Функции

Function Declaration

Локальные переменные
Переменные, объявленные внутри функции, видны только внутри этой функции.

Раздаточный материал № 1
```
function showMessage() {
  let message = "Привет, я JavaScript!"; // локальная переменная
  alert( message );
}
showMessage(); // Привет, я JavaScript!
alert( message ); //  будет ошибка, т.к. переменная видна только внутри функции
```

Внешние переменные
У функции есть доступ к внешним переменным.

Раздаточный материал №2
```
let userName = 'Вася';
function showMessage() {
  let message = 'Привет, ' + userName;
  alert(message);
}
showMessage(); // Привет, Вася
```

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Раздаточный материал №3
```
let userName = 'Вася';

function showMessage() {
  userName = "Петя"; // изменяем значение внешней переменной
  let message = 'Привет, ' + userName;
  alert(message);
}

alert( userName ); // Вася перед вызовом функции
showMessage();
alert( userName ); // Петя, значение внешней переменной было изменено функцией
```

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Раздаточный материал №4
```
let userName = 'Вася';

function showMessage() {
  let userName = "Петя"; // объявляем локальную переменную
```

```
  let message = 'Привет, ' + userName; // Петя
  alert(message);
}

// функция создаст и будет использовать свою собственную локальную переменную userName
showMessage();

alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную
```

Глобальные переменные

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в вышеприведённом коде – называются глобальными.

Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

Повторение параметров функции

Параметры

Значения по умолчанию

Если при вызове функции аргумент не был указан, то его значением становится undefined.

Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом:

```
showMessage("Аня");
```

Это не приведёт к ошибке. Такой вызов выведет "*Аня*: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined.

Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

Раздаточный материал №5

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}

showMessage("Аня"); // Аня: текст не добавлен
```

Теперь, если параметр text не указан, его значением будет "текст не добавлен"

В данном случае "текст не добавлен" это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра.

Раздаточный материал №6

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() выполнится только если не передан text
  // результатом будет значение text
}
```

Функциональные выражения (Function Expression)

Позволяет создавать новую функцию в середине любого выражения.

Раздаточный материал №7

```
let sayHi = function() {
  alert( "Привет" );
};
```

После ключевого слова function нет имени. Для Function Expression допускается его отсутствие.

Независимо от того, как создаётся функция – она является значением. Это значение можно вывести с помощью alert:

Раздаточный материал №8

```
function sayHi() {
  alert( "Привет" );
}

alert( sayHi ); // выведет код функции, т.к. нет круглых скобок
```

Можно скопировать функцию в другую переменную:

Раздаточный материал №9

```
function sayHi() {   // создаём
  alert( "Привет" );
}

let func = sayHi;    // копируем

func(); // Привет     // вызываем копию
sayHi(); // Привет    //    эта тоже все ещё работает
```

Функции-«колбэки»

Создадим функцию ask(question, yes, no) с тремя параметрами:

question - текст вопроса
yes- функция, которая будет вызываться, если ответ будет «yes»
no - функция, которая будет вызываться, если ответ будет «no»

Функция должна задать вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no()

Раздаточный материал №10

```
function callback1(guestion, yes, no) {
            if (confirm(guestion)) yes()
            else no();
        }

        function callbackYes(){
            alert("Вы согласились!");
        }

        function callbackNo(){
            alert("Вы отказались!");
        }

        callback1("Вы согласны?", callbackYes, callbackNo);
```

Функции callbackYes, callbackNo передаются в качестве аргументов callback1

Аргументы callbackYes, callbackNo функции callback1 называются **функциями-колбэками или просто колбэками.**

Ключевая идея в том, что передаётся функции и ожидается, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо.

Тот же пример с использованием анонимных функций:

Раздаточный материал №11

```
function callback2(guestion, yes, no){
            if (confirm(guestion)) yes();
            else no();
        }
        callback2("Наш сайт испольеут cookies. Нам нужно Ваше
согласие",
                function() {alert("Вы согласны!")},    //анонимная
функция
                function() {alert("Вы отказались!")}); //анонимная
функция
```

Здесь функции объявляются прямо внутри вызова callback2 (...). У них нет имён, поэтому они называются анонимными. Такие функции недоступны снаружи callback2, потому что они не присвоены переменным.

Функции – это значения. Они могут быть присвоены, скопированы или объявлены в любом месте кода.

Если функция объявлена как отдельная инструкция в основном потоке кода, то это “Function Declaration”.

Если функция была создана как часть выражения, то это “Function Expression”.

Function Declaration обрабатываются перед выполнением блока кода. Они видны во всём блоке.

Функции, объявленные при помощи Function Expression, создаются только когда поток выполнения достигает их.

