МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
для проведения практических занятий
по дисциплине «**ОСНОВЫ ВЕБ-ПРОГРАММИРОВАНИЯ**»
для студентов специальности 10.05.01 Компьютерная безопасность
специализация «Информационно-аналитическая и техническая экспертиза компьютерных систем»

Ставрополь
2026

Содержание

Занятие № 1 Первая программа на JavaScript

Для разработки на JavaScript нам потребуется прежде всего любой текстовый редактор для написания кода на данном языке программирования.

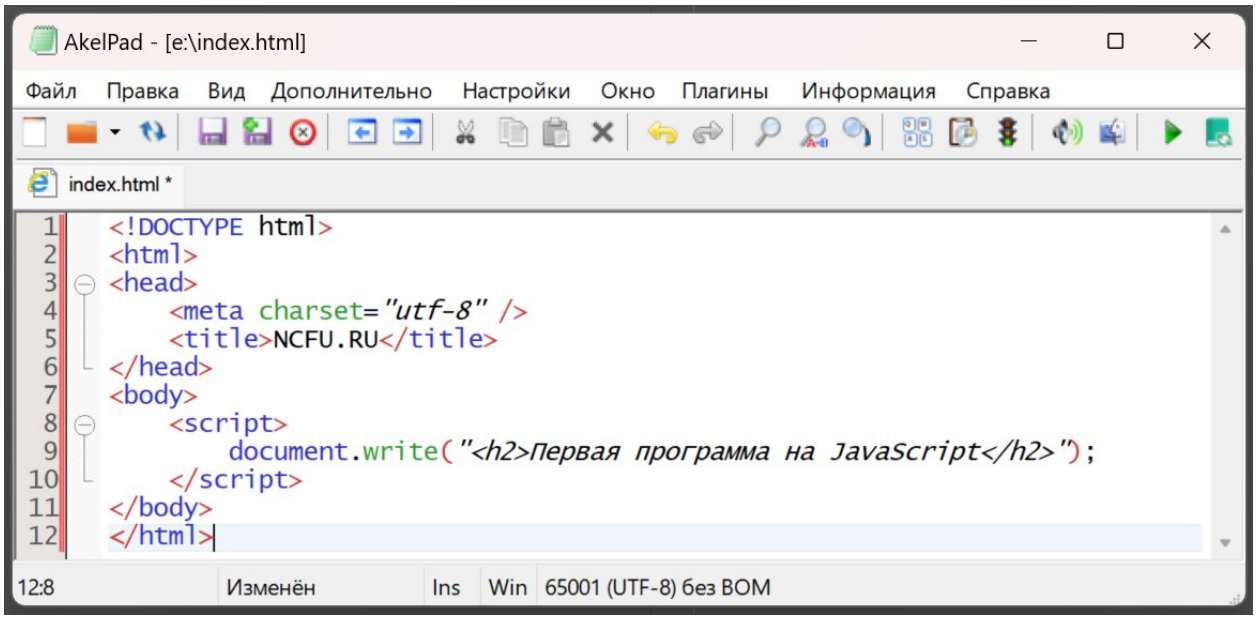
Для проверки выполнения программы нам потребуется также интерпретатор JavaScript, который можем выполнять программы JavaScript. В качестве интерпретатора мы будем использовать стандартный веб-браузер. Можно использовать последние версии любых распространенных браузеров - Google Chrome, Microsoft Edge, Mozilla Firefox, Opera, Safari.

Также существуют различные среды разработки, которые поддерживают JavaScript и облегчают разработку на этом языке, например, Visual Studio, WebStorm, Netbeans и так далее, которые могут упростить какие-то моменты создания программ на JavaScript. При желании можно использовать также эти среды разработки.

Первая программа на JavaScript

определим для нашего приложения какой-нибудь каталог. Например, создадим на диске C папку **app**. В этой папке создадим файл под названием **index.html**. То есть данный файл будет представлять веб-страницу с кодом HTML.

Откроем этот файл в текстовом редакторе и определим в файле следующий код:



Здесь мы определяем стандартные элементы html. В элементе **head** определяется кодировка utf-8 и заголовок (элемент title). В элементе **body** определяется тело веб-страницы, которое в данном случае состоит только из одного элемента **<script>**

Подключение кода javascript на html-страницу осуществляется с помощью тега **<script>**. Данный тег следует размещать либо в заголовке (между тегами <head> и </head>), либо в теле веб-странице (между тегами <body> и </body>). Нередко подключение скриптов происходит перед закрывающим тегом </body> для оптимизации загрузки веб-страницы.

Раньше надо было в теге <script> указывать тип скрипта, так как данный тег может использоваться не только для подключения инструкций javascript, но и для других целей. Так, даже сейчас вы можете встретить на некоторых веб-страницах такое определение элемента script:

<script type="text/javascript">

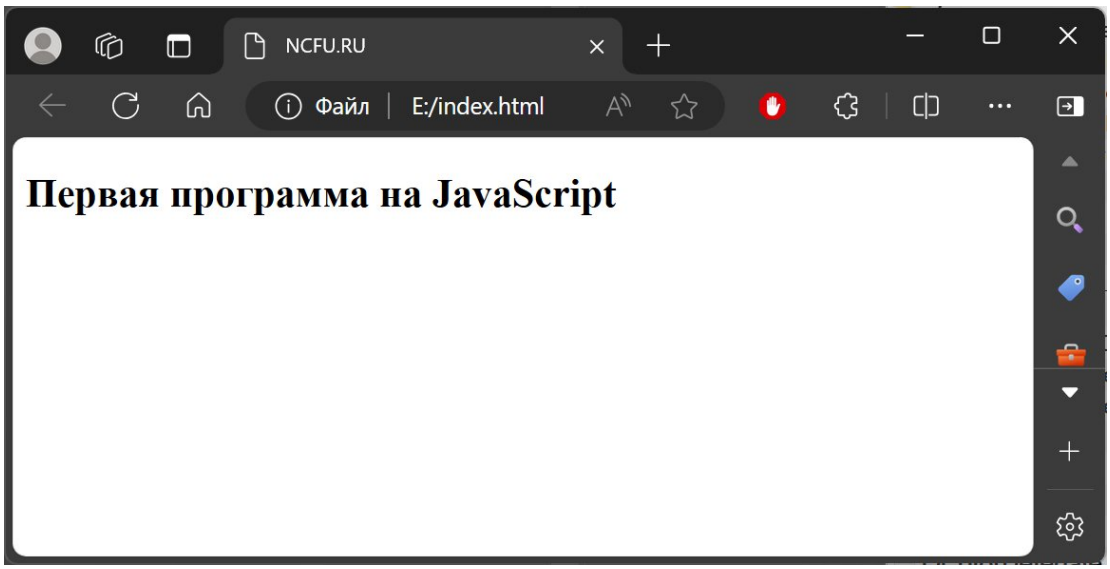
Но в настоящее время предпочтительнее опускать атрибут type, так как браузеры по умолчанию считают, что элемент script содержит инструкции javascript.

Используемый нами код javascript содержит одно выражение:

```
document.write("<h2>Первая программа на JavaScript</h2>");
```

Код javascript может содержать множество инструкций и каждая инструкция завершается точкой с запятой. Наша инструкция вызывает метод **document.write()**, который выводит на веб-страницу некоторое содержимое, в данном случае это заголовок <h2>Первая программа на JavaScript</h2>.

Теперь, когда веб-страница готова, откроем ее в веб-браузере:

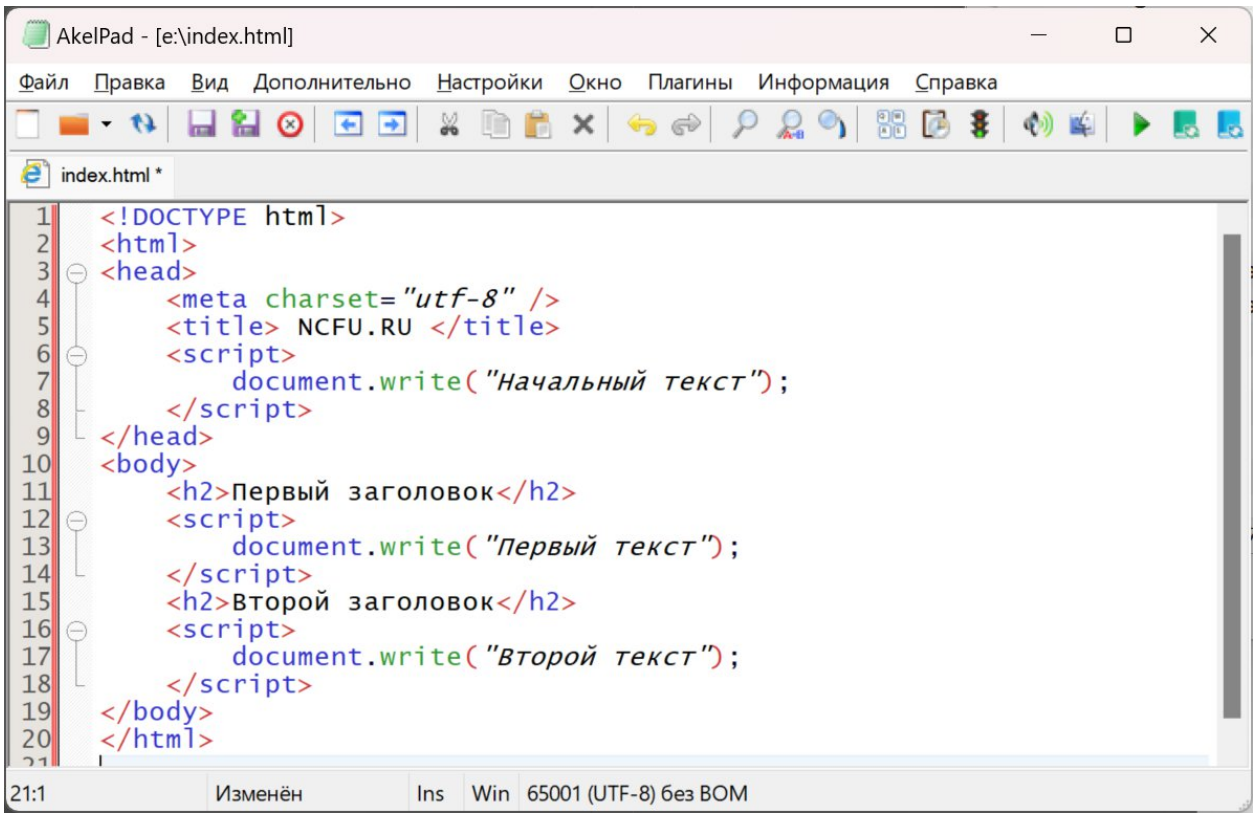


И веб-браузер отобразит заголовок, который мы передали в метод document.write() в коде javascript.

Когда браузер получает веб-страницу с кодом html и javascript, то он ее интерпретирует. Результат интерпретации в виде различных элементов - кнопок, полей ввода, текстовых блоков и т.д., мы видим перед собой в браузере. Интерпретация веб-страницы происходит последовательно сверху вниз.

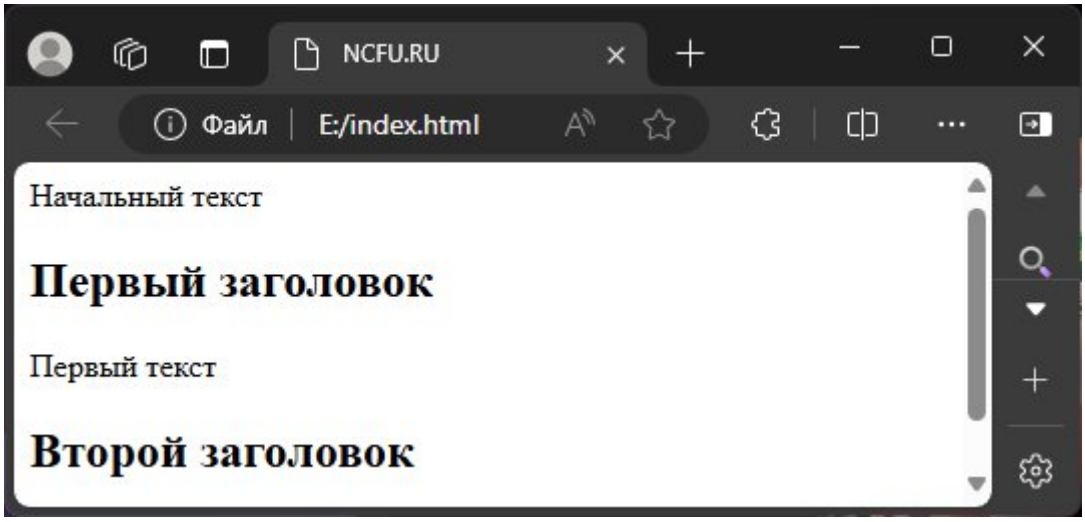
Когда браузер встречает на веб-странице элемент <script> с кодом javascript, то вступает в действие встроенный интерпретатор javascript. И пока он не закончит свою работу, дальше интерпретация веб-страницы не идет.

Рассмотрим небольшой пример и для этого определим файл **index.html** со следующим кодом:



Здесь три вставки кода javascript - один в секции <head> и по одному после каждого заголовка.

Откроем веб-страницу в браузере и мы увидим, что браузер последовательно выполняет код веб-страницы:



Здесь мы видим, что вначале выполняется код javascript из секции head, который выводит на веб-страницу некоторый текст:

document.write("Начальный текст");

Далее выводится первый стандартный html-элемент <h2>:

<h2>Первый заголовок</h2>

После этого выполняется вторая вставка кода на javascript:

document.write("Первый текст");

Затем выводится второй html-элемент <h2>:

<h2>Второй заголовок</h2>

И в конце выполняется последняя вставка кода на javascript:

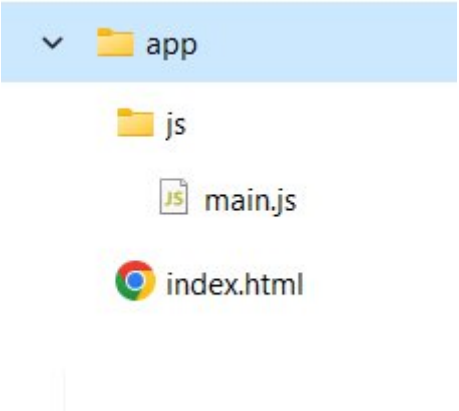
document.write("Второй текст");

После этого браузер закончит интерпретацию веб-страницы, и веб-страница окажется полностью загружена. Данный момент очень важен, поскольку может влиять на производительность. Поэтому нередко вставки кода javascript идут перед закрывающим тегом </body>, когда основная часть веб-страницы уже загружена в браузере.

