Зеленый
```

В связи с этим существует способ доступа ко всем радио-кнопкам с одним именем. Значение `document.forms.имяФормы.elements.имяРадиокнопок` будет содержать список со всеми радио-кнопками. Так как необходимо проверить, что хотя бы одна радио-кнопка отмечена, то потребуется просмотреть все эти радио-

кнопки. Если хотя бы одна из них отмечена, функция проверки должна вернуть `true`. Для такой проверки нам потребуется цикл `for`.

Циклы `for`, `while`

Самый распространённый цикл — цикл `for`. Синтаксис цикла:

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

Давайте разберёмся на нашем примере, как он работает:

```
function validateForm(){  
    var radios = document.forms.firstform.elements.color;  
    for(var i=0; i<radios.length; i++){  
        if(radios[i].checked) return true;  
    }  
    alert('Вы должны выбрать цвет!');  
    return false;  
}
```

Цикл `for` делает следующее:

1. задает `i` равным `0` ;
2. проверяет, что `i` меньше `radios.length`, которое равно `6` ;
3. если это справедливо (`true`), выполняет код в цикле `for` ;
4. после выполнения кода в цикле `for` добавляет `1` к переменной `i` ;
5. переходит к шагу 2, пока выполняется условие `i<radios.length`. Это условие не выполнится после шестого выполнения цикла, когда `i=6`.

На этом же примере рассмотрим другой тип цикла – `while`:

```
var i=0;  
while(i<radios.length){  
    if(radios[i].checked) return true;  
    i++;  
}  
alert('Вы должны выбрать цвет!');  
return false;
```

Принцип работы этих двух циклов очень похож. Обычно `for` применяется в случае, если количество перебираемых величин известно, а `while` если нам нужно повторять какое-то действие до определённого события, выполнения какого-то условия.

В этом кусочке кода осталось объяснить только ещё один момент:

`if(radios[i].checked)`. Переменная `radios` содержит массив радио-кнопок с именем `color`.

Подробнее о массивах мы поговорим чуть позже. Пока просто скажем, что по индексу `i` перебираются все радиобатоны подряд.

В форме осталось проверить еще ввод адреса e-mail. Это в действительности достаточно сложное для проверки поле, и правильная ее реализация выходит за рамки того, что изучается в этой лекции, но можно выполнить некоторую базовую проверку. Что нужно сделать? Мы знаем, что любой адрес e-mail должен содержать один и только один символ @. Он должен также содержать по крайней мере одну точку после символа @ (например ivanov@gmail.com)

```
function validateForm(){
    var email = document.forms.tutform.elements.email.value;
    if(email.indexOf('@')<0){
        alert('В адресе e-mail должен присутствовать символ @');
        return false;
    } else if(email.indexOf('@') != email.lastIndexOf('@')){
        alert('В адресе e-mail не может быть больше одного символа @');
        return false;
    } else if(email.indexOf('.')<0){
        alert('В адресе e-mail должна присутствовать хотя бы одна точка. ');
        return false;
    } else if(email.lastIndexOf('.')<email.indexOf('@')){
        alert('В адресе e-mail должна присутствовать хотя бы одна точка после символа @');
        return false;
    }
    return true;
}
```

Здесь имеются две новые сходные функции, которые требуют пояснения. Функция `indexOf` возвращает число, определяющее позицию одной строки в другой строке. `'abcdef'.indexOf('a')` вернет 0 (здесь 0 снова означает первую позицию). `'abcdef'.indexOf('cdef')` вернет 2, а `'abcdef'.indexOf('aaa')` вернет -1. -1 означает, что строка не найдена. Во многих случаях возвращается -1, когда функция не может получить результат.

Аналогично, функция `lastIndexOf` возвращает позицию последнего вхождения одной строки в другую. `'abcba'.lastIndexOf('a')` вернет 4, в то время как `'abcba'.indexOf('a')` вернет 0.

Вообще, это не идеальная функция для проверки поля ввода емейла, т.к. проверку пройдет адрес « @ . ».

В данной практике мы не будем останавливаться на доработке метода, студенты могут попробовать доделать его самостоятельно.

Концепция объектов

Что, если потребуется вызвать функцию, когда пользователь выполняет определенную задачу? В `JavaScript` можно соединить функцию практически с любым событием, которое может породить пользователь. Давайте посмотрим это в действии и напомним функцию, которая подсчитывает, сколько раз пользователь щелкнул на странице.