В большинстве случаев, когда нам нужно объявить функцию, Function Declaration предпочтительнее, т.к функция будет видна до своего объявления в коде. Это даёт нам больше гибкости в организации кода, и, как правило, делает его более читабельным.

https://learn.javascript.ru/function-expressions

Практическое занятие «Работа с колбек-функциями»

Написать скрипт, содержащий колбек-функцию для вычисления значения математической функции. Полученные значения выводить в консоль:

Вариант 1 $y = \begin{cases} x^3, если X > 0 \\ 4x, если X \le 0 \end{cases}$	Вариант 2 $y = \begin{cases} x^2, если X < 2 \\ 2 + 3x, если X \ge 2 \end{cases}$	Вариант 3 $y = \begin{cases} x^4, если X > 1 \\ 4 - x, если X \le 1 \end{cases}$
Вариант 4 $y = \begin{cases} x^3, если X \le 4 \\ 5x, если X > 4 \end{cases}$	Вариант 5 $y = \begin{cases} x^2, если X > -5 \\ 6x, если X \le -5 \end{cases}$	Вариант 6 $y = \begin{cases} x, если X < 6 \\ 7x^2, если X \ge 6 \end{cases}$
Вариант 7 $y = \begin{cases} x^2, если X > -7 \\ 7 - x, если X \le -7 \end{cases}$	Вариант 8 $y = \begin{cases} x^2, если X \le -8 \\ 8x, если X > -8 \end{cases}$	Вариант 9 $y = \begin{cases} x^2 + 1, если X > 9 \\ 2x, если X \le 9 \end{cases}$
Вариант 10 $y = \begin{cases} x^2 + 1, если X > -9 \\ 4 + x, если X \le -9 \end{cases}$	Вариант 11 $y = \begin{cases} x - 1, если X < 12 \\ 5x - 2, если X \ge 12 \end{cases}$	Вариант 12 $y = \begin{cases} 2x^3, если X > 11 \\ 2x - 3, если X \le 11 \end{cases}$
Вариант 13 $y = \begin{cases} x, если X \le 6 \\ \frac{4}{x}(x+1), если X > 6 \end{cases}$	Вариант 14 $y = \begin{cases} 1 - x^2, если X > -15 \\ \frac{4x}{3}, если X \le -15 \end{cases}$	Вариант 15 $y = \begin{cases} \frac{x^3}{6}, если X > -5 \\ \sqrt{4x^2 + 1}, если X \le -5 \end{cases}$
Вариант 16 $y = \begin{cases} 2x^2 + 3, если X > -2 \\ 4x, если X \le -2 \end{cases}$	Вариант 17 $y = \begin{cases} 1 - x^2, если X > -3 \\ 6 + x, если X \le -3 \end{cases}$	Вариант 18 $y = \begin{cases} x - 1, если X > 1 \\ 2x^2, если X \le 1 \end{cases}$
Вариант 19 $y = \begin{cases} x^3 + 5, если X > 1 \\ 4 - x, если X \le 1 \end{cases}$	Вариант 20 $y = \begin{cases} 1 - x^2, если X < 0 \\ 3x, если X \ge 0 \end{cases}$	Вариант 21 $y = \begin{cases} x^3, если X > 2 \\ 4 - 3x, если X \le 2 \end{cases}$
Вариант 22 $y = \begin{cases} x^3 - 2, если X \le 3 \\ 4 - x, если X > 3 \end{cases}$	Вариант 23 $y = \begin{cases} 3x^2, если X > -1 \\ 6 - x, если X \le -1 \end{cases}$	Вариант 24 $y = \begin{cases} x - 3, если X < 2 \\ 2x^2 - 1, если X \ge 2 \end{cases}$

Стрелочные функции

Существует ещё один очень простой и лаконичный синтаксис для создания функций, который часто лучше, чем Function Expression. Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

let func = (arg1, arg2, ...argN) => expression;

Это создаёт функцию func, которая принимает аргументы arg1..argN, затем вычисляет expression в правой части с их использованием и возвращает результат.

Другими словами, это сокращённая версия:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Пример:

```
let sum = (a, b) => a + b;
       console.log(sum(1, 2));
```

```
/* Эта стрелочная функция представляет собой более короткую форму:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

Здесь (a, b) => a + b задаёт функцию, которая принимает два аргумента с именами a и b. И при выполнении она вычисляет выражение a + b и возвращает результат.

Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

```
let double = n => n * 2;
// примерно тоже что и: let double = function(n) { return n * 2 }

alert( double(3) ); // 6
```

Если аргументов нет, круглые скобки будут пустыми, но они должны присутствовать:

```
let sayHi = () => alert("Hello!");
sayHi();
```

Стрелочные функции можно использовать так же, как и Function Expression.

Например, для динамического создания функции:

```
let age = prompt("Сколько Вам лет?", 18);
```

```
let welcome = (age < 18) ?
  () => alert('Привет!') :
  () => alert("Здравствуйте!");

welcome();
```

Многострочные стрелочные функции

Иногда нам нужна более сложная функция, с несколькими выражениями и инструкциями. Это также возможно, нужно лишь заключить их в фигурные скобки. При этом важное отличие – в том, что в таких скобках для возврата значения нужно использовать return (как в обычных функциях).