Подключение внешнего файла JavaScript

В первом примере код javascript непосредственно определялся на веб-странице. Но есть еще один способ подключения кода JavaScript, который представляет вынесение кода во внешние файлы и их подключение с помощью тега <script>

Создайте каталог app, а в нем пустой файл страницы index.html. Теперь создадим в этом каталоге новый подкаталог. Назовем его js. Он будет предназначен для хранения файлов с кодом javascript. В этом подкаталоге создадим новый текстовый файл, который назовем main.js. Файлы с кодом javascript имеют расширение .js. То есть у нас получится следующая структура проекта в папке app:

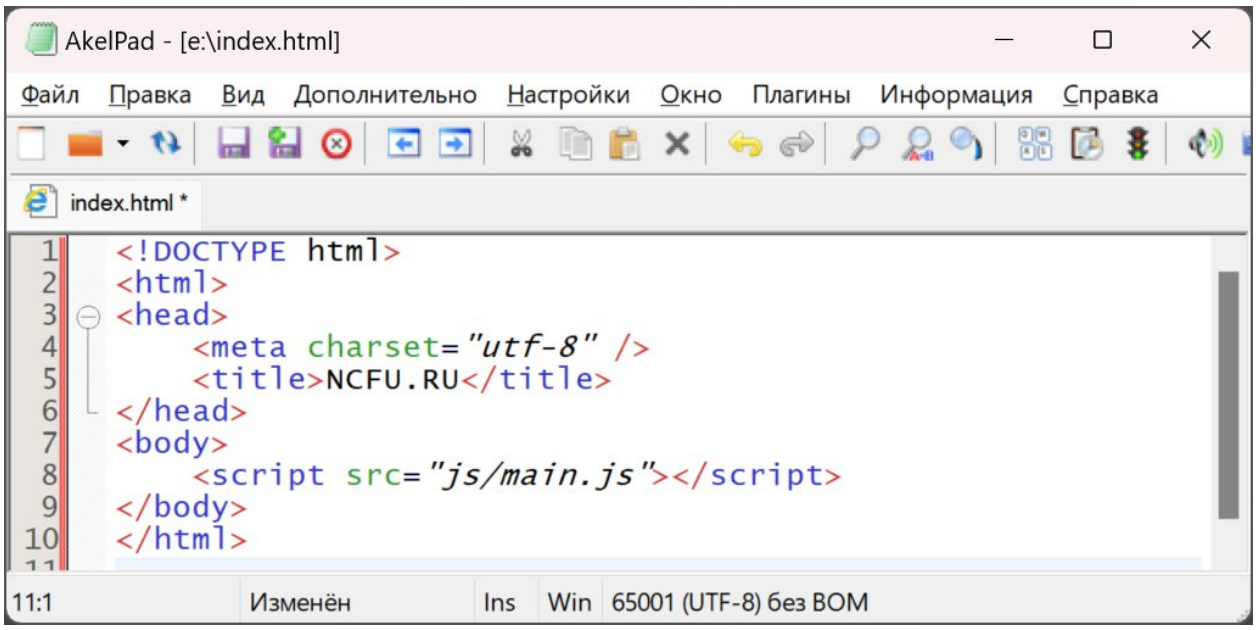


Откроем файл main.js в текстовом редакторе и определим в нем следующий код:

```
document.write("<h2>Первая программа на JavaScript</h2>");
document.write("Привет мир!");
```

Здесь для добавления на веб-страницу некоторого содержимого применяется метод document.write. Первый вызов метода document.write выводит заголовок <h2>, а второй вызов - обычный текст.

Теперь подключим этот файл на веб-страницу index.html:

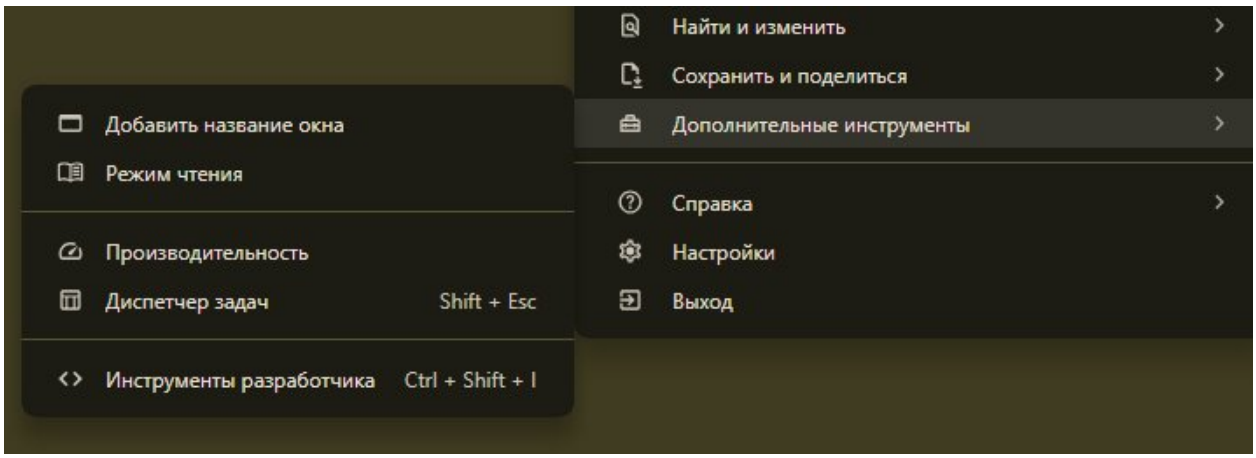


Чтобы подключить файл с кодом javascript на веб-страницу, применяется также тег <script>, у которого устанавливается атрибут src. Этот атрибут указывает на путь к файлу скрипта. В нашем случае используется относительный путь. Так как веб-страница находится в одной папке с каталогом js, то в качестве пути мы можем написать js/main.js.

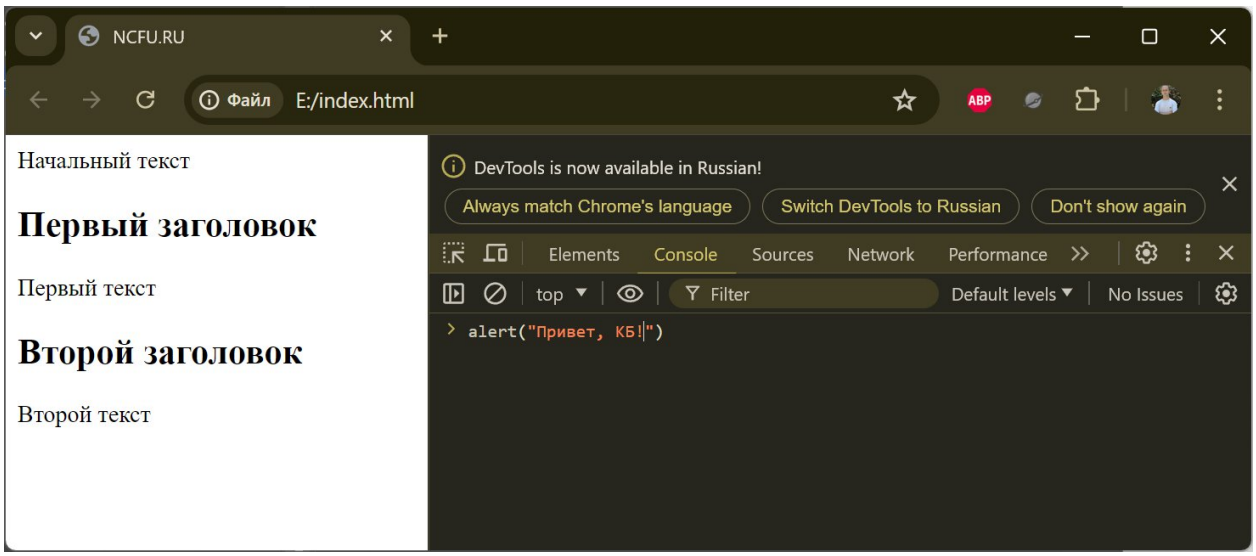
Также надо учитывать, что за открывающим тегом script должен обязательно следовать закрывающий </script>

Консоль браузера и console.log

Незаменимым инструментом при работе с JavaScript является консоль браузера, которая позволяет производить отладку программы. Во многих современных браузерах есть подобная консоль. Например, чтобы открыть консоль в Google Chrome, нам надо перейти в меню **Дополнительные инструменты (More Tools) -> Инструменты разработчика (Developer tools)**:



После этого в окне браузера откроется консоль. Мы можем напрямую вводить в консоль браузера выражения JavaScript, и они будут выполняться.

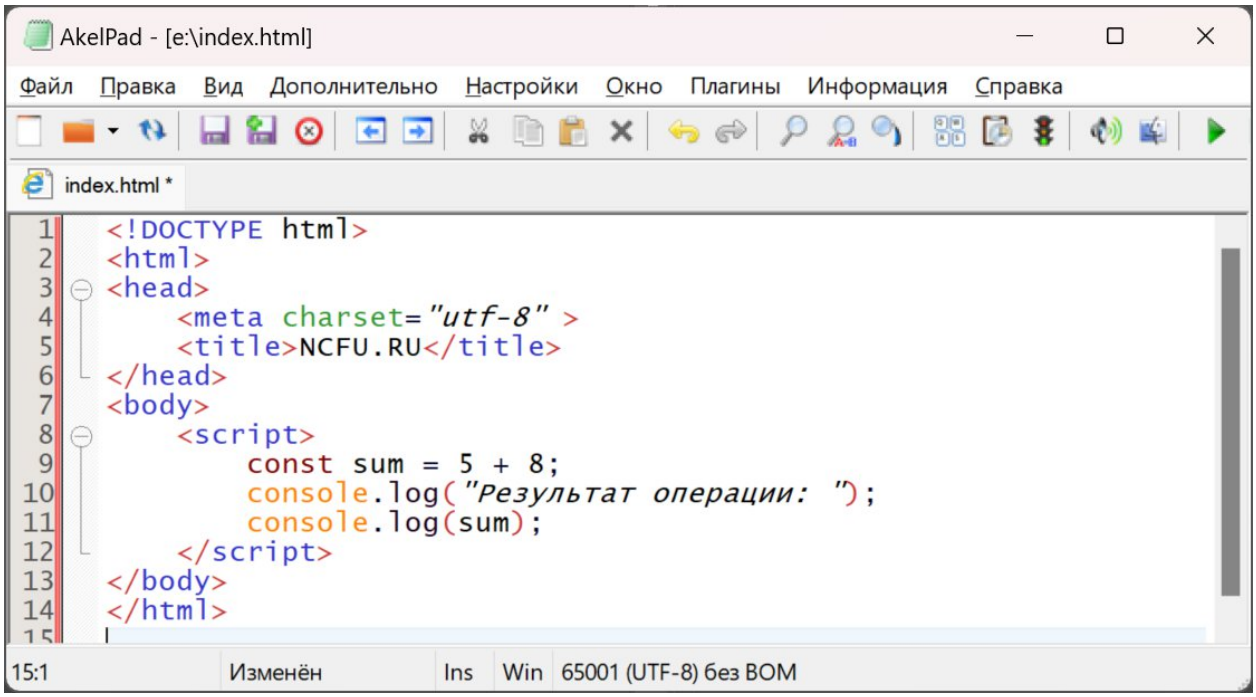


Функция **alert()** выводит в браузере окно с сообщением. В итоге после ввода этой команды и нажатия на клавишу Enter браузер выполнит эту функцию и отобразит нам окно с сообщением!

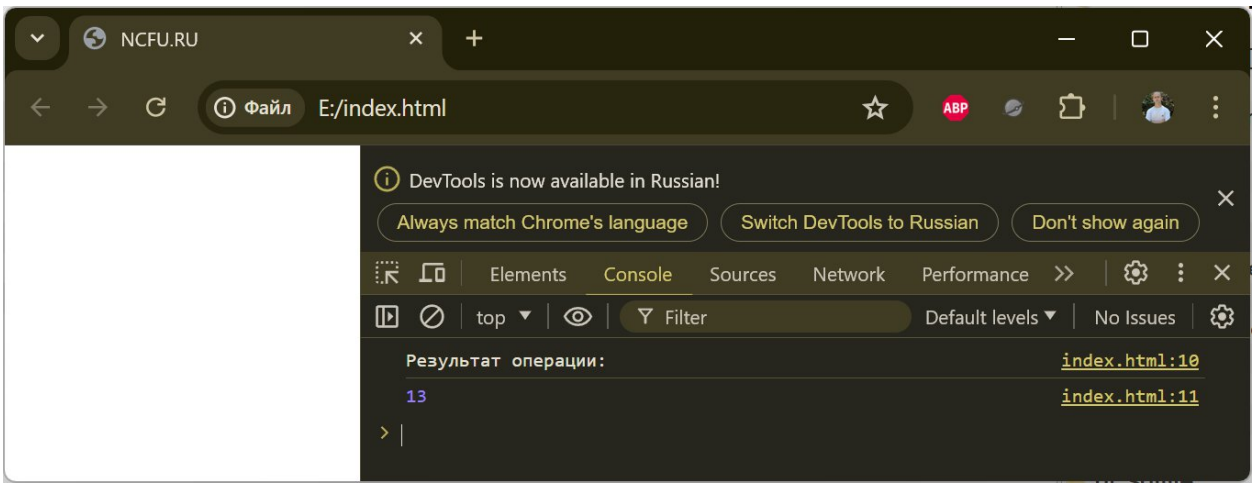
Таким образом, для написания и выполнения кода JavaScript нам в принципе не нужен ни текстовый редактор, ни файл веб-страницы, который содержит код javascript, достаточно одной консоли браузера. Однако набирать большой и сложный код JavaScript в консоли может быть не очень удобно, поэтому в дальнейшем для всех примеров по прежнему будет использовать код JavaScript, который встраивается на html-страницу.

Вывод на консоль и console.log

Для вывода различного рода информации в консоли браузера используется специальная функция **console.log()**. Например, определим html-страницу со следующим кодом



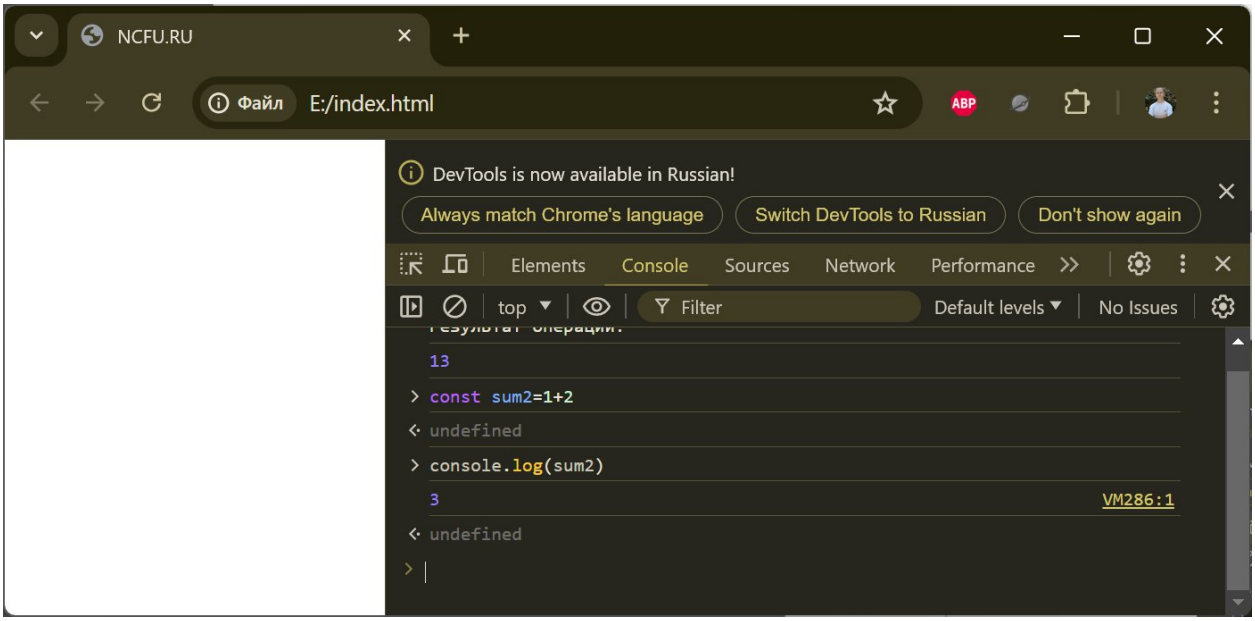
После запуска веб-страницы в браузере мы увидим в консоли результат выполнения кода:



Что очень полезно, в консоли браузера вы также можете заметить номера строк кода, где именно выполнялся вывод на консоль.