```
HTML
1 <p>Вы щелкнули на этой странице <input
  id="clicked" size="3" onfocus="this.blur();"
  value="0"> раз. </p>

JS
1 var clickCount = 0;
2 function documentClick(){
3   document.getElementById('clicked').value =
4     ++clickCount;
5 }
6 document.onclick = documentClick;
```

Вы щелкнули на этой странице раз.

Console Assets Comments Shortcuts Last saved LESS THAN A MINUTE AGO Delete Add to Collection Fork Embed Export Share

<https://codepen.io/annaprovorova/pen/eYKpMvg>

Вместо вызова функции `documentClick()` код содержит указание, что функция должна выполняться всякий раз, когда пользователь щелкает на документе. `document.onclick` связывает функцию с событием документа `onclick` ("при щелчке").

Наиболее растространёнными событиями являются:

`onclick`, `onload`, `onblur`, `onfocus`, `onchange`, `onmouseover`, `onmouseout` и `onmousemove`.

Функцию можно связать с событиями любого объекта, такого, как изображение или поле ввода, а не только документа. Например, события `onmouseover` и `onmouseout` используются обычно с изображениями для создания эффекта изменения.

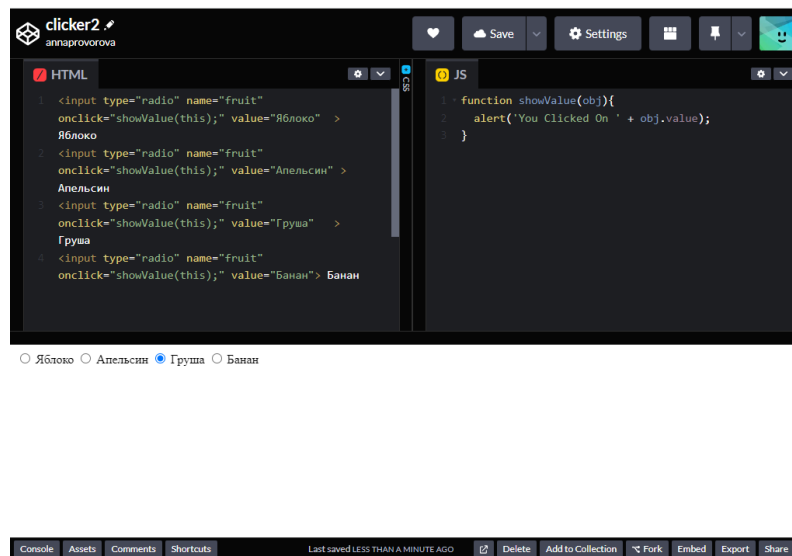
Поскольку это поле ввода используется как счетчик, то нежелательно, чтобы пользователи щелкали на нем и изменяли его значение. Здесь на помощь приходит другое связывание, `onfocus`, которое срабатывает, когда курсор перемещается на объект. Поэтому при щелчке на поле ввода или при перемещении на поле ввода с помощью клавиши `Tab` вызывается `onfocus`.

Событие `onfocus` имеет очень короткий код, но он также очень важен. В нем появляется ключевое слово `this`, которое важно понимать в `JavaScript`. Ключевое слово `this` указывает на тот объект, на котором выполняется код. В данном примере `this` указывает на поле ввода. Выражение `this.blur()` "размывает" поле ввода, другими словами, заставляет его терять фокус ввода. Так как это происходит, как

только пользователь активизирует поле ввода, то это делает "невозможным" изменение данных.

Если указатель `this` используется в функции, то он указывает на саму функцию. Если `this` используется в коде `JavaScript` вне функции, то он указывает на объект окна. Наиболее часто `this` используется для изменения свойства текущего объекта, как в примере выше, или для передачи текущего объекта функции.

Ещё один пример:



<https://codepen.io/annaprovorova/pen/PoaPRjR>

Можно видеть, что событие `onclick` для каждой из этих радио-кнопок одинаково. Однако, если щелкнуть на каждой из них, то будут получены разные сообщения. Это связано с использованием `this`. Так как `this` указывает на каждую отдельную радио-кнопку, то каждая радио-кнопка передается в функцию `showValue` по отдельности.

Строки

В `JavaScript` строка является любым фрагментом текста. Как и многие другие объекты в `JavaScript`, строки можно определять несколькими различными способами:

```
var myString = 'Hello, World!';  
var myString = new String('Hello, World!');
```

Второй метод применяется редко и только для гарантии, что получаемый объект является строкой. Например:

```
var n = 5;  
var s = new String(n*20);
```

Экранирование

Что, если в строке имеется апостроф? Следующий код работать не будет:

```
var n = 'The dog took it's bone outside';
```

Легко видеть, что апостроф в "it's" заканчивает строку. Поэтому мы получаем строку "The dog took it", за которой следует "s bone outside". Это продолжение само по себе не является допустимым кодом JavaScript (или правильным грамматически, если на то пошло), поэтому будет получена ошибка.

Здесь можно сделать две вещи. Так как строка определяется с помощью одиночных или двойных кавычек, то можно задать строку с помощью двойных кавычек. Другая возможность состоит в экранировании апострофа. Чтобы экранировать символ, необходимо просто подставить перед ним символ \. Символ \ в этом контексте сообщает JavaScript, что следующий символ необходимо воспринимать в точности так, как он есть, в номинальном значении, а не как специальный символ.

```
var n = "The dog took it's bone outside";  
var n = 'The dog took it\'s bone outside';
```

Если в строке должен присутствовать символ \, то он экранируется таким же образом: '\\'.
\\

Существует несколько других полезных функций для работы со строками, которые мы перечислим и кратко поясним.

- `charAt()` сообщает, какой символ находится в определенной позиции строки. Поэтому `'Test'.charAt(1) = 'e'`.
- `length` сообщает длину строки. `'Test'.length = 4`.
- `substring()` выдает строку между двумя индексами. `'Test'.substring(1, 2) = 'e'`.
- `substr()` аналогична `substring()`, только второе число является не индексом, а длиной возвращаемой строки. Если это число указывает на позицию за пределами строки, то `substr()` вернет существующую часть строки. `'Test'.substr(1, 2) = 'es'`;
- `toLowerCase()` и `toUpperCase()` делают то, что обозначают: преобразуют строку в нижний или верхний регистр символов соответственно. `'Test'.toUpperCase() = 'TEST'`;
- `indexOf` и `lastIndexOf` возвращают -1, если строка не существует, является очень полезным и позволяет использовать эти функции для достаточно распространенной задачи - проверки того, что одна строка существует внутри другой.
- `eval()` получает строку и выполняет строку, как если бы это был код JavaScript. то может быть очень полезно, так как позволяет создать содержащую код строку, а затем ее выполнить.

```
eval("alert('Hello, World!')");
```

Числа

Объект `Math` в JavaScript содержит функции, позволяющие сделать почти все, что можно сделать с числами помимо обычной арифметики. `Math.PI`, например, содержит просто число 3.141592653589793. В нем содержатся

тригонометрические функции (`sin`, `cos`, `tan`, и т.д.), функции для округления чисел (`Math.floor` возвращает число, округленное с недостатком, `Math.ceil` возвращает число, округленное с избытком, а `Math.round` округляет число "нормально") и многие другие. Существует очень много функций, объяснять которые здесь не имеет смысла. Их всегда можно найти в подходящем справочнике.

Очень распространённой операцией является преобразование числа в строку и строки в число:

```
var n = parseInt("3.14"); // 3
var n = parseFloat("3.14") // 3.14
```

Функция `parseInt` возвращает целое значение своего аргумента. Аргументы "3.14", "3", "3.00001" и "3.9999" превратятся в значение 3. Функция `parseFloat`, с другой стороны, возвращает также любое десятичное значение. Обе эти функции пытаются "очистить" данные перед возвращением числа. Например, `parseInt("3a")` вернет значение 3.

Существует также несколько методов, которые можно использовать, когда надо преобразовать число в строку:

```
var n = 5;

var m = n.toString();
var m = n+"";
var m = new String(n);
```

Массивы

Массив является по сути списком элементов. Каждый элемент массива может быть чем угодно, но обычно они связаны друг с другом.

Например, создадим список студентов:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
```

Создание пустого массива:

```
var students = [];
var students = new Array();
```

Чтобы обратиться к любому элементу массива, необходимо знать его индекс.