```
let sum1 = (a, b) => {  // фигурная скобка, открывающая тело
многострочной  //функции
        let result = a + b;
        return result; // если мы используем фигурные скобки, то нам
нужно явно           //указать "return"
        };

        alert(sum1(1, 2) ); // 3
```

Практическое занятие «Работа со стрелочными функциями» №

Написать скрипт, в котором создать массив на 10 элементов, используя стрелочные функции выполнить указанные преобразования элементов. Использовать методы для работы с массивами map и filter. Полученные значения выводить в консоль: исходный массив, массив с результатами работы метода filter, массив с результатами работы метода map.

Вариант 1
- вывести элементы больше 0 и меньше 4
- возвести все элементы в квадрат

Вариант 2
- вывести элементы меньше 3 и больше -9
- отнять 5 от всех элементов

Вариант 3
- вывести элементы больше 3 и меньше 5
- разделить на 3 все элементы

Вариант 4
- вывести элементы равные 5 или 15
- прибавить 10 ко всем элементам

Вариант 5
- вывести элементы больше 6 и
- извлечь корень квадратный их каждого элемента

Вариант 6
- вывести элементы меньше 2
- возвести все элементы в квадрат

Вариант 7
- вывести четные элементы
- отнять 5 от всех элементов

Вариант 8
- вывести нечетны элементы
- разделить на 3 все элементы

Вариант 9
- вывести элементы кратные 3
- прибавить 10 ко всем элементам

Вариант 10
- вывести элементы некратные 3
- извлечь корень квадратный их каждого элемента

Вариант 11
- вывести элементы больше 6 и меньше 2
- прибавить 10 ко всем элементам

Вариант 12

- вывести четные элементы
- извлечь корень квадратный их каждого элемента

Вариант 13

- вывести нечетны элементы
- возвести все элементы в квадрат

Вариант 14

- вывести элементы кратные 3
- отнять 5 от всех элементов

Вариант 15

- вывести элементы некратные 3
- разделить на 3 все элементы

Вариант 16

- вывести элементы больше 0 и меньше 4
- отнять 5 от всех элементов

Вариант 17

- вывести элементы меньше 3 и больше -9
- разделить на 3 все элементы

Вариант 18

- вывести элементы больше 3 и меньше 5
- прибавить 10 ко всем элементам

Вариант 19

- вывести элементы равные 5 или 15
- извлечь корень квадратный из каждого элемента

Вариант 20

- вывести элементы кратные 3
- возвести все элементы в квадрат

Вариант 21

- вывести элементы некратные 3
- извлечь корень квадратный их каждого элемента

Вариант 22

- вывести элементы больше 0 и меньше 4
- возвести все элементы в квадрат

Вариант 23

- вывести элементы меньше 3 и больше -9
- отнять 5 от всех элементов

Вариант 24

- вывести элементы больше 3 и меньше 5
- разделить на 3 все элементы

Архитектура и принципы работы браузера

Высокоуровневая архитектура браузера состоит из:

- Пользовательского интерфейса.
- Движка браузера, который выступает посредником между пользовательским интерфейсом и движком рендеринга.
- Движком рендеринга, который отвечает за отображение запрошенного контента.
- Сети: для сетевых вызовов, таких как HTTP-запросы, с использованием различных реализаций для разных платформ за независимым от платформы интерфейсом.
- Бэкэнда пользовательского интерфейса. Этот бэкэнд предоставляет общий интерфейс, не зависящий от платформы.
- Интерпретатора JavaScript: он используется для анализа и выполнения кода JavaScript.
- Хранилища данных: браузеру может потребоваться локальное сохранение всех видов данных, например, файлов cookie. Браузеры также поддерживают такие механизмы хранения, как localStorage, IndexedDB, WebSQL и FileSystem.

В процессе загрузки веб-страницы от момента, когда пользователь ввел адрес до фактической загрузки страницы, находящейся по этому адресу, в браузере происходит множество процессов:

1. Навигация — первый шаг к загрузке страницы. Это процесс, когда пользователь запрашивает страницу: нажимает на ссылку, пишет адрес в адресной строке браузера, отправляет форму и т. д.

2. Поиск DNS (разрешение адреса)

3. Установление безопасного соединения

4. Получение

После того, как соединение установлено, браузер может получать ресурсы загружаемой страницы. Он начинается с получения документа разметки для страницы. Это делается с помощью протокола HTTP.

Для получения страницы выполняется запрос (идемпотентный) не изменяющий состояние сервера. Используется HTTP-запрос GET.

GET - запрашивает информацию с заданного сервера, используя унифицированный идентификатор ресурса (URI).

Как только веб-сервер получит запрос, он его проанализирует и попытается его выполнить. Тогда сервер выдаст HTTP-ответ, прикрепив соответствующие заголовки и содержимое запрошенного HTML-документа к телу этой структуры ответа.

5. Парсинг

Как только браузер получил ответ сервера, он начинает парсить полученную информацию. Этот процесс необходим для преобразования данных в деревья DOM и CSSOM, на основании которых рендерный движок затем создаст изображение сайта на экране.

> Объектная модель документа (DOM) - это внутреннее представление объектов, которые составляют структуру и содержимое документа разметки (в данном случае HTML), только что полученного браузером. Он представляет собой страницу, поэтому программы могут изменять структуру, стиль и содержимое документа.

Объектная модель CSS (CSSOM) - это набор API-интерфейсов, позволяющих манипулировать CSS из JavaScript. Вкратце: это тот же DOM, но для CSS, а не для HTML. Он позволяет пользователям динамически читать и изменять стиль CSS. Он представлен очень похоже на DOM в виде дерева и будет использоваться вместе с DOM для формирования дерева рендеринга, чтобы браузер мог начать процесс рендеринга.

Дальше браузер начинает строить дерево DOM. Дерево DOM описывает содержимое документа. В нём также указываются взаимосвязи и иерархия различных тегов. Теги, которые расположены внутри других тегов называются «дочерними» узлами. Чем больше узлов DOM, тем дольше происходит построение дерева.

Следующий этап в этом шаге – обработка CSS и построение дерева CSSOM. Браузер строит «узловую» модель дерева точно так же, как в случае с DOM: формирует родительские, дочерние, сиблинговые узлы.

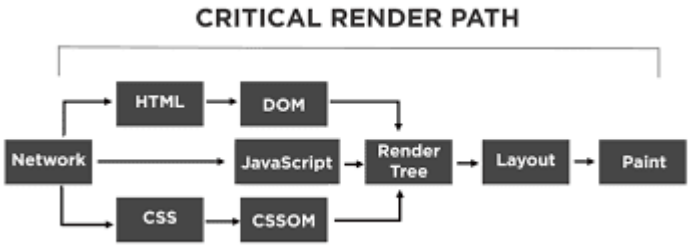
Когда оба дерева сформированы, их нужно объединить в единое дерево рендеринга. Такое дерево нужно для вычисления макета каждого видимого элемента. Оно выступает источником данных для отрисовки пикселей на экране.

Конечный результат - это визуализация, в которой есть как содержимое, так и информация о стиле всего видимого на экране.

6. Рендеринг

Теперь, когда информация проанализирована, браузер может начать её отображать. Для этого браузер теперь будет использовать дерево рендеринга для визуального представления документа. Этапы рендеринга включают в себя макет, раскраску и, в некоторых случаях, композицию.

Критический путь рендеринга.



Оптимизация критического пути рендеринга позволяет ускорить начало рендеринга: повышение скорости загрузки страницы за счёт определения приоритетов загружаемых ресурсов, контроля порядка их загрузки и уменьшения размеров файлов этих ресурсов.