В дальнейшем мы часто будем обращаться к консоли и использовать функцию console.log.

Причем подобный код мы могли бы ввести в самой консоли:



Если нам надо, чтобы код в консоли переносился на новую строку без выполнения, то в конце выражения javascript нажимаем на комбинацию клавиш **Shift + Enter**. После ввода последней инструкции для выполнения введенного кода javascript нажимаем на Enter.

Задание

Создайте веб-страницу с кодом HTML и вставкой на языке JS

Занятие № 2 Введение в массивы

Для работы с наборами данных предназначены массивы. Для создания массива применяются квадратные скобки **[]**. Внутри квадратных скобок определяются элементы массива.

Простейшее определение массива:

const myArray = [];

В данном случае мы создаем массив, который называется myArray. Он пустой, поскольку внутри квадратных скобок не определено ни одного элемента. Но можно также добавить в него начальные данные:

```
> const people = ["Tom", "Alice", "Sam"];
  console.log(people);
  ▶ (3) ['Tom', 'Alice', 'Sam']
```

В этом случае в массиве myArray будет три элемента. Графически его можно представить так:

Индекс	Элемент
0	Tom
1	Alice
2	Sam

Для обращения к отдельным элементам массива используются индексы. Отсчет начинается с нуля, то есть первый элемент будет иметь индекс 0, а последний - 2:

```
> const people = ["Tom", "Alice", "Sam"];
  console.log(people[0]); // Tom
  const person3 = people[2]; // Sam
  console.log(person3); // Sam
  Tom
  Sam
```

Если мы попробуем обратиться к элементу по индексу больше размера массива, то мы получим undefined:

```
> const people = ["Tom", "Alice", "Sam"];
  console.log(people[7]); // undefined
  undefined
```

Также по индексу осуществляется установка значений для элементов массива:

```
> const people = ["Tom", "Alice", "Sam"];
  console.log(people[0]);
  people[0] = "Bob";
  console.log(people[0]);
  Tom
  Bob
```

Причем в отличие от других языков, как C# или Java, можно установить элемент, который изначально не установлен:

```
> const people = ["Tom", "Alice", "Sam"];
  console.log(people[7]); // undefined - в массиве только три элемента
  people[7] = "Bob";
  console.log(people[7]); // Bob
  undefined
  Bob
< undefined
```

Также стоит отметить, что в отличие от ряда языков программирования в JavaScript массивы не являются строго типизированными, один массив может хранить данные разных типов:

```
> const objects = ["Tom", 12, true, 3.14, false];
  console.log(objects);
  ▶ (5) ['Tom', 12, true, 3.14, false]
```

Многомерные массивы

Массивы могут быть одномерными и многомерными. Каждый элемент в многомерном массиве может представлять собой отдельный массив. Выше мы рассматривали одномерный массив, теперь создадим многомерный массив:

```
> const numbers1 = [0, 1, 2, 3, 4, 5 ]; // одномерный массив
  const numbers2 = [[0, 1, 2], [3, 4, 5] ]; // двумерный массив
```

Визуально оба массива можно представить следующим образом:

Одномерный массив numbers1

0	1	2	3	4	5

Двухмерный массив numbers2

0	1	2
3	4	5

Поскольку массив numbers2 двухмерный, он представляет собой простую таблицу. Каждый его элемент может представлять отдельный массив.

Рассмотрим еще один двумерный массив:

```
> const people = [
          ["Tom", 25, false],
          ["Bill", 38, true],
          ["Alice", 21, false]
  ];

  console.log(people[0]);
  console.log(people[1]);
  ▶ (3) ['Tom', 25, false]
  ▶ (3) ['Bill', 38, true]
```

Массив people можно представить в виде следующей таблицы:

Tom	25	false
Bill	38	true
Alice	21	false

Чтобы получить отдельный элемент массива, также используется индекс:

```
> const tomInfo = people[0];
```

Только теперь переменная tomInfo будет представлять массив. Чтобы получить элемент внутри вложенного массива, нам надо использовать его вторую размерность:

```
> console.log("Имя: ", people[0][0]); // Tom
  console.log("Возраст: ", people[0][1]); // 25
  Имя:  Tom
  Возраст:  25
```

То есть если визуально двумерный массив мы можем представить в виде таблицы, то элемент people[0][1] будет ссылаться на ячейку таблицы, которая находится на пересечении первой строки и второго столбца (первая размерность - 0 - строка, вторая размерность - 1 - столбец).

Также мы можем выполнить присвоение:

```
> const people = [
          ["Tom", 25, false],
          ["Bill", 38, true],
          ["Alice", 21, false]
  ];
  people[0][1] = 56; // присваиваем отдельное значение
  console.log(people[0][1]); // 56

  people[1] = ["Bob", 29, false]; // присваиваем массив
  console.log(people[1][0]); // Bob
  56
  Bob
```

При создании многомерных массивов мы не ограничены только двумерными, но также можем использовать массивы больших размерностей:

```
> const numbers = [];
  numbers[0] = []; // теперь numbers - двумерный массив
  numbers[0][0]=[]; // теперь numbers - трехмерный массив
  numbers[0][0][0] = 5; // первый элемент трехмерного массива равен 5
  console.log(numbers[0][0][0]);
  5
```

Задание

1. Дано: 1-мерный массив целых чисел размерности N.
    a. Заполните массив случайными числами в диапазоне от 1 до 99 (см. метод Math.random()).
    b. Выведите содержимое массива на экран.
    c. Найдите сумму элементов массива.
    d. Найдите минимальный элемент массива.
    e. Поменяйте местами минимальный и максимальный элемент массива.

f. Выведите содержимое массива на экран.

2. Дано: 2-мерный массив целых чисел размерности N x N.
    a. Заполните массив случайными числами в диапазоне от 1 до 99.
    b. Выведите содержимое массива на экран.
    c. Заполните главную диагональ значением 0.
    d. Посчитайте сумму элементов выше главной диагонали.
    e. Выведите на экран только элементы ниже главной диагонали.

Занятие № 3. Строки

Для создания строк мы можем как напрямую присваивать переменной или константе строку:

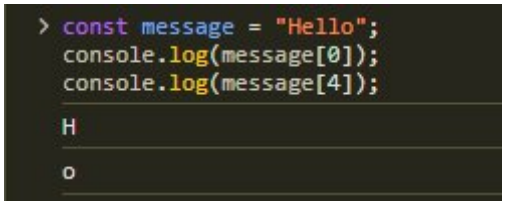
const message = "Hello";

Для работы со строками предназначен объект String, поэтому также можно использовать конструктор String:

const message = new String("Hello");

Но как правило, используется первый более краткий способ. В первом случае JavaScript при необходимости автоматически преобразует переменную примитивного типа в объект String.

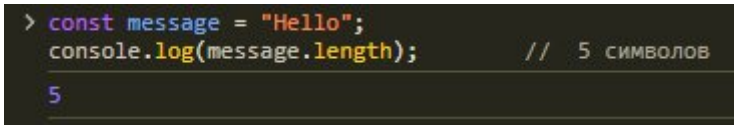
С помощью индексов можно обращаться к отдельным символам строки, как к элементам массива (как и в массивах индексация начинается с нуля):



Объект String имеет большой набор свойств и методов, с помощью которых мы можем манипулировать строками.

Длина строки

Свойство length указывает на длину строки:



Повторение строки

Метод repeat() позволяет создать строку путем многократного повторения другой строки. Количество повторов передается в качестве аргумента:

Поиск в строке

Для поиска в строке некоторой подстроки используются методы indexOf() (индекс первого вхождения подстроки) и lastIndexOf() (индекс последнего вхождения подстроки). Эти методы принимают два параметра:

indexOf(str, index)

lastIndexOf(str, index)

1. Подстроку, которую надо найти.
2. Индекс, с которого идет поиск (необязательный параметр).

Оба этих метода возвращают индекс символа, с которого в строке начинается подстрока. Если подстрока не найдена, то возвращается число -1.

```
> const hello = "привет мир. пока мир";
  const key = "мир";
  const firstPos = hello.indexOf(key);
  const lastPos = hello.lastIndexOf(key);
  console.log("Первое вхождение: ", firstPos);    // 7
  console.log("Последнее вхождение: ", lastPos);  // 17
Первое вхождение:  7
Последнее вхождение:  17
```

Применим поиск относительно индекса, например, начиная с индекса 10:

```
> const hello = "привет мир. пока мир";
  const key = "мир";
  const firstPos = hello.indexOf(key, 10);    // поиск с 10-го индекса
  console.log("Первое вхождение: ", firstPos);    // 17
Первое вхождение:  17
```

Следует учитывать, что поиск регистрозависимый:

```
> const hello = "привет мир. пока мир";
  const key = "Мир";
  const firstPos = hello.indexOf(key);
  console.log(firstPos);  // -1
-1
```

includes

Еще один метод - includes() возвращает true, если строка содержит определенную подстроку.

```
> const hello = "привет мир. пока мир";
   console.log(hello.includes("мир")); // true
  console.log(hello.includes("миг")); // false
true
false
```

С помощью второго дополнительного параметра можно определить индекс, с которого будет начинаться поиск подстроки:

```
> const hello = "привет мир. пока мир";
    console.log(hello.includes("мир", 5));  // true
   console.log(hello.includes("привет", 6));   // false
true
false
```

Выбор подстроки

Для того, чтобы вырезать из строки подстроку, применяются методы substring() и slice().

Метод substring() принимает два параметра:

substring(startIndex, endIndex)

1. индекс символа в строке, начиная с которого надо проводить обрезку строки. Обязательный параметр
2. индекс, до которого надо обрезать строку. Необязательный параметра - если он не указан, то обрезается вся остальная часть строки

```
> const hello = "привет мир. пока мир";
  const world = hello.substring(7, 10); // с 7-го по 10-й индекс
  console.log(world); // мир
  const bye = hello.substring(12);    // с 12 индекса до конца строки
  console.log(bye); // пока мир

  мир

  пока мир
```

Метод slice также позволяет получить из строки какую-то ее часть. Она принимает два параметра:

slice(startIndex, endIndex)

1. Начальный индекс подстроки в строке. Обязательный параметр
2. Конечный индекс подстроки в строке. Необязательный параметра - если он не указан, то обрезается вся остальная часть строки

```
> const hello = "привет мир. пока мир";
  const world = hello.slice(7, 10); // с 7-го по 10-й индекс
  console.log(world); // мир
  const bye = hello.slice(12);    // с 12 индекса до конца строки
  console.log(bye); // пока мир

  мир

  пока мир
```

Можно заметить, что этот метод похож на метод substring(), тем не менее между ними есть небольшие различие. Прежде всего, в slice() начальный индекс должен быть меньше чем конечный. В substring(), если начальный индекс больше конечного, то они меняются местами (то есть substring(5, 1) будет равноценно substring(1, 5)):

```
> const hello = "привет мир. пока мир";
  const world1 = hello.slice(6, 0); // не работает
  console.log(world1); //     пустая строка
  const world2 = hello.substring(6, 0); // аналогично hello.substring(0, 6)
  console.log(world2); // привет


  привет
```

Другое отличие, что slice позволяет использовать отрицательные индексы. Отрицательный индекс указывает на индекс символа относительно конца строки. substring() же отрицательные индексы не поддерживает:

```
> const hello = "привет мир. пока мир";
  const bye1 = hello.slice(-8, -4); // с 8-го индекса с конца до 4 индекса с конца
  console.log(bye1); // пока
  const bye2 = hello.substring(-8, -4); // не работает
  console.log(bye2); //

  пока
```

Следует отметить, что еще есть метод substr(). Этот метод не является частью стандарта и в целом не рекомендуется к использованию, однако он все еще может поддерживаться браузерами, и его до сих пор можно встретить в различных программах. Он принимает два параметра:

substr(startIndex, count)

1. Начальный индекс подстроки в строке. Обязательный параметр
2. Количество выбираемых символов. Необязательный параметра - если он не указан, то выбирается вся остальная часть строки

Применение:

```
> const hello = "привет мир. пока мир";
  const world = hello.substr(7, 3); // с 7-го индекса 3 символа
  console.log(world); // мир
  const bye = hello.substr(12); // с 12-го индекса до конца
  console.log(bye); // пока мир
  мир
  пока мир
```

Управление регистром символов

Для изменения регистра символов имеются методы toLowerCase() (для перевода в нижний регистр) и toUpperCase() (для перевода в верхний регистр).

```
> const hello = "Привет Том";
  console.log(hello.toLowerCase()); // привет том
  console.log(hello.toUpperCase()); // ПРИВЕТ ТОМ
  привет том
  ПРИВЕТ ТОМ
```

Получение символа по индексу

Чтобы получить определенный символ в строке по индексу, можно применять синтаксис массивов. Но также JavaScript предоставляет методы charAt() и charCodeAt(). Оба этих метода в качестве параметра принимают индекс символа:

```
> const hello = "Привет Том";
  console.log(hello.charAt(2));       // и
  console.log(hello.charCodeAt(2));   // 1080
  и
  1080
```

Но если в качестве результата метод charAt() возвращает сам символ, то метод charCodeAt() возвращает числовой код этого символа.

Удаление пробелов

Для удаления начальных и концевых пробелов в стоке используется метод trim():

```
> let hello = "   Привет Том  ";
  const beforeLength = hello.length;
  hello = hello.trim();
  const afterLength = hello.length;
  console.log("Длина строки до: ", beforeLength);     // 15
  console.log("Длина строки после: ", afterLength);   // 10
  Длина строки до:  15
  Длина строки после:  10
```

Дополнительно есть ряд методов, которые удаляют пробелы с определенной стороны строки:

- trimStart() удаляет пробел с начала строки (в зависимости от того, является ли письмо правостронним или левостронним, это может быть правый или левый край строки)
- trimEnd() удаляет пробел с конца строки (в зависимости от того, является ли письмо правостронним или левостронним, это может быть правый или левый край строки)

- trimLeft() удаляет пробел с левой части строки
- trimRight() удаляет пробел с правой части строки

Объединение строк

Метод concat() объединяет две строки:

```
> let hello = "Привет ";
  const world = "мир";
  hello = hello.concat(world);
  console.log(hello); // Привет мир
  Привет мир
```

Замена подстроки

Метод replace() заменяет первое вхождение одной подстроки на другую:

```
> let hello = "Добрый день";
  hello = hello.replace("день", "вечер");
  console.log(hello); // Добрый вечер
  Добрый вечер
```

Первый параметр метода указывает, какую подстроку надо заменить, а второй параметр - на какую подстроку надо заменить.