ВАЖНО: Массивы в `JavaScript` начинают индексацию с 0, а не с 1

Наиболее распространенной задачей при работе с массивами, помимо изменения его данных, является проверка его длины, обычно для того, чтобы можно было перебрать весь массив и выполнить некоторую задачу с каждым элементом.

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
var suffixes = ['1st', '2nd', '3rd', '4th'];

for(var i=0; i<students.length; i++){
    alert(suffixes[i]+' студент -- '+students[i]);
}
```

Важный момент, который необходимо знать о массивах, состоит в том, что каждый элемент массива может содержать любой произвольный объект. В этих примерах каждый элемент массива является строкой, но они могут быть также числами, объектами, функциями, даже другими массивами. Электронная таблица

(такая, как `Excel`) является хорошим примером массива, содержащего другие массивы. Прежде всего имеется массив столбцов. Каждый столбец будет в свою очередь содержать в себе массив строк. Этот массив создается точно таким же образом, как и массив `students`:

```
var spreadsheet = [  
  ['A1', 'B1', 'C1', 'D1'],  
  ['A2', 'B2', 'C2', 'D2'],  
  ['A3', 'B3', 'C3', 'D3'],  
  ['A4', 'B4', 'C4', 'D4']  
];
```

Чтобы добавить новый элемент к массиву нужно просто написать его значение:
`students.push('Steve');`

Чтобы удалить элемент из массива задействуется функция `splice`, которая позволяет добавить или удалить любое количество элементов массива, но в данный момент мы собираемся использовать ее для удаления одного студента, `Mike`, который переехал в другой город:

```
var students = ['Sam', 'Joe', 'Sue', 'Beth', 'Mike', 'Sarah', 'Steve'];  
students.splice(4, 1);
```

Чаще всего точно неизвестно, где в массиве находится элемент. К сожалению, единственным способом выяснить это является перебор всех элементов массива.

Еще две полезные функции для работы с массивами - `pop` и `shift`. Функция `pop` удаляет последний элемент из массива и возвращает его. Функция `shift` удаляет первый элемент из массива и возвращает его.

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];
```

```
while(students.length>0){  
  alert(students.pop());  
}
```

К сожалению, при этом массив был уничтожен: он теперь пуст. Иногда именно это и надо сделать. Если требуется просто очистить массив, то проще всего задать его длину (`length`) равной `0`:

```
students.length = 0
```

Теперь массив пуст. Даже если снова задать длину массива больше `0`, все данные в массиве уже будут уничтожены.

Все использованные до сих пор массивы называются "индексными массивами", так как каждый элемент массива имеет индекс, который необходимо использовать для доступа к элементу. Существуют также "ассоциативные массивы", в которых каждый элемент массива ассоциирован с именем в противоположность индексу:

```
var grades = [];  
grades['Sam'] = 90;  
grades['Joe'] = 85;  
grades['Sue'] = 94;  
grades['Beth'] = 82;
```

Ассоциативные массивы действуют немного иначе, чем индексные. Прежде всего, длина массива в этом примере будет равна 0. Как же узнать, какие элементы находятся в массиве? Единственный способ сделать это - использовать цикл "for-in":

```
for(student in grades){  
    alert("Оценка " + student + "будет: " + grades[student]);  
}
```

Синтаксис цикла for-in следующий: "for(item in object){". Цикл пройдет через все элементы в объекте, и элемент будет именем элемента. В данном случае элементом является "Sam", затем "Joe", "Sue" и "Beth".

Последнее замечание о массивах состоит в том, что в действительности можно объединять ассоциативные и индексные массивы, хотя это обычно не рекомендуется, так как может вызывать некоторые проблемы. При правильном использовании, однако, можно с успехом это применять.

```
var students = ['Sam', 'Joe', 'Sue', 'Beth'];  
  
students['Sam'] = 90;  
students['Joe'] = 85;  
students['Sue'] = 94;  
students['Beth'] = 82;  
  
alert('Всего имеется '+(students.length)+' студентов: '+students.join(', '));  
for(var i=0; i<students.length; i++){  
    alert("Оценка " +students[i]+"будет: "+students[students[i]]);  
}
```

На этом мы закончим нашу ознакомительную практику с JavaScript. На следующем практическом занятии будет рассмотрена Объектная модель документа, или DOM (Document Object Model). А пока, для закрепления полученных навыков, предлагаем перейти к Лабораторной работе.