Этапы рендеринга.

1. Макет - это первый этап рендеринга, на котором определяется геометрия и расположение узлов.
2. Рисование – следующий этап рендеринга. Браузер преобразует каждый блок, вычисленный на этапе макета, в фактические пиксели на экране. Рисование включает в себя визуализацию всех элементов на экране.
3. В ряде случаев при рендеринге страницы потребуется компоновка или выстраивание композиции. Когда разделы документа отрисовываются на разных слоях, перекрывая друг друга, наложение необходимо для обеспечения того, чтобы они отображались на экране в правильном порядке и содержимое отображалось правильно.

7. Загрузка JS, если ранее она была отложена.

https://freehost.com.ua/ukr/faq/articles/puteshestvie-veb-stranitsi-printsipi-raboti-veb---brauzera
https://habr.com/ru/company/kts/blog/669784/

BOM (BOM – Browser Object Model)

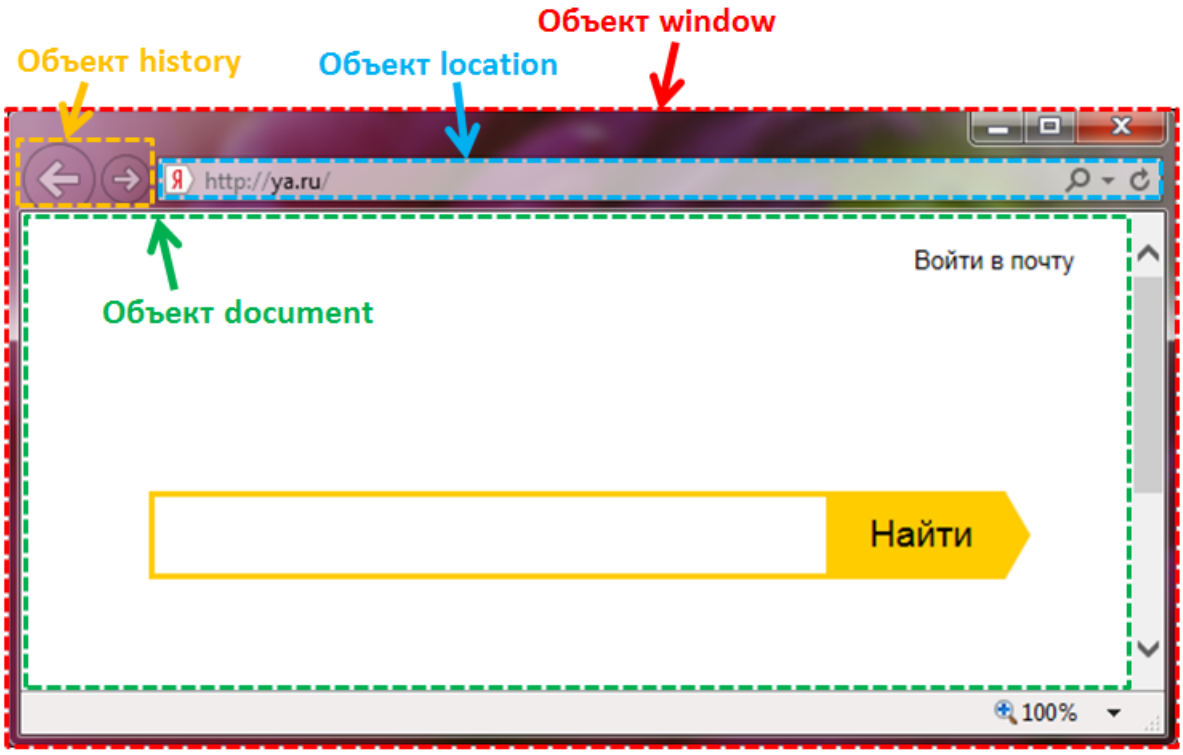
В сценариях JavaScript браузер веб-разработчику предоставляет множество "готовых" объектов, с помощью которых он может взаимодействовать с элементами веб-страницы и самим браузером. В совокупности все эти объекты составляют объектную модель браузера (BOM – Browser Object Model).

На самом верху этой модели находится глобальный объект window. Он представляет собой одно из окон или вкладку браузера с его панелями инструментов, меню, строкой состояния, HTML страницей и другими объектами. Доступ к этим различным объектам окна браузера осуществляется с помощью следующих основных объектов: navigator, history, location, screen, document и т.д. Так как данные объекты являются дочерними по отношению к объекту window, то обращение к ним происходит как к свойствам объекта window.

Например, для того чтобы обратиться к объекту screen, необходимо использовать следующую конструкцию: window.screen. Но если мы работаем с текущим окном, то "window." можно опустить. Например, вместо window.screen можно использовать просто screen.

Из всех этих объектов, наибольший интерес и значимость для разработчика представляет объект document, который является корнем объектной модели документа (DOM – Document Object Modal). Данная модель в отличие от объектной модели браузера стандартизована в спецификации и поддерживается всеми браузерами.

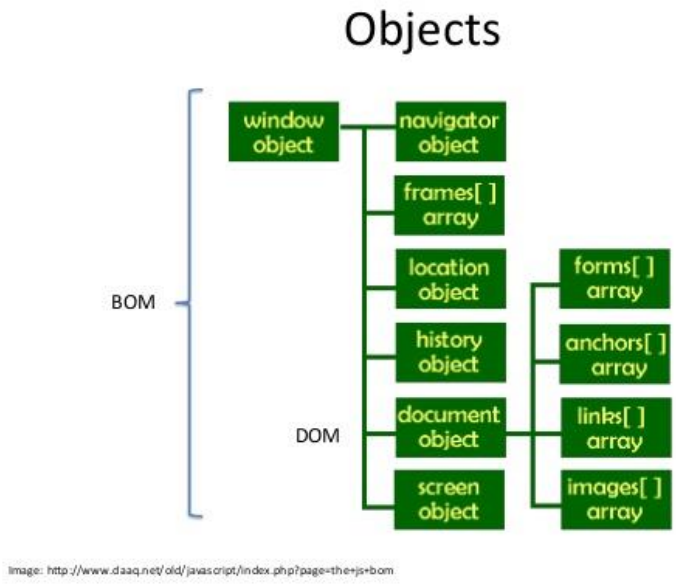
Объект document представляет собой HTML документ, загруженный в окно (вкладку) браузера. С помощью свойств и методов данного объекта Вы можете получить доступ к содержимому HTML-документа, а также изменить его содержимое, структуру и оформление.



Объектная модель браузера не стандартизована в спецификации, и поэтому её реализация может отличаться в разных браузерах.

Основная задача при создании динамических веб-страниц в основном сводится к отбору нужных объектов (элементов) и выполнению над ними различных действий. Результаты этих действий сразу отображаются на экране пользователя, а точнее в тех местах, за которые эти объекты отвечают.

Основные объекты Browser Object Model: window, navigator, history, location, screen, document.



Объект window

window – самый главный объект в браузере, который отвечает за одно из окон (вкладок) браузера. Он является корнем иерархии всех объектов доступных веб-разработчику в сценариях JavaScript. Объект window кроме глобальных объектов (document, screen, location, navigator и др.) имеет собственные свойства и методы, которые предназначены для:

- открытия нового окна (вкладки);
- закрытия окна (вкладки) с помощью метода close();
- распечатывания содержимого окна (вкладки);
- передачи фокуса окну или для его перемещения на задний план (за всеми окнами);
- управления положением и размерами окна, а также для осуществления прокручивания его содержимого;
- изменения содержимого статусной строки браузера;
- взаимодействия с пользователем посредством следующих окон: alert (для вывода сообщений), confirm (для вывода окна, в котором пользователю необходимо подтвердить или отменить действия), prompt (для получения данных от пользователя);
- выполнения определённых действий через определённые промежутки времени и др.

Если в браузере открыть несколько вкладок (окон), то браузером будет создано столько объектов window, сколько открыто этих вкладок (окон). Т.е. каждый раз открывая вкладку (окно), браузер создаёт новый объект window связанный с этой вкладкой (окном).

Примеры:
document.write("Строчка текста");
alert("Строчка текста");

Объект navigator

navigator – информационный объект с помощью которого Вы можете получить различные данные, содержащиеся в браузере:

- информацию о самом браузере в виде строки (User Agent);
- внутреннее "кодовое" и официальное имя браузера;
- версию и язык браузера;
- информацию о сетевом соединении и местоположении устройства пользователя;
- информацию об операционной системе и многое другое.

Объект history

history – объект, который позволяет получить историю переходов пользователя по ссылкам в пределах одного окна (вкладки) браузера. Данный объект отвечает за кнопки forward (вперёд) и back (назад). С помощью методов объекта history можно имитировать нажатие на эти кнопки, а также переходить на определённое количество ссылок в истории вперёд или назад. Кроме этого, с появлением HTML5 History API веб-разработчику стали доступны методы для добавления и изменения записей в истории, а также событие, с помощью которого Вы можете обрабатывать нажатие кнопок forward (вперёд) и back (назад).

Объект location

location – объект, который отвечает за адресную строку браузера. Данный объект содержит свойства и методы, которые позволяют: получить текущий адрес страницы браузера, перейти по указанному URL, перезагрузить страницу и т.п.

Объект screen

screen – объект, который предоставляет информацию об экране пользователя: разрешение экрана, максимальную ширину и высоту, которую может иметь окно браузера, глубина цвета и т.д.

Объект document

document – HTML документ, загруженный в окно (вкладку) браузера. Он является корневым узлом HTML документа и "владельцем" всех других узлов: элементов, текстовых узлов, атрибутов и комментариев. Объект document содержит свойства и методы для доступа ко всем узловым объектам. document как и другие объекты, является частью объекта window и, следовательно, он может быть доступен как window.document.

https://itchief.ru/javascript/bom

Объект window: открытие и закрытие окон

Методы объекта window:

open() - предназначен для открытия окон (вкладок);

close() - предназначен для закрытия окон. В основном используется для закрытия окон открытых методом open();

print() - предназначен для печати содержимого окна;

focus() - предназначен для передачи фокусу указанному окну;

blur() - предназначен для удаления фокуса с указанного окна.

В JavaScript открыть новое окно или вкладку из существующего документа можно с помощью метода «window.open».

Синтаксис:

```
window.open([url] [, windowName] [,windowFeature]);
```

Параметры:

url – адрес ресурса, который необходимо загрузить в это окно или вкладку (если в качестве url указать пустую строку, то туда будет загружена пустая страница «about:blank»);
windowName – имя окна;
windowFeature – необязательный параметр для настройки свойств окна (они указываются в формате «свойство=значение» через запятую и без пробелов).

Настройки окна windowFeature:

left и top – положение левого верхнего угла окна относительно экрана (значения этих свойств должны быть больше или равны 0);
height и width — размеры окна (его высота и ширина); основная масса браузеров имеет ограничения на минимальные значения этих свойств (в большинстве случаев – это не меньше 100);
menubar – во включённом состоянии отображает строку меню;
toolbar – включает показ кнопок панели инструментов («Назад», «Вперёд», «Обновить» «Остановить») и панель закладок (если она отображается в родительском окне);
location – определяет нужно ли показывать адресную строку;
resizable — свойство, которое позволяет включить (yes) или выключить (no) возможность изменения размеров окна;
scrollbars – предназначено для включения (yes) или выключения (no) полос прокрутки;
status – определяет нужно ли отображать строку состояния или нет.
Настройки menubar, toolbar, location, resizable, scrollbars, status является логическими, если их нужно включить, то устанавливаем значение true, в противном случае – false.

Упражнения:

1. Открыть пустую страницу about:blank в новом окне. Данное окно должно иметь ширину и высоту, равную 250px:

```
window.open("","","width=250,height=250");
```

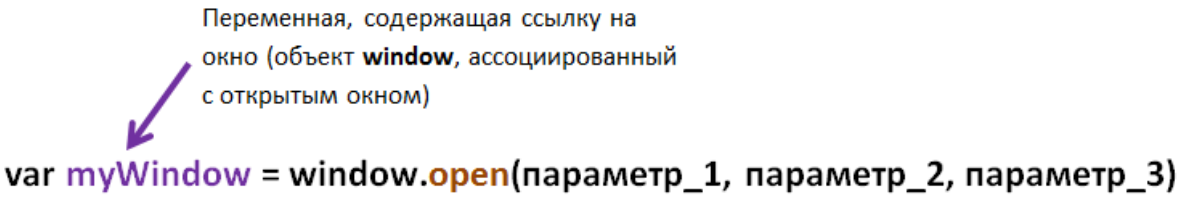
2. Открыть веб-страницу "http://itchief.ru/" в текущем окне:

```
window.open("http://itchief.ru/", "_self");
```

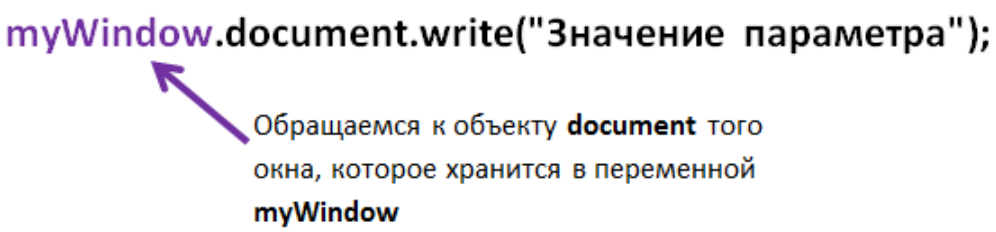
3. Открыть новое окно, имеющее определённые свойства (top=100, left=100,width=400,height=500,scrollbars=yes,resizabie=yes):