В то же время у этого метода есть одна особенность - он заменяет только первое вхождение подстроки:

```
> let menu = "Завтрак: каша, чай. Обед: суп, чай. Ужин: салат, чай.";
  menu = menu.replace("чай", "кофе");
  console.log(menu);  // Завтрак: каша, кофе. Обед: суп, чай. Ужин: салат, чай.
  Завтрак: каша, кофе. Обед: суп, чай. Ужин: салат, чай.
```

Однако еще один метод - replaceAll() позволяет заменить все вхождения подстроки:

```
> let menu = "Завтрак: каша, чай. Обед: суп, чай. Ужин: салат, чай.";
  menu = menu.replaceAll("чай", "кофе");
  console.log(menu);  // Завтрак: каша, кофе. Обед: суп, кофе. Ужин: салат, кофе.
  Завтрак: каша, кофе. Обед: суп, кофе. Ужин: салат, кофе.
```

Разделение строки

Метод split() разбивает строку на массив подстрок по определенному разделителю. В качестве разделителя используется строка, которая передается в метод:

```
> const message = "Сегодня была прекрасная погода";
  const messageParts = message.split(" ");
  console.log(messageParts);  // ["Сегодня", "была", "прекрасная", "погода"]
  ▶ (4) ['Сегодня', 'была', 'прекрасная', 'погода']
```

В данном случае строка разделяется по проблему, то есть в итоге в массиве messageParts окажется четыре элемента.

Проверка начала и окончания строки

Метод startsWith() возвращает true, если строка начинается с определенной подстроки. А метод endsWith() возвращает true, если строка оканчивается на определенную подстроку.

```
> const hello = "let me speak from my heart";
  console.log(hello.startsWith("let"));       // true
  console.log(hello.startsWith("Let"));       // false
  console.log(hello.startsWith("lets"));      // false

  console.log(hello.endsWith("heart"));       // true
  console.log(hello.startsWith("bart"));      // false
  true
  false
  false
  true
  false
```

При этом играет роль регистр символов, и из примера выше мы видим, что "let" не эквивалентно "Let".

Дополнительный второй параметр позволяет указать индекс (для startsWith - индекс с начала, а для endsWith - индекс с конца строки), относительно которого будет производиться сравнение:

```
> const hello = "let me speak from my heart";
  console.log(hello.startsWith("me", 4));     // true, "me" - 4 индекс с начала строки
   console.log(hello.startsWith("my", hello.length-8));    // true, "my" - 8 индекс с конца
  true
  true
```

Заполнение строки

Методы padStart() и padEnd() растянуть строку на определенное количество символов и заполнить строку слева и справа соответственно.

```
> let hello = "hello".padStart(8);  // "   hello"
  console.log(hello);
  hello = "hello".padEnd(8);      // "hello   "
  console.log(hello);
     hello
  hello
```

Вызов "hello".padStart(8) будет рястягивать строку "hello" на 8 символов. То есть изначально в строке "hello" 5 символов, значит, к ней будет добавлено 3 символа. При чем они будут добавлено в начале строки. По умолчанию добавляемые символы представляют пробелы. Аналогично вызов "hello".padEnd(8) растянет строку на 8 символов, но оставшиеся символы в виде пробелов будут добавлены в конец строки.

По умолчанию эти методы используют пробелы для заполнения, но в качестве второго параметра мы можем передать методам значение, которым надо дополнить строку:

```
> let hello = "hello".padStart(17, "JavaScript, ");  // "JavaScript, hello"
  hello = "hello".padEnd(12, " Eugene");      // "hello Eugene"
< 'hello Eugene'
```

Если добавляемое количество символов больше добавляемой строки, то добавляемая строка повторяется:

```
> let hello = "123".padStart(6, "0");  // "000123"
  hello = "123".padEnd(6, "0");       // "123000"
< '123000'
```

Шаблоны строк

Шаблоны строк (template strings / template literals) позволяют вставлять в строку различные значения. Подобный прием еще называют интерполяцией. Для этого строки заключаются в косые кавычки, а вставляемое значение предваряется символом $ и заключается в фигурные скобки:

```
> const name = "Tom";
  const hello = `Hello ${name}`;
  console.log(hello);     // Hello Tom
  Hello Tom
```

Здесь на место ${name} будет вставляться значение константы name. Таким образом, из шаблона `Hello ${name}` мы получим строку Hello Tom.

Подобным образом в строку можно вставлять сразу несколько значений:

```
> const name = "Tom";
  const age = 37;
  const userInfo = `${name} is ${age} years old`;
  console.log(userInfo);      // Tom is 37 years old
  Tom is 37 years old
```

Также вместо скалярных значений могут добавляться свойства сложных объектов:

```
> const tom ={
      name: "Tom",
      age: 22
  }
  const tomInfo = `${tom.name} is ${tom.age} years old`;
  console.log(tomInfo);       // Tom is 22 years old
  Tom is 22 years old
```

Любо можно вставлять более сложные вычисляемые выражения:

```
> function sum(x, y){
      return x + y;
  }
  const a = 5;
  const b = 4;

  const result = `${a} + ${b} = ${sum(a, b)}`;
  console.log(result);    // 5 + 4 = 9


  const expression = `${a} * ${b} = ${ a * b}`;
  console.log(expression);    // 5 * 4 = 20
  5 + 4 = 9
  5 * 4 = 20
```

В первом случае в шаблоне вызывается функция sum(), параметрам которой передаются значения констант a и b: ${sum(a, b)}. В итоге в это место будет вставлена сумма a и b.

Во втором случае в шаблоне выполняется операция умножения констант: ${ a * b}.

html-код в шаблонах

Шаблоны также могут хранить html-код, который будет динамически формироваться.

index.html

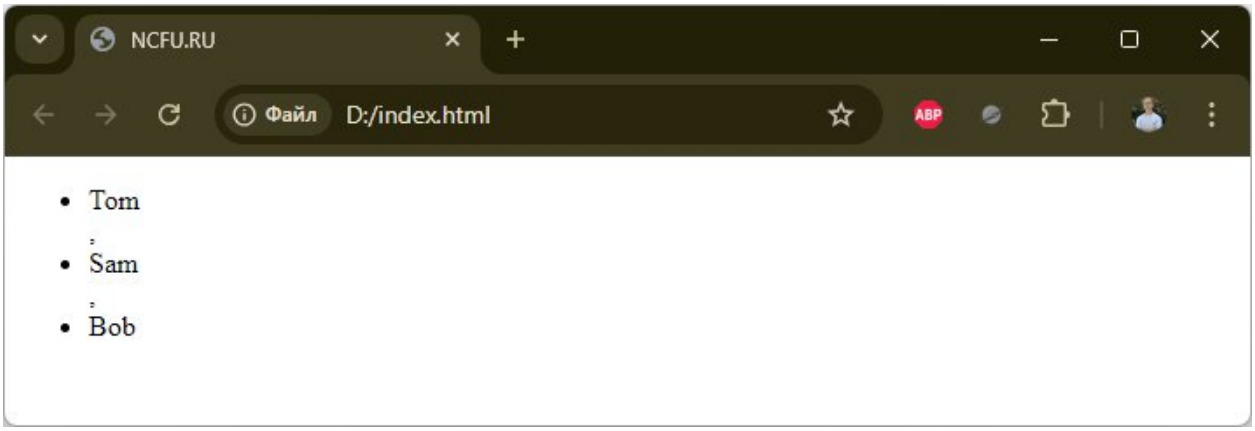
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.RU</title>
</head>
<body>
<script>
const tom = {name: "Tom", age: 37};
const markup = `<div>
    <p><b>Name</b>: ${tom.name}</p>
    <p><b>Age</b>: ${tom.age}</p>
</div>`;

document.body.innerHTML = markup;
</script>
</body>
</html>
```

Вложенные шаблоны

Рассмотрим другой пример - создадим из элементов массива список html:

index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.RU</title>
</head>
<body>
<script>
const people = [{name: "Tom", age: 37}, {name:"Sam", age: 41}, {name: "Bob", age: 21}];
const markup = `<ul>
    ${people.map(person => `<li>${person.name}</li>`)}
</ul>`;

document.body.innerHTML = markup;
</script>
</body>
</html>
```



Шаблоны строк в JavaScript

В данном случае мы имеем дело с вложенным шаблоном. То есть вначале определяется общий внешний шаблон:

const markup = `

  ${.....}

`;

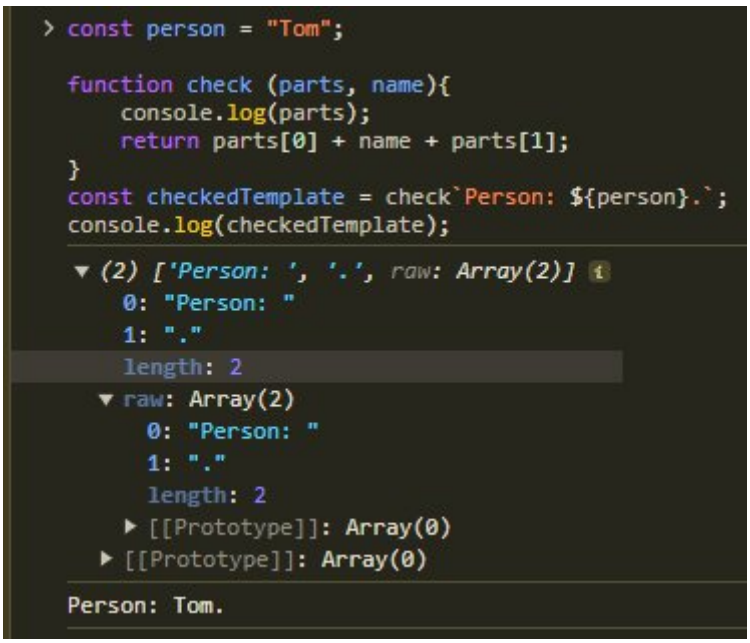
А в динамически формируемом выражении применяется еще один шаблон:

${people.map(person => `<li>${person.name}</li>`)}

В данном случае у массива people вызывается функция map(), которое определяет некоторое действие для каждого элемента массива. Это действие передается в map() в виде функции. Здесь для упрощения в качестве такой функции применяется лямбда-выражение. Оно получает каждый элемент массива через параметр person и для него формирует шаблон строки `<li>${person.name}</li>`.

Тег-функции и передача шаблона строки в функцию

JavaScript позволяет передать в функцию шаблон строки, причем не просто как строку, но и все ее динамически вычисляемые фрагменты в виде отдельных параметров. Для этого применяются тег-функции (tag function). Подобная возможность может применяться, например, для предобработки шаблонов и их значений. Рассмотрим следующий пример:



Здесь определена tag-функция check(), которая имеет два параметра: parts и name

Параметр parts - это массив частей шаблона, разделенных вставляемыми динамическими фрагментами. Второй параметр - name - это динамически вычисляемый фрагмент шаблона. То есть в данном случае мы предполагаем, что шаблон строки, который передается в функцию check(), будет иметь только один динамчески вычисляемый фрагмент. Соответственно в массиве parts будет два элемента: статическая часть шаблона, которая идет до вычисляемого фрагмента, и часть шаблона, которая идет после.

Чтобы было более ясно, о чем идет речь, в функции выводим на консоль эти элементы массива parts.

Функция возвращает return parts[0] + name + parts[1], то есть по сути мы просто возвращаем ранее сформированный шаблон, ничего не меняя.

Обратите внимание, как мы передаем этой функции шаблон:

const checkedTemplate = check`Person: ${person}.`;

шаблон просто указывается после названия функции. Или иначе говоря, tag-функция указывается как префикс перед шаблоном.

Из консольного вывода мы видим, что элементами массива parts являются подстроки "Person: " и ".". А в качестве значения параметра name передается строка "Tom". Стоит отметить, что даже если после динамически вычисляемого фрагмента больше не было бы никаких символов (например, `Person: ${person}`), то массив parts все равно имел бы два элемента, только вторым элементом тогда была бы пустая строка.

Задание

1. Строка содержат только заглавные буквы латинского алфавита (ABC…Z). Найдите какой из символов встречается чаще всего.
2. Строка содержат только заглавные буквы латинского алфавита (ABC…Z). Определите символ, который чаще всего встречается в файле сразу после буквы A. Например, в тексте ABCAABADDD после буквы A два раза стоит B, по одному разу — A и D. Для этого текста ответом будет B.
3. Строка содержит только заглавные буквы латинского алфавита (ABC…Z). Определите символ, который чаще всего встречается в файле между двумя одинаковыми символами.

Например, в тексте CBCABABACCC есть комбинации CBC, ABA (два раза), BAB и CCC. Чаще всего — 3 раза — между двумя одинаковыми символами стоит B, в ответе для этого случая надо написать B.

Занятие 4. Регулярные выражения

Регулярные выражения представляют шаблон, который используется для поиска или модификации строки. Для работы со строками мы, конечно, можем использовать стандартный набор методов типа String. Однако регулярные выражения предоставляют более гибкий инструмент.

Для работы с регулярными выражениями в JavaScript определен объект RegExp.

Определить регулярное выражение можно двумя способами:

```
> const myExp1 = /hello/;
  const myExp2 = new RegExp("hello");
```

Используемое здесь регулярное выражение довольно простое: оно состоит из одного слова "hello". В первом случае выражение помещается между двумя косыми чертами (так называемая литеральная нотация), а во втором случае используется конструктор RegExp, в который выражение передается в виде строки.

Обычно, если регулярное выражение определяется динамически (например, с помощью ввода пользователя), то применяется конструктор. Если выражение статическое, неизменяемое, то используется литеральная нотация, как в первом случае.

После определения регулярного выражения для обработки строк мы можем использовать один из методов RegExp:

- **test()**: проверяет, присутствует ли определенный шаблон в строке. Если строка соответствует шаблону, то этот метод возвращает **true**, иначе возвращается **false**.
- **exec()**: ищет вхождения определенного шаблона в строку и возвращает эти вхождения в виде массива.

Кроме того, дополнительно для работы с регулярными выражениями мы можем использовать ряд методов типа String:

- **match()**: также ищет вхождения определенного шаблона в строку и возвращает эти вхождения в виде массива.
- **replace()**: заменяет вхождения определенного шаблона в строке.
- **search()**: ищет вхождения определенного шаблона в строку и возвращает индекс первого вхождения.
- **split()**: разделяет строку в соответствии с определенным шаблоном и возвращает индекс полученные части строки в виде массива.

test. Проверка соответствия строки шаблону

Для проверки встречается ли в строке текст, который соответствует регулярному выражению, применяется метод **test()** объекта RegExp. Этот метод возвращает true, если строка содержит текст, который соответствует регулярному выражению, и false, если не содержит.

```
> const initialText = "hello world!";      // строка для поиска
  const exp = /hello/;     // регулярное выражение
  const result = exp.test(initialText);    // проверяем наличие в строке выражения exp
  console.log(result); // true

  const initialText2 = "Hi all";
  const result2 = exp.test(initialText2);
  console.log(result2); // false - в строке "Hi all" нет "hello"
  true
  false
```

В данном случае мы проверяем, есть ли в строках initialText ("hello world!") и initialText2 ("Hi all") выражение exp (/hello/). Поскольку в первой строке такой текст присутствует, то exp.test(initialText) возвращает true. В случае со второй строкой такой текст отсутствует, поэтому возвращается false.

Метод exec

Аналогично работает метод **exec** - он также проверяет, есть ли в строке текст, который удовлетворяет регулярному выражению. При наличии такого текста метод возвращает ту часть строки, которая соответствует выражению. Если соответствий нет, то возвращается значение **null**.

```
> const initialText = "hello world!";      // строка для поиска
  const exp = /hello/;     // регулярное выражение
  const result = exp.exec(initialText);    // проверяем наличие в строке выражения exp
  console.log(result); // ['hello', index: 0, input: 'hello world!', groups: undefined]

  const initialText2 = "Hi all";
  const result2 = exp.exec(initialText2);
  console.log(result2); // null - в строке "Hi all" нет "hello"
  ▼ ['hello', index: 0, input: 'hello world!', groups: undefined] i
      0: "hello"
      groups: undefined
      index: 0
      input: "hello world!"
      length: 1
    ▶ [[Prototype]]: Array(0)
```

Так, в строке "hello world!" есть текст, который соответствует шаблону /hello/, поэтому вызов метода exp.exec(initialText) возвратит вхождения текста, который соответствует выражению, в строку. И по консольному выводу мы видим, что это не просто текст, а массив значений.

Первый элемент массива - собственно текст, который соответствует выражению (в нашем случае "hello"). Второй элемент - индекс этого текста в строке (в нашем случае индекс 0 - начало строки). Третий элемент представляет входную строку. А четвертый элемент представляет группу, которая в примере выше не определена (группы мы разберем далее).

Группы символов

Выше в качестве регулярного выражения применялась строка "hello". И в реальности для проверки наличия этой подстроки в строке мы могли бы использовать и стандартные методы типа String. Однако это не просто строка, а шаблон, который также может включать специальные элементы синтаксиса регулярных выражений. Рассмотрим самые базовые шаблоны:

- **.** (точка) соответствует любому символу
- **a** (одиночный символ) соответствует символу "a"
- **ab** (объединение символов) соответствует последовательности символов "ab"

- **a|b** соответствует либо символу "a", либо символу "b" (символ | можно рассматривать как аналогию операции OR)

Например, нам надо проверить длину пароля - что она не менее 8 символов:

```
> const exp = /......../;    // регулярное выражение - 8 символов

  const password1 = "1234567890";
  const password2 = "qwery5";
  // проверяем первый пароль
  console.log(exp.test(password1));   // true -  password1 соответствует выражению exp
  // проверяем второй пароль
  console.log(exp.test(password2));   // false -  password2 не соответствует выражению exp
  true
  false
```

Здесь регулярное выражение "/......../" имеет 8 точек. То есть для соответствия этому выражению строка должна иметь как минимум 8 символов.

Другой пример:

```
> const exp = /word|work/;    // соответствует либо "word", либо "work"

  const str1 = "hello world";
  const str2 = "hello work";
  const str3 = "hello word";
  console.log(exp.test(str1));   // false
  console.log(exp.test(str2));   // true
  console.log(exp.test(str3));   // true
  false
  true
  true
```

Здесь выражение "/word|work/" соответствует тексту, который содержит либо "word", либо "work". Однако это выражение не оптимально - в обоих вариантах повторяются группу символов "wor". Общую группу символов мы можем взять в скобки: "/(wor)d|k/". Результат будет тот же:

```
> const exp = /(wor)d|k/;    // соответствует либо "word", либо "work"

  const str1 = "hello world";
  const str2 = "hello work";
  const str3 = "hello word";
  console.log(exp.test(str1));   // false
  console.log(exp.test(str2));   // true
  console.log(exp.test(str3));   // true
  false
  true
  true
```

Другой пример - проверим, принадлежит ли почтовый адрес "yandex.ru" или "mail.ru":

```
> const exp = /@yandex|@mail(.ru)/;    // соответствует либо "@yandex.ru" либо "@mail.ru"

  const email1 = "tom@mail.ru";
  const email2 = "tom@gmail.ru";
  const email3 = "tom@yandex.ru";
  console.log(exp.test(email1));   // true
  console.log(exp.test(email2));   // false
  console.log(exp.test(email3));   // true

  true

  false

  true
```

Определение классов символов

Для определения регулярных выражений мы можем использовать классы символов. Для определения класса символов применяются квадратные скобки:

- **[xyz]** (альтернативное соответствие): соответствует одному из символов: x, y или z (аналог x|y|z)
- **[^xyz]** (отрицание): соответствует тексту, который содержит любые символы КРОМЕ x, y или z
- **[a-zA-Z]** (диапазон): соответствует любому символу из диапазона a-z или A-Z

Например, нам надо проверить, есть ли в тексте символы "a", "b" или "c":

```
> const exp = /[abc]/;    // соответствует либо "a", либо "b", либо "c"

  const str1 = "JavaScript";
  const str2 = "Pascal";
  const str3 = "Python";
  console.log(exp.test(str1));   // true
  console.log(exp.test(str2));   // true
  console.log(exp.test(str3));   // false

  true

  true

  false
```

Выражение [abc] указывает на то, что строка должна иметь одну из трех букв. Выражение "[abc]" также эквивалентно выражению "a|b|c".

Возьмем более практический пример. Допустим, у нас есть 4-х символьные pin-коды, и нам надо проверить, что pin-код содержит только цифры:

```
> const exp = /[0-9][0-9][0-9][0-9]/;    // соответствует четырем цифрам подряд

  const code1 = "1234";
  const code2 = "jav5";
  const code3 = "3452";
  console.log(exp.test(code1));   // true
  console.log(exp.test(code2));   // false
  console.log(exp.test(code3));   // true

  true

  false

  true
```

Выражение [0-9][0-9][0-9][0-9] соответствует любой последовательности из 4 цифр подряд. Например, такому шаблону соответствует строка "3452", но НЕ соответствует строка "jav5" (здесь только

одна цифра). Строка "jav5" соответствовала бы шаблону "[a-z][a-z][a-z][0-9]" (первые три алфавитных символа в нижнем регистре, за которыми идет цифра).

Сразу стоит отметить, что выражение [0-9][0-9][0-9][0-9] не оптимально, и далее мы посмотрим, как его можно упростить.

Еще один пример - применим отрицание:

```
> const exp = /[^a-z]/;    // соответствует любым символам, кроме символов из диапазона a-z

  const code1 = "zorro";
  const code2 = "zorro5";
  const code3 = "34521";
  console.log(exp.test(code1));   // false
  console.log(exp.test(code2));   // true
  console.log(exp.test(code3));   // true

  false

  true

  true
```

Здесь строки проверяются на соответствие выражению "[^a-z]", которое соответствует любым символам, кроме символов из диапазона a-z. Например, строка "zorro" НЕ соответствует этому выражению. Однако ему соответствует строка "zorro5", потому что в ней есть символ, не входящий в диапазон "a-z".

При необходимости мы можем собирать комбинации выражений:

```
> const exp = /[дт]о[нм]/;    // соответствует строкам "дом", "том", "дон", "тон"

  const str1 = "дома";
  const str2 = "сома";
  const str3 = "тона";
  console.log(exp.test(str1));   // true
  console.log(exp.test(str2));   // false
  console.log(exp.test(str3));   // true

  true

  false

  true
```

Выражение [дт]о[нм] указывает на те строки, которые могут содержать подстроки "дом", "том", "дон", "тон".

Метасимволы

Вместо определения своих классов символов мы можем использовать встроенные, которые еще называют метасимволы - символы, которые имеют определенный смысл:

- \d: соответствует любой цифре от 0 до 9. Аналогичен выражению [0-9]
- \D: соответствует любому символу, который не является цифрой. Аналогичен выражению [^0-9]
- \w: соответствует любой букве, цифре или символу подчеркивания (диапазоны A–Z, a–z, 0–9). Аналогичен выражению [a-zA-Z_0-9]
- \W: соответствует любому символу, который не является буквой, цифрой или символом подчеркивания (то есть не находится в следующих диапазонах A–Z, a–z, 0–9). Аналогичен выражению [^\w]

- \s: соответствует пробелу. Аналогичен выражению [\t\n\x0B\f\r]
- \S: соответствует любому символу, который не является пробелом. Аналогичен выражению [^\s]
- .: соответствует любому символу

Здесь надо заметить, что метасимвол \w применяется только для букв латинского алфавита, кириллические символы для него не подходят.

Так, выше для проверки, что код имеет только 4 цифры, использовалось выражение /[0-9][0-9][0-9][0-9]/. Мы его можем сократить, используя метасимвол "\d":

```
> const exp = /\d\d\d\d/;    // соответствует четырем цифрам подряд

  const code1 = "1234";
  const code2 = "jav5";
  const code3 = "3452";
  console.log(exp.test(code1));   // true
  console.log(exp.test(code2));   // false
  console.log(exp.test(code3));   // true

  true

  false

  true
```

Другой пример. Допустим, нам надо найти строки, где определен номер телефона. Причем, номер телефона в формате +x-xxx-xxx-xxxx:

```
> const exp = /\+\d-\d\d\d-\d\d\d-\d\d\d\d/;
  const contact1 = "Email: mycomp@gmail.com";
  const contact2 = "Phone: +1-234-567-8901";
  console.log(exp.test(contact1));    // false
  console.log(exp.test(contact2));    // true

  false

  true
```

Так, номеру телефона +1-234-567-8901 соответствует /\+\d-\d\d\d-\d\d\d-\d\d\d\d/:

\+	\d	-	\d	\d	\d	-	\d	\d	\d	-	\d	\d	\d	\d
+	1	-	2	3	4	-	5	6	7	-	8	9	0	2

Обратите внимание на слеш перед плюсом (\+). Поскольку плюс + имеет специальное значение, то, чтобы указать, что мы имеет ввиду именно плюс как символ строки, перед ним ставится слеш.

В результате в строке "Phone: +1-234-567-8901" метод exp.test(contact2) сопоставит с регулярным выражением подстроку "+1-234-567-8901"

Ограничение применения регулярных выражений

Ряд специальных символов позволяют ограничить диапазон применения регулярных выражений:

^: соответствует началу строки. Например, ^h соответствует строке "home", но не "ohma", так как h должен представлять начало строки

- **$**: соответствует концу строки. Например, м$ соответствует строке "дом", так как строка должна оканчиваться на букву м
- **\b**: соответствует началу или концу слова.

- **\B**: не учитывает границы слова

Например, нам нужно найти строки с номером телефона:

```
> const exp = /\d\d\d\d\d\d\d\d\d\d/;    // соответствует 10 цифрам подряд

  const phone1 = "+12345678901";
  const phone2 = "42345678901";
  console.log(exp.test(phone1));   // true
  console.log(exp.test(phone2));   // true

  true

  true
```

Шаблону /\d\d\d\d\d\d\d\d\d\d/ соответствуют как строка "+12345678901", так и строка "42345678901". Но, допустим, нам надо найти номера телефонов, которые не предваряются плюсом +. В этом случае мы можем использовать регулярное выражение /^\d\d\d\d\d\d\d\d\d\d/. Другой пример. Пусть нам надо проверить, есть ли в строке упоминание языка "Java". Наивный подход состоял бы в использовании регулярного выражения /Java/:

```
> const exp = /Java/;

  const str1 = "Java is a high-level, object-oriented programming language";
  const str2 = "JavaScript is a programming language of the World Wide Web";
  console.log(exp.test(str1));   // true
  console.log(exp.test(str2));   // true

  true

  true
```

Однако в реальности шаблон "/Java/" соответствует любой строке, которая содержит подстроку "Java", в том числе строке "JavaScript". Однако нам надо найти только те строки, где речь идет именно о Java, а не о JavaScript. И в этом случае мы можем ограничить поиск границами слова с помощью "\b":

```
> const exp = /Java\b/;    //

  const str1 = "Java is a high-level, object-oriented programming language";
  const str2 = "JavaScript is a programming language of the World Wide Web";
  console.log(exp.test(str1));   // true
  console.log(exp.test(str2));   // false

  true

  false
```

Флаг "\B", наоборот, указывать сопоставлять шаблон с подстроками, которые не являются слова:

```
> const exp = /Java\B/;    //

  const str1 = "Java is a high-level, object-oriented programming language";
  const str2 = "JavaScript is a programming language of the World Wide Web";
  console.log(exp.test(str1));   // false
  console.log(exp.test(str2));   // true

  false

  true
```

Флаги выражений

Флаги позволяют настроить поведение регулярных выражений. Каждый флаг представляет отдельный символ, который ставится в конце регулярного выражения. В JavaScript применяются следующие флаги:

- Флаг **global** позволяет найти все подстроки, которые соответствуют регулярному выражению. По умолчанию при поиске подстрок регулярное выражение выбирает первую попавшуюся подстроку из строки, которая соответствует выражению. Хотя в строке может быть множество подстрок, которые также соответствуют выражению. Для этого применяется данный флаг в виде символа **g** в выражениях
- Флаг **ignoreCase** позволяет найти подстоки, которые соответствуют регулярному выражению, вне зависимости от регистра символов в строке. Для этого в регулярных выражениях применяется символ **i**
- Флаг **multiline** позволяет найти подстроки, которые соответствуют регулярному выражению, в многострочном тексте. Для этого в регулярных выражениях применяется символ **m**
- Флаг **dotAll** позволяет сопоставить точку в регулярном выражении с любым символом текста, в том числе с разделителем строки. Для этого в регулярных выражениях применяется символ **s**

Флаг i. Регистр символов

Рассмотрим следующий пример:

```
> const str = "Hello World";
  const exp = /world/;
  console.log(exp.test(str)); // false
  false
```

Здесь совпадения строки с выражением нет, так как "World" отличается от "world" по регистру. В этом случае надо изменить регулярное выражение, добавив в него флаг **i**:

```
> const str = "Hello World";
  const exp = /world/i;
  console.log(exp.test(str)); // true
  true
```

Обратите внимание, где в регулярном выражении указывается флаг: /world/i - в самом конце регулярного выражения.

Флаг s

Флаг s позволяет сопоставить символ . (точка) с любым символом, в том числе и с разделителем строки. Например, возьмем следующий пример:

```
> const str = "hello\nworld";
  const exp = /hello world/;
  console.log(exp.test(str)); // false
  false
```

Здесь в строке "hello\nworld" слова "hello" и "world" разделены переносом строки (например, мы имеем дело с многострочным текстом). Однако, например, мы хотим, чтобы JavaScript не учитывал перенос строки и чтобы данный текст соответствовал регулярному выражению /hello world/. В этом случае мы можем применить флаг s:

```
> const str = "hello\nworld";
  const exp = /hello.world/s;
  console.log(exp.test(str)); // true
  true
```

В выражении /hello.world/s точка означает произвольный символ. Однако без флага **s** данное выражение не будет соответствовать многострочному тексту.