```
window.open("http://itchief.ru", "_blank", "top=100,left=100,width=400,height=500,scrollbars=yes,resizable=yes");
```

Метод open() позволяет не только открыть окно, но и получить ссылку на данное окно. Данная ссылка позволяет взаимодействовать с этим окном посредством вызова определённых свойств и методов. Т.е. мы можем с помощью JavaScript кода, расположенного в одном окне управлять другим окном.



Например, для того чтобы обратиться к объекту document открытого окна:



Открыть пустое новое окно и вывести в ней некоторый текст:

```
let myWindow = window.open("", "", "width=250, height=250");
myWindow.document.write("<p>Некоторый текст</p>");
```

Взаимодействовать Вы можете только с теми окнами, которые сами открыли, с другими окнами Вы работать не можете.

Метод close()

Он предназначен для закрытия окна. Данный метод не имеет параметров. Он обычно используется для закрытия окон созданных методом open(). В противном случае, когда Вы попытаетесь закрыть окно (вкладку), открытое самим пользователем (не из JavaScript), то браузер из-за соображений безопасности запросит у пользователя подтверждение на выполнение этого действия.

Метод focus()

Он предназначен для передачи фокусу указанному окну. Данный метод не имеет параметров.

Метод blur()

Он предназначен, чтобы убрать фокус с указанного окна, т.е. перемещает его на задний план. Данный метод не имеет параметров.

Раздаточный материал 1

```
//создать переменную, в которой будем хранить ссылку на объект window открытого
окна
        let myWindow;
        function myWindowOpen() {
        myWindow = window.open("http://www.yandex.ru", "myWindow", "top=200,
left=200,width=250, height=250");
        }
        function myWindowClose() {
        if (myWindow) {
            myWindow.close();
            myWindow = null;
```

```
        }
        }

        function myWindowFocus() {
              myWindow.focus();
            }

        function myWindowBlur() {
              myWindow.blur();
            }
        </script>

        <button onClick="myWindowOpen()">Открыть окно</button>
        <button onClick="myWindowClose()">Закрыть окно</button>
        <button onclick="myWindowFocus()">Передать фокус окну</button>
        <button onclick="myWindowBlur()">Переместить окно на задний план</button>
```

Метод print()

Он предназначен для печати содержимого окна. Данный метод не имеет параметров.

```
<script>
function myPrint() {
  window.print();
}
</script>

<button onclick="myPrint()">Печать страницы</button>
```

Свойства объекта window: name, opener, closed

Свойство name

Данное свойство очень часто используется для изменения внутреннего имени окна, после того как оно уже открыто. Кроме этого, свойство name может вернуть текущее значение внутреннего имени окна.

Внутреннее имя окна, это не строка, заключённая между открывающим и закрывающим тегом title - это имя окна которое предназначено для разработчика. Т.е. данное имя невидимо для пользователя.

Данное имя в основном используется в гиперссылках и формах для указания окна, в котором необходимо открыть страницу. Например, для указания внутреннего имени окна в гиперссылке используется атрибут target. Если элемент *a* имеет атрибут target="searchWindow", то при нажатии на данную ссылку браузер сначала пытается найти окно с таким внутренним именем (searchWindow), если окна с таким внутренним именем не существует, то он открывает новое окно и присваивает ему имя searchWindow. А если окно с таким именем существует, то новое окно не открывается, а перезагружается страница по указанной ссылке в этом окне. По умолчанию окна в браузере не имеют внутреннего имени.

Примеры:

1. откроем страницу "http://www.google.com/" в окне, имеющем имя myWindow:

```
<!-- Изменим имя текущего окна на "myWindow" -->
<script>
```

```
window.name = "myWindow";
</script>

<!-- Откроем URL, указанный в атрибуте href, в окне, имеющей имя "myWindow" -->
<a href="http://www.google.com/" target="myWindow">ссылка</a>
```

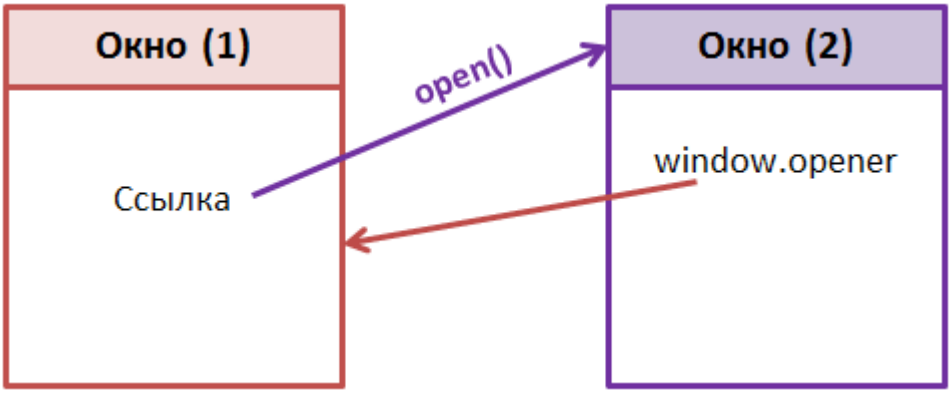
2. откроем окно с помощью метода open() и выведем в нём его имя:

```
<!-- Откроем окно и зададим ему внутреннее имя "myTest" -->
<script>
   let wnd = window.open("","myTest","top=100,left=100,width=250,height=350");
   //выведем с помощью метода write объекта document внутреннее имя окна
    wnd.document.write("<p>Это окно имеет имя:  " + wnd.name +".</p>");
</script>
```

Свойство opener

Данное свойство позволяет получить в окне, ссылку на исходное окно (объект window), т.е. на окно из которого было открыто данное окно.

Например, у Вас есть исходное окно (1), в котором Вы с помощью метода open() открываете другое окно (2). В этом окне (2) Вы можете с помощью свойства opener получить окно (1).



Раздаточный материал

```
<script>
    function openMyWindow()
    {
      // Открываем пустое окно, имеющее ширину и высоту, равные 200px
      var myWindow=window.open("","","top=100,left=100,width=200,height=200");
      // Выводим в открывшееся окно заголовок h1
      myWindow.document.write("<h1>Открытое окно (2)</h1>");
      // В окне myWindow (2), с помощью свойства opener получаем исходное окно
(1), в котором посредством метода write объекта document выводим заголовок h1
      myWindow.opener.document.write("<hl>Это исходное окно(1), с помощью
которого мы открыли окно (2)</h1>");
    }
    </script>

    <a href="javascript:openMyWindow()">Открыть окно</a>
```

Свойство closed

Свойство closed возвращает логическое значение, указывающее закрыто окно или нет.

Раздаточный материал

```
<button onclick="openWindow()">Открыть окно</button>
    <button onclick="closeWindow()">Закрыть окно</button>
    <button onclick="stateWindow()">Состояние окна</button>

    <script>
    let myTestWindow;
    function openWindow() {
      myTestWindow = window.open ("",""
,"left=200,top=250,width=250,height=250");
    }
    function closeWindow() {
      if (myTestWindow) {
        myTestWindow.close();
      }
    }
    function stateWindow() {
      if (!myTestWindow) {
        alert("Окно не открыто");
      }
      else {
        if (myTestWindow.closed)
          alert ("Окно закрыто");
        else
          alert ("Окно открыто");
      }
    }
    </script>
```

Размеры окна и позиция прокрутки в JavaScript

Как правило ширину и высоту браузера определяют во время создания разнообразных слайдеров галерей и прочих визуальной составляющей страницы. Например чтобы весь первый экран у нас занимало изображение или слайдер, мы определяем параметры окна и подставляю необходимые переменны в размеры объекта. Или когда необходимо, чтобы блок полностью закрывал все окно браузера, либо при создании резинового дизайна.

Свойства innerWidth и innerHeight

innerWidth – это свойство, которое позволяет получить внутреннюю ширину окна в пикселях (включая при этом в этот размер ширину вертикальной полосы прокрутки при её наличии).

innerHeight - предназначено соответственно для возвращения внутренней высоты окна в пикселях.

Раздаточный материал

```
<script>
// получим внутреннюю ширину окна в пикселях
```

```
const width = window.innerWidth;
console.log(width);
// получим внутреннюю высоту окна в пикселях
const height = window.innerHeight;
console.log(height);
//измените размер окна и перезапустите скрипт!!!
</script>
```

Свойства innerWidth и innerHeight доступны только для чтения и не имеют значения по умолчанию.

Если код выполняется в контексте объекта window, то его свойства и методы можно использовать без указания window:

// получим внутреннюю ширину окна в пикселях
const width = innerWidth;
// получим внутреннюю высоту окна в пикселях
const height = innerHeight;

Свойства «window.outerWidth» и «window.outerHeight» применяются довольно редко. Они предназначены для получения соответственно ширины и высоты всего окна браузера (включая границы самого окна, панель закладок и т.д.).

Раздаточный материал

```
// ширина всего окна браузера
const width1 = window.outerWidth;
// высота всего окна браузера
const height1 = window.outerHeight;
console.log(width1, height1);
//измените размер окна и перезапустите скрипт!!!
```

Свойства screenX и screenY

«window.screenX» и «window.screenY» предназначены для получения положения окна браузера (т.е. его x и y координат) относительно экрана.

Раздаточный материал

```
const screenX = window.screenX;
const screenY = window.screenY;
console.log(screenX, screenY);
//измените положение браузера на экране и перезапустите скрипт!!!
```

Свойства scrollX и scrollY (pageXOffset и pageYOffset)

scrollX и scrollY используются, когда нужно получить количество пикселей, на которые документ пролистали в данный момент соответственно по горизонтали и вертикали. Эти свойства доступны только для чтения. Возвращаемое ими значение является числом с плавающей точкой.

Раздаточный материал

```
const scrollX = window.scrollX;
const scrollY = window.scrollY;
console.log(scrollX, scrollY);
//что-бы оценить эффект - нужно добавить произвольный текст более чем на один
экран
```

Выполнить необходимое преобразование что бы получить целочисленное значение!!!

https://itchief.ru/javascript/window-sizes-and-scrolling