Комбинация флагов

Также можно использовать сразу несколько флагов:

```
> const str = "hello\nWorld";
  const exp = /hello.world/si;
  console.log(exp.test(str)); // true
  true
```

Кроме рассмотренных в прошлых статьях специальных классов символов регулярных выражений есть еще одна группа комбинаций, которая указывает, как символы в строке будут повторяться. Такие комбинации еще называют квантификаторами:

- *: соответствует любому количеству повторений или отсутствию последовательности символов
- ?: соответствует одному вхождению последовательности символов или ее отсутствию в строке. Например, /h?ome/ соответствует подстрокам "home" и "ome".
- +: соответствует одному и более повторений последовательности символов
- {n}: соответствует n-ому количеству повторений предыдущего символа. Например, h{3} соответствует подстроке "hhh"
- {n,}: соответствует n и более количеству повторений предыдущего символа. Например, h{3,} соответствует подстрокам "hhh", "hhhh", "hhhhh" и т.д.
- {n,m}: соответствует от n до m повторений предыдущего символа. Например, h{2, 4} соответствует подстрокам "hh", "hhh", "hhhh".

Необязательные символы

Номер телефона может иметь дефисы для разделения отдельных блоков цифр, например, "+1-234-567-8901", а может не иметь разделителей, например, "12345678901". То есть в данном случае дефисы-разделители необязательны. И мы можем отразить это с помощью квантификатора ?

```
> const exp = /\d-?\d\d\d-?\d\d\d-?\d\d\d\d/;

  const phone1 = "+1-234-567-8901";
  const phone2 = "12345678901";
  const phone3 = "1-2345678901";
  console.log(exp.test(phone1));   // true
  console.log(exp.test(phone2));   // true
  console.log(exp.test(phone3));   // true
  true
  true
  true
```

Здесь "-?" отражает, что символ "-" может быть необязательным, однако если он присутствует, то только один раз.

Произвольное количество символов

Символ * указывает, что предыдущий символ может встречаться произвольное число раз (в том числе 0 раз). Например:

```
> const exp = /;*/;    // соответствует любому количеству символов ;

  const str1 = "number1 = 3";
  const str2 = "number2 = 4;";
  const str3 = "number3 = 5;;;";
  console.log(exp.test(str1));   // true
  console.log(exp.test(str2));   // true
  console.log(exp.test(str3));   // true
  true
  true
  true
```

Регулярное выражение /;*/ указывает, что точка с запятой (;) может встречаться 1 и более раз, либо может вообще не встречаться.

Символ встречается как минимум один раз

Квантификатор + указывает, что предыдущий символ может встречаться один или большее количество раз. Например:

```
> const exp = /;+/;    // соответствует 1 и более символов ;

  const str1 = "number1 = 3";
  const str2 = "number2 = 4;";
  const str3 = "number3 = 5;;;";
  console.log(exp.test(str1));   // false
  console.log(exp.test(str2));   // true
  console.log(exp.test(str3));   // true
  false
  true
  true
```

Регулярное выражение /;+/ указывает, что точка с запятой (;) может встречаться как минимум 1 раз (или большее количество раз).

Точное количество вхождений

Квантификатор **{n}** позволяет определить точное количество повторений предыдущего символа через значение n. Например, для определения телефонного номера может использоваться выражение /\d\d\d\d\d\d\d\d\d\d\d/ - 11 цифр подряд. Однако оно неоптимально. И для его сокращения мы можем использовать другое выражение: /\d{11}/ - символ "\d" (цифровой символ) встречается 11 раз подряд. Например:

```
> const exp = /\d{11}/;

  const phone1 = "+12345678901";
  const phone2 = "1-2345678901";
  const phone3 = "12345678901";
  console.log(exp.test(phone1));    // true
  console.log(exp.test(phone2));    // false
  console.log(exp.test(phone3));    // true

  true

  false

  true
```

Но что, если блоки цифр у нас разделены разделителем-дефисом типа "+1-234-567-8901". В этот случае мы могли бы задать длину для каждого блока: /\d-\d{3}-\d{3}-\d{4}/. Например:

```
> const exp = /\d-\d{3}-\d{3}-\d{4}/;

  const phone1 = "+12345678901";
  const phone2 = "1-234-567-8901";
  const phone3 = "12345678901";
  console.log(exp.test(phone1));    // false
  console.log(exp.test(phone2));    // true
  console.log(exp.test(phone3));    // false

  false

  true

  false
```

Комбинируя с другими квантификаторами, можно сделать разделители-дефисы необязательными:

```
> const exp = /\d-?\d{3}-?\d{3}-?\d{4}/;

  const phone1 = "+12345678901";
  const phone2 = "1-234-567-8901";
  const phone3 = "1-2345678901";
  console.log(exp.test(phone1));    // true
  console.log(exp.test(phone2));    // true
  console.log(exp.test(phone3));    // true

  true

  true

  true
```

Определение минимального количества вхождений

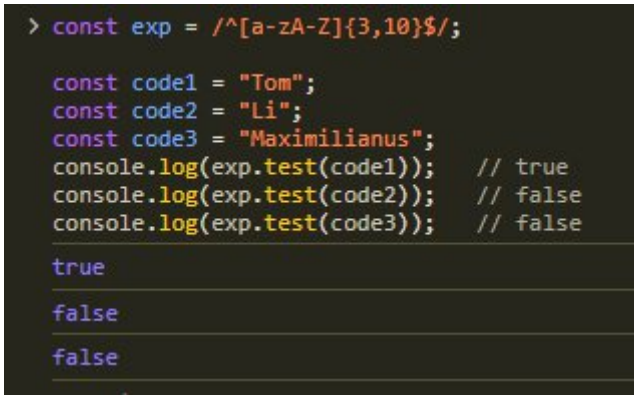
Квантификатор {n,} позволяет задать через n минимальное количество вхождений. Допустим, у нас пароли должны иметь как минимум 8 символов:

```
> const exp = /\w{8,}/;

  const code1 = "1234567890";
  const code2 = "qwery5";
  const code3 = "password123";
  console.log(exp.test(code1));    // true
  console.log(exp.test(code2));    // false
  console.log(exp.test(code3));    // true

  true

  false

  true
```

Метасимвол "\w" соответствует любому цифровому и алфавитному символу или символу подчеркивания. Соответственно выражение /\w{8,}/ соответствует строкам, где есть подстрока из как минимум 8 таких символов.

Определение минимального и максимального количества вхождений

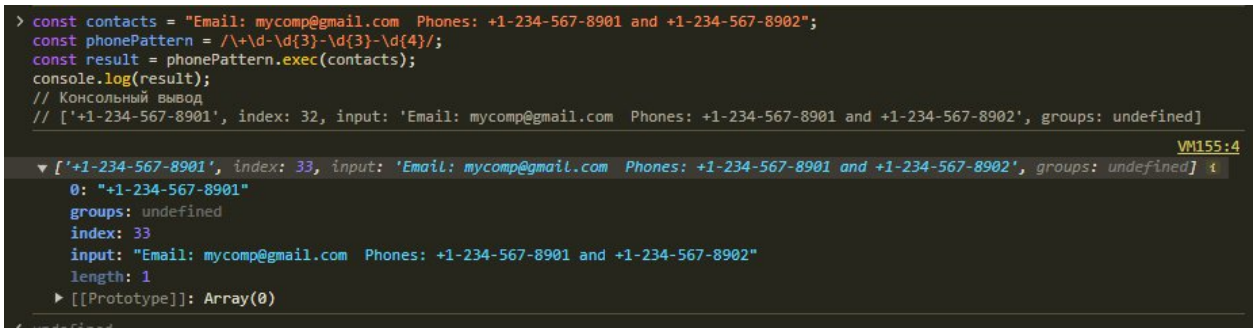
Квантификатор {n,m} позволяет определить одновременно минимальное (n) и максимальное (m) количество повторений. Например, мы хотим, чтобы имена у нас были не слишком короткими, скажем, не меньше 3 символов, и не слишком длинными (например, не больше 10 символов):



Выражение /^[a-zA-Z]{3,10}$/ говорит, что любой символ из диапазонов "a-z" и "A-Z" должен повторяться не меньше 3 раз и не больше 10 раз. Причем здесь также указано, что это должно быть отдельное слово. Для этого вначале указывается символ "^", а в конце символ "$".

Поиск в строке

Для поиска в строке подстроки, которая соответствует регулярному выражению, применяется метод **exec()** объекта RegExp. Этот метод принимает строку для поиска и возвращает результат в виде массива. Например:



Здесь проверяем, есть ли в строке contacts текст, который соответствует регулярному выражению phonePattern (то есть представляет номер телефона). В качестве результата возвращается массив из следующих элементов:

- Первый элемент массива - непосредственно тот текст, который соответствует регулярному выражению. Так, в примере выше это текст "+1-234-567-8901"
- Второй параметр - index - индекс найденного текста в строке
- Третий параметр - input - входная строка
- Последний элемент представляет отдельные группы

Если в строке не найден текст, который соответствует регулярному выражению, то возвращается **null**

Получим отдельные элементы этого массива:

```
> const contacts = "Email: mycomp@gmail.com Phones: +1-234-567-8901 and +1-234-567-8902";
  const phonePattern = /\+\d-\d{3}-\d{3}-\d{4}/;
  const result = phonePattern.exec(contacts);
  if(result){
      console.log("Phone number:", result[0]);    // +1-234-567-8901
      console.log("Index:", result.index);     // 32
  }
  Phone number: +1-234-567-8901
  Index: 32
```

Однако в строке у нас два телефонных номера (может быть и больше). И мы хотим извлечь все эти номера, а не только первый номер. В этом случае нам надо воспользоваться флагом **g**

Так, изменим предыдущий пример, применив флаг **g**:

```
> const contacts = "Email: mycomp@gmail.com Phones: +1-234-567-8901 and +1-234-567-8902";
  const phonePattern = /\+\d-\d{3}-\d{3}-\d{4}/g;
  let result;
  while ((result = phonePattern.exec(contacts)) !== null){
      console.log("Phone number:", result[0]);
      console.log("Index: ", result.index);
  }
  Phone number: +1-234-567-8901
  Index:  32
  Phone number: +1-234-567-8902
  Index:  52
```

В цикле while извлекаем все сопоставления шаблона с текстом в переменную result, пока не останется сопоставлений. Обратите внимание, где в регулярном выражении указывается флаг **g**: /\+\d-\d{3}-\d{3}-\d{4}/**g**.

Определение групп в регулярных выражениях

Для поиска в строке более сложных соответствий применяются группы. В регулярных выражениях группы заключаются в скобки. Например, нам надо получить дату в определенном формате. С помощью метода exec() объекта RegExp мы можем получить все совпадение полностью. Допустим, дата представлена в формате "yyyy-mm-dd":

```
> const exp = /\d{4}-\d{2}-\d{2}/;
  const text = "Publication Date: 2024-10-10"
  const result = exp.exec(text);

  console.log(result[0]);
  2024-10-10
```

Из результата метода exec мы можем извлечь результат - нужную нам дату. Однако что, если мы хотим получить отдельные компоненты дат - год, месяц, день? В этом случае мы можем воспользоваться группами.

Каждая группа представляет некоторый сегмент регулярного выражения, который заключен в круглые скобки. Например:

```
> const exp = /(\d{4})-(\d{2})-(\d{2})/;
  const text = "Publication Date: 2024-10-10"
  const result = exp.exec(text);

  console.log(result);
  ▸ (4) ['2024-10-10', '2024', '10', '10', index: 18, input: 'Publication Date: 2024-10-10', groups: undefined]
```

Здесь регулярное выражение /(\d{4})-(\d{2})-(\d{2})/ содержит три группы. Первая группа - (\d{4}) состоит из 4 цифр и представляет год. Вторая группа - (\d{2}) состоит из 2 цифр и представляет месяц. Третья группа также состоит из 2 цифр и представляет день.

Здесь применяется регулярное выражение "/(\d{4})-(\d{2})-(\d{2})/", где определены три группы:

1. Первая группа (\d{4}) соответствует числу из четырех цифр
2. Вторая группа (\d{2}) соответствует числу из двух цифр
3. Третья группа аналогична второй

И если мы посмотрим на результат метода **exec()**, то мы увидим, что кроме собственно соответствия дате он содержит соответствия для каждой группы:

```
  ▸ (4) ['2024-10-10', '2024', '10', '10', index: 18, input: 'Publication Date: 2024-10-10', groups: undefined]
```

Полученный результат представляет массив, где первый элемент (с индексом 0) всегда представляет подстроку, совпавшую с регулярным выражением. Все последующие элементы этого массива представляют группы. То есть первая группа имеет индекс 1, вторая - индекс 2 и так далее. Соответственно, применяя индексы, мы можем получить все соответствия группам из регулярного выражения:

```
> const exp = /(\d{4})-(\d{2})-(\d{2})/;
  const text = "Publication Date: 2024-10-10"
  const result = exp.exec(text);

  console.log(result[0]);
  console.log(result[1]);
  console.log(result[2]);
  console.log(result[3]);
  2024-10-10
  2024
  10
  10
```

Получив значения отдельных групп, можно произвести с ними некоторые действия, например, преобразовать в другой формат даты:

```
> console.log(`${result[3]}.${result[2]}.${result[1]}`);
  10.10.2024
```

Группировка упрощает создание более сложных регулярных выражений. К группам, как и к отдельным символам, можно применять квантификаторы. Например, выражение (la)+ представляет одно и более повторений строки "la"&

Именованные группы

JavaScript позволяет назначить каждой группе в регулярных выражениях определенное имя. С помощью этого имени потом можно получить значение, которое соответствует этой группе. Для

установки имени группы внутри скобок, которые определяют группу, ставится знак вопроса, после которого в угловых скобках идет название группы:

(?<название_группы> ...)

Рассмотрим следующий пример:

```
> const exp = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
  const text = "Publication Date: 2024-10-10"
  const result = exp.exec(text);
  console.log(result.groups);
  console.log(result.groups.year);
  console.log(result.groups.month);
  console.log(result.groups.day);
  ▶ {year: '2024', month: '10', day: '10'}
  2024
  10
  10
```

Здесь регулярное выражение определяет три группы. Первая группа называется "year", вторая - "month" и третья "day". При получении результата мы можем обаться к каждой группе через свойство groups. Это свойство представляет объект, в котором свойства называются так же, как и группы, и содержат значения для каждой группы:

```
> console.log(result.groups);
  ▶ {year: '2024', month: '10', day: '10'}
```

Соответственно с помощью названия группы мы можем получить значение для определенной группы.

Утверждения

Утверждения или assertions позволяют получить подстроку, которая соответствует регулярному выражению и которая предваряется или, наоборот, не предваряется определенным выражением.

Положительное утверждение (когда подстрока должна предваряться другой подстрокой) определяется с помощью выражения:

(?<=...)

После знака равно = идет выражение, которым должна предваряться подстрока.

Отрицательное утверждение (когда подстрока НЕ должна предваряться другой подстрокой) определяется с помощью выражения:

(?<!...)

После восклицательного знака ! идет выражение, которым НЕ должна предваряться подстрока.

Возьмем простую задачу. Допустим у нас есть некоторая информация с какой-то суммой. Но эта сумма может быть определена в долларах, в евро, в рублях и так далее. Что-то наподобие:

```
> const text1 = "All costs: $10.53";
  const text2 = "All costs: €10.53";

  const exp = /\d+(\.\d*)?/;
  const result1 = exp.exec(text1);
  console.log(result1[0]);    // 10.53

  const result2 = exp.exec(text2);
  console.log(result2[0]);     // 10.53
10.53
10.53
```

Здесь мы видим, что и сумма в долларах () и сумма в евро соответствует нашему регулярному выражению. Но что, если мы хотим получить только сумму в долларах. Для этого применим положительное утверждение:

```
> const text1 = "All costs: $10.53";
  const text2 = "All costs: €10.53";

  const exp = /(?<=\$)\d+(\.\d*)?/;

  const result1 = exp.exec(text1);
  console.log(result1);   // ["10.53", ".53", index: 12, input: "All costs: $10.53", groups: undefined]

  const result2 = exp.exec(text2);
  console.log(result2);    // null
▶ (2) ['10.53', '.53', index: 12, input: 'All costs: $10.53', groups: undefined]
null
```

Группа (?<=\$) указывает, что перед строкой должен идти знак доллара $. Если его нет, то метод exec() не найдет соответствий и возвратит null.

Регулярные выражения в методах String

Ряд методов объекта String могут использовать регулярные выражения в качестве параметра.

Метод match

Для поиска всех соответствий в строке применяется метод **match()**:

```
> const initialText = "Он пришел домой и сделал домашнюю работу";
  const exp = /дом[а-я]*/gi;
  const result = initialText.match(exp);
  result.forEach(value => console.log(value));
  // или так
  // console.log(result[0]);
  // console.log(result[1]);
домой
домашнюю
```

Символ звездочки указывает на возможность наличия после строки "дом" неопределенного количества символов от а до я. В итоге в массиве result окажутся следующие слова:

домой

домашнюю

С одной стороны, этот метод похож на метод **exec()** объекта RegExp за тем исключением, что **exec()** возвращает только первое вхождение:

```
> const initialText = "Он пришел домой и сделал домашнюю работу";
  const exp = /дом[а-я]*/gi;
  const result2 = exp.exec(initialText);
  result2.forEach(value => console.log(value));
  домой
```

Разделение строки. Метод split

Метод split может использовать регулярные выражения для разделения строк. Например, разделим приложение по словам (а точнее по пробелам) с помощью метасимвола "\s":

```
> const initialText = "Сегодня была прекрасная погода";
  const exp = /\s/;
  const result = initialText.split(exp);
  result.forEach(value => console.log(value));
  Сегодня
  была
  прекрасная
  погода
```

Поиск в строке. Метод search

Метод search находит индекс первого включения соответствия в строке:

```
> const initialText = "hello world";
  const exp = /wor/;
  const result = initialText.search(exp);
  console.log(result); // 6
  6
```

Замена. Методы replace

Метод replace позволяет заменить все соответствия регулярному выражению определенной строкой. Первый параметр метода - регулярное выражение, а второй - на что заменяем совпадения. Пример замены:

```
> let menu = "Завтрак: каша, чай. Обед: суп, чай. Ужин: салат, чай.";
  const exp = /чай/gi;
  menu = menu.replace(exp, "кофе");
  console.log(menu);
  Завтрак: каша, кофе. Обед: суп, кофе. Ужин: салат, кофе.
```

Другая форма метода в качестве второго параметра принимает функцию замены. Эта функция принимает соответствие регулярному выражению и возвращает строку, на которое заменяется соответствие. Например, заменим чай на кофе только для завтрака:

```
> let menu = "Завтрак: каша, чай. Обед: суп, чай. Ужин: салат, чай.";
  const exp = /чай/gi;
  let i = 0;
  menu = menu.replace(exp, function(tee){
      if(i++ == 0) return "кофе";
      else return tee;
  });
  console.log(menu);  // Завтрак: каша, кофе. Обед: суп, чай. Ужин: салат, чай.
  Завтрак: каша, кофе. Обед: суп, чай. Ужин: салат, чай.
```

Для индикатора замены используем счетчик - переменную i. Если она равна 0, то производим замену. В остальных случаях возвращаем соответствие - строку "чай".

Более сложный случай. Пусть у нас есть следующий текст:

Publication Date: 2024-09-06

Updated on: 2024-19-14

Здесь у нас применяются даты в формате yyyy-MM-dd. Допустим, нам надо изменить формат дат на "dd.MM.yyyy". Для этого определим следующую программу:

```
> const exp = /\d{4}-\d{2}-\d{2}/g;
  let text = "Publication Date: 2024-09-06\nUpdated on: 2024-10-14"

  text = text.replace(exp, function(date){
      const arr = date.split("-");
      return `${arr[2]}.${arr[1]}.${arr[0]}`;
  });
  console.log(text);
  Publication Date: 06.09.2024
  Updated on: 14.10.2024
```

Здесь мы извлекаем все соответствия регулярному выражению /\d{4}-\d{2}-\d{2}/g. В методе replace в функции обратного вызова получаем соответствие через параметр date, с помощью функции **split()** разбиваем его на три части по разделителю-дефису:

```
const arr = date.split("-");
```

То есть у нас получаем массив из трех компонентов даты. И затем возвращаем дату в другом формате:

```
return `${arr[2]}.${arr[1]}.${arr[0]}`;
```

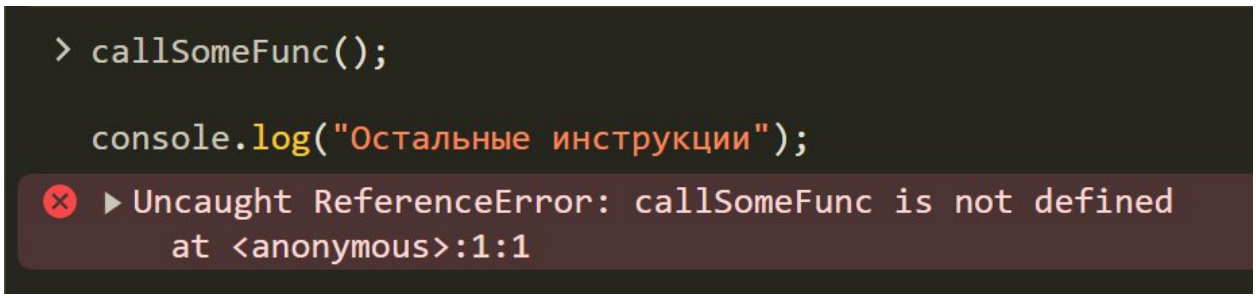
Задание

1. Предложите решение позволяющее осуществлять валидацию:
   - IPV4 -адреса.
   - Электронной почты.
   - Серии и номера паспорта.
   - URL-адреса начинающегося с http или https.
2. Дано: const text =
   'Visit our website at https://www.example.com for more information. For online shopping, go to https://shop.example.com.'
   Напишите выражение, которое извлечет из строки все URL адреса.

Занятие 5. Обработка ошибок

В процессе работы программы могут возникать различные ошибки, которые нарушают привычный ход программы и даже заставляют ее прервать выполнение. Язык JavaScript имеет инструменты для обработки таких ситуаций.

Простейшая ситуация - вызов функции, которой не существует:



Здесь вызывается функция callSomeFunc(), которая нигде не определена. Соответственно при вызове этой функции мы столкнемся с ошибкой:

Uncaught ReferenceError: callSomeFunc is not defined

Все остальные инструкции, которые идут после строки, на которой возникла ошибка, не выполняется. Программа заканчивает свою работу.

Ситуация может показаться искусственной, поскольку мы знаем, что функция callSomeFunc нигде не определена. Однако, когда мы имеем дело с большой программой, различные куски которой определяли разные разработчики, становится сложнее контролировать код. И таких ситуаций может быть много. Какие-то мы можем сами отследить и предупредить, а какие-то нет.

Для обработки подобных ситуаций JavaScript предоставляет конструкцию **try...catch...finally**, которая имеет следующее формальное определение:

```
try {
  //инструкции блока try
}
catch (error) {
  //инструкции блока catch
}
finally {
  //инструкции блока finally
}
```

После оператора **try** определяется блок кода. В этот блок помещаются инструкции, при выполнении которых может возникнуть потенциальная ошибка.

Затем идет оператор **catch**. После этого оператора в круглых скобках указывается название объекта, который будет содержать информацию об ошибке. И далее идет блок catch. Этот блок выполняется только при возникновении ошибки в блоке try.

После блока catch идет оператор **finally** со своим блоком инструкций. Этот блок выполняется в конце после блока try и catch вне зависимости, возникла ошибка или нет.

Стоит отметить, что только блок **try** является обязательным. А один из остальных блоков - **catch** или **finally** мы можем опустить. Однако один из этих блоков (не важно catch или finally) обязательно должен присутствовать. То есть мы можем использовать следующие варианты этой конструкции:

- try...catch
- try...finally
- try...catch...finally

Например, обработаем с помощью этой конструкцию предыдущую ситуацию с несуществующей функцией:

```
try{
    callSomeFunc();
    console.log("Конец блока try");
}
catch{
    console.log("Возникла ошибка!");
}
console.log("Остальные инструкции");
```

Итак, сначала выполняется блок **try**. Однако при выполнении первой же инструкции - вызова функции callSomeFunc() возникает ошибка. Это приведет к тому, что все последующие инструкции в блоке **try** НЕ будут выполняться. А управление перейдет к блоку **catch**. В этом блоке выводится сообщение, что возникла ошибка. После выполнения блока **catch** выполняются остальные инструкции программы. Таким образом, программа не прерывает свою работу при возникновении ошибки и продолжает свою работу.

Рассмотрим другой пример:

```
function callSomeFunc(){console.log("Функция callSomeFunc");}
try{
    callSomeFunc();
    console.log("Конец блока try");
}
catch(error){
    console.log("Возникла ошибка!");
}

console.log("Остальные инструкции");
Функция callSomeFunc
Конец блока try
Остальные инструкции
undefined
```

Теперь функция callSomeFunc() определена в программе, поэтому при вызове функции ошибки не произойдет, и блок **try** доработает до конца. А блок **catch** при отсутствии ошибки не будет выполняться.

Получение ошибки в блоке catch

В качестве в качестве параметра в блок catch передается объект с информацией об ошибке:

```
> try{
      callSomeFunc();
      console.log("Конец блока try");
  }
  catch(error){
      console.log("Возникла ошибка!");
      console.log(error);
  }
```

В этом случае мы получим консольный вывод на подобие следующего:

Возникла ошибка!

ReferenceError: callSomeFunc is not defined

    at index.html:35

Блок finally

Конструкция **try** также может содержать блок **finally**. Мы можем использовать этот блок вместе с блоком **catch** или вместо него. Блок **finally** выполняется вне зависимости, произошла ошибка или нет. Например:

```
> try{
      callSomeFunc();
      console.log("Конец блока try");
  }
  catch{
      console.log("Произошла ошибка");
  }
  finally{
      console.log("Блок finally")
  }

  console.log("Остальные инструкции");
```

Если мы уберем блок **catch** и оставим только блок **finally**, то ошибка не будет обработана, и программа завершится ошибкой. Однако блок **finally** все равно выполнится.

```
try{
    callSomeFunc();
    console.log("Конец блока try");
}
finally{
    console.log("Блок finally")
}

console.log("Остальные инструкции");
```

Консольный вывод программы:

Блок finally

Uncaught ReferenceError: callSomeFunc is not defined

Генерация ошибок и оператор throw

Интерпретатор JavaScript генерирует ошибки для ряда ситуаций, например, при вызове несуществующей функции, при повторном присвоении константе значения и т.д. Но при необходимости мы сами можем генерировать ошибки и определить условия, когда будет генерироваться ошибка.

Например, рассмотрим следующую ситуацию:

```
class Person{

    constructor(name, age){
        this.name = name;
        this.age = age;
    }
    print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
}
const tom = new Person("Tom", -123);
tom.print();    // Name: Tom  Age: -123
```

Класс Person описывает человека. В конструкторе класс получает значения для свойств name (имя) и age (возраст). Исходя из здравого смысла мы понимаем, что возраст не может быть отрицательным. Тем не менее пока, исходя из логики класса, ничего не мешает при создании объекта Person передать ему для возраста отрицательное значение. С точки зрения интерпретатора JavaScript ошибки нет, однако с точки логики и здравого смысла - это ошибка. Как исправить эту ситуацию? Есть различные способы, и один из них заключается в генерации исключения.

Для генерации исключения применяется оператор **throw**, после которого указывается информация об ошибке:

```
throw информация_об_ошибке;
```

Информация об ошибке может представлять любой объект.

Так, сгенерируем исключение при передаче в конструктор Person отрицательного значения для свойства age:

```
> class Person{

      constructor(name, age){
          if(age < 0) throw "Возраст должен быть положительным";
          this.name = name;
          this.age = age;
      }
      print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
  }
  const tom = new Person("Tom", -123);    // Uncaught Возраст должен быть положительным
  tom.print();
```

В итоге при вызове конструктора Person будет сгенерировано исключение и программа завершится ошибкой. А на консоли браузера мы увидим информацию об ошибке, которая указана после оператора throw:

Uncaught Возраст должен быть положительным

Как и в общем случае мы можем обработать эту ошибку с помощью блока **try...catch**:

```
> class Person{

      constructor(name, age){
          if(age < 0) throw "Возраст должен быть положительным";
          this.name = name;
          this.age = age;
      }
      print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
  }

  try{
      const tom = new Person("Tom", -123);    // Uncaught Возраст должен быть
  положительным
      tom.print();
  }
  catch(error){
      console.log("Произошла ошибка");
      console.log(error);     // Возраст должен быть положительным
  }
```

throw в try..catch..finally

Оператор **throw** может вызываться в различных контекстах, например, в том же блоке **try**:

```
> try{
       throw "Непредвиденная ошибка!";
  }
  catch(error){
      console.log(error); // Непредвиденная ошибка!
  }
```

Это может быть полезно по ряду причин. Во-первых, мы можем тут же обработать ошибку. Во-вторых, мы можем определить с помощью блока finally некоторые завершающие действия, которые будут выполняться, даже если сгенерирована ошибка. Например:

```
// класс условной базы данных
class Database{
    constructor(){
        this.data = ["Tom", "Sam", "Bob"];
    }
    // получение данных
    getItem(index){
        this.open();
        if(index > 0 && index < this.data.length)
            return this.data[index];
        else
            throw "Некорректный индекс";
        this.close();   // при генерации исключения эта строка не будет выполняться
    }
    // открытие бд
    open(){ console.log("Подключение к базе данных установлено"); }
    // закрытие бд
    close(){ console.log("Подключение к базе данных закрыто"); }
}

const db = new Database();
try {
    db.getItem(5);   // возвращаем полученный элемент
} catch(err) {
    console.error(err); // если произошла ошибка обрабатываем ее
}
```

Здесь определен класс Database - класс условной базы данных. Все данные хранятся в массиве data. Для взаимодействия с базой данных определены три метода. Методы open и close условно открывают и закрывают подключение к базе данных. Метод getItem получает по индексу элемент из массива data. Если же индекс некорректный, то генерируется ошибка. При этом до получения элемента по индексу метод getItem должен открыть подключение методом open, а после получения - закрыть методом close. Однако в примере выше при генерации ошибки закрытия подключения не произойдет:

```
else
    throw "Некорректный индекс";
this.close();   // при генерации исключения эта строка не будет выполняться
```

В итоге при передаче в метод getItem некорректного индекса консольный вывод программы будет следующим:

Подключение к базе данных установлено

Некорректный индекс

Однако, что делать, если нам все-таки надо вызвать метод close? Мы можем поместить его вызов в блок **finally**:

```
class Database{
    constructor(){
        this.data = ["Tom", "Sam", "Bob"];
    }
    // получение данных
    getItem(index){
        this.open();
        try{
            if(index > 0 && index < this.data.length)
                return this.data[index];
            else
                throw "Некорректный индекс";
        }
        finally{    // даже если сгенерирована ошибка, то этот блок выполняется
            this.close();   // при генерации исключения эта строка также будет выполняться
        }
    }
    // открытие бд
    open(){ console.log("Подключение к базе данных установлено"); }
    // закрытие бд
    close(){ console.log("Подключение к базе данных закрыто"); }
}

const db = new Database();
try {
    db.getItem(5);   // возвращаем полученный элемент
} catch(err) {
    console.error(err); // если произошла ошибка обрабатываем ее
}
```

И теперь консольный вызов будет иным:

Подключение к базе данных установлено

Подключение к базе данных закрыто

Некорректный индекс

Типы ошибок

В блоке **catch** мы можем получить информацию об ошибке, которая представляет объект. Все ошибки, которые генерируются интерпретатором JavaScript, предоставляют объект типа **Error**, который имеет ряд свойств:

- **message**: сообщение об ошибке
- **name**: тип ошибки

Стоит отметить, что отдельные браузеры поддерживают еще ряд свойств, но их поведение в зависимости от браузера может отличаться:

- **fileName**: название файла с кодом JavaScript, где произошла ошибка
- **lineNumber**: строка в файле, где произошла ошибка
- **columnNumber**: столбец в строке, где произошла ошибка

- **stack**: стек ошибки

Получим данные ошибки, например, при вызове несуществующей функции:

```
> try{
      callSomeFunc();
  }
  catch(error){
      console.log("Тип ошибки:", error.name);
      console.log("Ошибка:", error.message);
  }
```

Консольный вывод:

Тип ошибки: ReferenceError

Ошибка: callSomeFunc is not defined

Выше мы рассмотрели, что генерируемая интерпретатором ошибка представляет тип Error, однако при вызове несуществующей функции генерируется ошибка типа **ReferenceError**. Дело в том, что тип Error представляет общий тип ошибок. В то же время есть конкретные типы ошибок для определенных ситуаций:

- **EvalError**: представляет ошибку, которая генерируется при выполнении глобальной функции eval()
- **RangeError**: ошибка генерируется, если параметр или переменная, представляют число, которое находится вне некоторого допустимого диапазона
- **ReferenceError**: ошибка генерируется при обращении к несуществующей ссылке
- **SyntaxError**: представляет ошибку синтаксиса
- **TypeError**: ошибка генерируется, если значение переменной или параметра представляют некорректный тип или пр попытке изменить значение, которое нельзя изменять
- **URIError**: ошибка генерируется при передаче функциям encodeURI() и decodeURI() некорректных значений
- **AggregateError**: предоставляет ошибку, которая объединяет несколько возникших ошибок

Например, при попытке присвоить константе второй раз значение, генерируется ошибка TypeError:

```
> try{
      const num = 9;
      num = 7;
  }
  catch(error){
      console.log(error.name);        // TypeError
      console.log(error.message);     // Assignment to constant variable.
  }
```

Применение типов ошибок

При генерации ошибок мы можем использовать встроенные типы ошибок. Например:

```
> class Person{

      constructor(name, age){
          if(age < 0) throw new Error("Возраст должен быть положительным");
          this.name = name;
          this.age = age;
      }
      print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
  }

  try{
      const tom = new Person("Tom", -45);
      tom.print();
  }
  catch(error){
      console.log(error.message); // Возраст должен быть положительным
  }
```

Здесь конструктор класса Person принимает значения для имени и возраста человека. Если передан отрицательный возраст, то генерируем ошибку в виде объекта Error. В качестве параметра в конструктор Error передается сообщение об ошибке:

```
> if(age < 0) throw new Error("Возраст должен быть положительным");
```

В итоге при обработке исключения в блоке catch мы сможем получить переданное сообщение об ошибке.

Все остальные типы ошибок в качестве первого параметра в конструкторе также принимают сообщение об ошибке. Так, сгенерируем несколько типов ошибок:

```
> class Person{

      constructor(pName, pAge){

          const age = parseInt(pAge);
          if(isNaN(age))  throw new TypeError("Возраст должен представлять число");
          if(age < 0 || age > 120) throw new RangeError("Возраст должен быть больше 0 и
  меньше 120");
          this.name = pName;
          this.age = age;
      }
      print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
  }
```

Поскольку для возраста можно передать не только число, но и вообще какое угодно значение, то вначале мы пытаемся преобразовать это значение в число с помощью функции parseInt():

```
> const age = parseInt(pAge);
  if(isNaN(age))  throw new TypeError("Возраст должен представлять число");
```

Далее с помощью функции isNaN(age) проверяем, является полученное число числом. Если age - НЕ число, то данная функция возвращает true. Поэтому генерируется ошибка типа TypeError.

Затем проверяем, что полученное число входит в допустимый диапазон. Если же не входит, генерируем ошибку типа RangeError:

```
> if(age < 0 || age > 120) throw new RangeError("Возраст должен быть больше 0 и меньше
  120");
```

Проверим генерацию исключений:

```
> try{
      const tom = new Person("Tom", -45);
  }
  catch(error){
      console.log(error.message); // Возраст должен быть больше 0 и меньше 120
  }

  try{
      const bob = new Person("Bob", "bla bla");
  }
  catch(error){
      console.log(error.message); // Возраст должен представлять число
  }

  try{
      const sam = new Person("Sam", 23);
      sam.print();    // Name: Sam  Age: 23
  }
  catch(error){
      console.log(error.message);
  }
```

Консольный вывод:

Возраст должен быть больше 0 и меньше 120

Возраст должен представлять число

Name: Sam Age: 23

Обработка нескольких типов ошибок

При выполнении одного и то же кода могут генерироваться ошибки разных типов. И иногда бывает необходимо разграничить обработку разных типов. В этом случае мы можем проверять тип возникшей ошибки. Например, пример выше с классом Person:

```
class Person{

    constructor(pName, pAge){

        const age = parseInt(pAge);
        if(isNaN(age))  throw new TypeError("Возраст должен представлять число");
        if(age < 0 || age > 120) throw new RangeError("Возраст должен быть больше 0 и
меньше 120");
        this.name = pName;
        this.age = age;
    }
    print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
}

try{
    const tom = new Person("Tom", -45);
    const bob = new Person("Bob", "bla bla");
}
catch(error){
    if (error instanceof TypeError) {
        console.log("Некорректный тип данных.");
    } else if (error instanceof RangeError) {
        console.log("Недопустимое значение");
    }
    console.log(error.message);
}
```

Создание своих типов ошибок

Мы не ограничены встроенными только встроенными типами ошибок и при необходимости можем создавать свои типы ошибок, предназначенные для каких-то конкретных ситуаций. Например:

```
class PersonError extends Error {
  constructor(value, ...params) {
    // остальные параметры передаем в конструктор базового класса
    super(...params)
    this.name = "PersonError"
    this.argument = value;
  }
}

class Person{

    constructor(pName, pAge){

        const age = parseInt(pAge);
        if(isNaN(age))  throw new PersonError(pAge, "Возраст должен представлять
число");
        if(age < 0 || age > 120) throw new PersonError(pAge, "Возраст должен быть
больше 0 и меньше 120");
        this.name = pName;
        this.age = age;
    }
    print(){ console.log(`Name: ${this.name}  Age: ${this.age}`);}
}

try{
    //const tom = new Person("Tom", -45);
    const bob = new Person("Bob", "bla bla");
}
catch(error){
    if (error instanceof PersonError) {
        console.log("Ошибка типа Person. Некорректное значение:", error.argument);
    }
    console.log(error.message);
}
```

Консольный вывод

Ошибка типа Person. Некорректное значение: bla bla

Возраст должен представлять число

Для представления ошибки класса Person здесь определен тип PersonError, который наследуется от класса Error:

```
class PersonError extends Error {
  constructor(value, ...params) {
    // остальные параметры передаем в конструктор базового класса
    super(...params)
    this.name = "PersonError"
    this.argument = value;
  }
}
```

В конструкторе мы определяем дополнительное свойство - argument. Оно будет хранить значение, которое вызвало ошибку. С помощью параметра value конструктора получаем это значение. Кроме того, переопреляем имя типа с помощью свойства this.name.

В классе Person используем этот тип, передавая в конструктор PersonError соответствующие значения:

```
> if(isNaN(age))  throw new PersonError(pAge, "Возраст должен представлять число");
  if(age < 0 || age > 120) throw new PersonError(pAge, "Возраст должен быть больше 0 и
  меньше 120");
```

Затем при обработки исключения мы можем проверить тип, и если он представляет класс PersonError, то обратиться к его свойству argument:

```
> catch(error){
      if (error instanceof PersonError) {
          console.log("Ошибка типа Person. Некорректное значение:", error.argument);
      }
      console.log(error.message);
  }
```

Задание

1. Предложите пример применения конструкции **try...catch...finally.**
2. Предложите пример принудительной генерации исключения с помощью **throw.**

Занятие 6. Итераторы

Итераторы представляют абстракцию для перебора наборов данных и применяются для организации последовательного доступа к элементам наборов данных - массивам, объектам Set, Map, строкам и т.д. Так, благодаря итераторам мы можем перебрать набор данных (например, массив) с помощью цикла **for-of**:

```
> const people = ["Tom", "Bob", "Sam"];
  for(const person of people){
      console.log(person);
  }
  Tom
  Bob
  Sam
```

В цикле **for-of** справа от оператора **of** указывается набор данных или перебираемый объект (то, что называется **Iterable**), из которого в цикле мы можем получить отдельные элементы. Но эта возможность перебора некоторого объекта, как, например, массива в примере выше, реализуются благодаря тому, что эти объекты применяют итераторы. Рассмотрим подробнее, что представляют итераторы и как можно создать свой итератор.

Получение итератора

Любой итерируемый объект (например, массив, Map, Set и т.д.) хранит в свойстве Symbol.iterator функцию, которая возвращает связанный с объектом итератор:

```
> const people = ["Tom", "Bob", "Sam"];
  // получаем итератор массива
  const iterator = people[Symbol.iterator]();
  console.log(iterator);  // Array Iterator {}
  ▼ Array Iterator {} i
    ▶ constructor: ƒ Iterator()
    ▶ [[Prototype]]: Array Iterator
```

Здесь получаем итератор массива, поэтому на консоль будет выведено что-то наподобие Array Iterator {}

Другой пример - строка тоже представляет перебираемый объект, которую можно перебрать посимвольно:

```
> const username = "Tom";
  for(char of username){
      console.log(char);
  }
  T
  o
  m
```

Соответственно для строки мы тоже можем получить итератор:

```
> const username = "Tom";
  // получаем итератор строки
  const iterator = username[Symbol.iterator]();
  console.log(iterator);  // StringIterator {}
  ▶ StringIterator {}
```

Итератор строки представляет тип StringIterator. Аналогичным образом можно получать итераторы и для других типов перебираемых объектов.

Стоит отметить, что у различных типов могут быть различные дополнительные методы для получения итератора. Например, у массивов есть метод **entries()**, который также возвращает итератор массива:

```
> const people = ["Tom", "Bob", "Sam"];
  console.log(people.entries()); // Array Iterator {}
  ▶ Array Iterator {}
```

Метод next итераторов

Итераторы предоставляют метод **next()**, который возвращает объект с двумя свойствами: **value** и **done.**

Свойство **value** хранит собственно значение текущего перебираемого элемента. А свойство **done** указывает, есть ли еще в коллекции объекты, доступные для перебора. Если в наборе еще есть элементы, то свойство done равно false Если же доступных элементов для перебора больше нет, то это свойство равно **true**, а метод next() возвращает объект {done: true}

Например:

```
> const people = ["Tom", "Bob", "Sam"];
  const iter = people[Symbol.iterator]();
  const result = iter.next();
  console.log(result);    // {value: "Tom", done: false}
  ▶ {value: 'Tom', done: false}
```

Здесь мы видим, что текущий объект представляет строку "Tom", а значение done: false указывает, что в массиве еще есть элементы для перебора.

Мы можем последовательно несколько раз вызвать метод next() для получения других элементов массива:

```
> const people = ["Tom", "Bob", "Sam"];
  const iter = people[Symbol.iterator]();
  console.log(iter.next());    // {value: "Tom", done: false}
  console.log(iter.next());    // {value: "Bob", done: false}
  console.log(iter.next());    // {value: "Sam", done: false}
  console.log(iter.next());    // {value: undefined, done: true}
  ▶ {value: 'Tom', done: false}
  ▶ {value: 'Bob', done: false}
  ▶ {value: 'Sam', done: false}
  ▶ {value: undefined, done: true}
```

При каждом новом вызове метода next() мы получаем из массива следующий объект. А когда объектов для перебора больше не останется, то свойство done будет равно true.

Используя метод next(), мы сами можем перебрать все объекты массива:

```
> const people = ["Tom", "Bob", "Sam"];
  const iter = people[Symbol.iterator]();
  while(!(item = iter.next()).done){
      console.log(item.value);
  }
  Tom
  Bob
  Sam
```

Здесь в цикле while из метода next() итератора получаем текущий объект в переменную item: item = items.next() И смотрим на ее свойство done - если оно равно false (то есть в наборе еще есть элементы), то продолжаем цикл. В цикле обращаемся к свойству value полученного объекта

Надо отметить, что все коллекции, которые возвращают итераторы, поддерживают перебор с помощью цикла **for...of**, который как раз и использует итератор для получения элементов.

Создание своего итератора

Для примера реализуем итератор, который перебирает массив с конца:

```
> const people = ["Tom", "Bob", "Sam"];

  function reverseArrayIterator(array) {
      let count = array.length;
      return {
          next: function(){
              if (count > 0) {
                  return {
                      value: array[--count],
                      done: false
                  };
              }
              else {
                  return {
                      value: undefined,
                      done: true
                  };
              }
          }
      }
  };
  const iter = reverseArrayIterator(people);
  while(!(item = iter.next()).done){
      console.log(item.value);
  }
  Sam
  Bob
  Tom
```

Здесь сначала инициализируется переменная count, которая количество перебранных элементов массива. Первоначально переменная имеет значение, равное длине массива.

Далее функция возвращает объект итератора. Его метод next() реализует поведение итерации: если счетчик count больше 0 (то есть имеются еще элементы для перебора), то next() возвращает объект, свойство done которого имеет значение false (поскольку итератор еще не достиг конца или точнее начала массива), а свойство value содержит соответствующий элемент из массива, на который указывает переменная count после декремента.

Когда переменная count станет равна 0 (т. е. итератор достиг конца), next() возвращает объект, у которого свойство done имеет значение true, а свойство value имеет значение undefined.

Таким образом, мы получим итератор, который перебирает объекты массива с конца.

Однако при выполнении цикла **for..of** элементы массива по прежнему перебираются с начала. Применим наш итератор глобально, чтобы он также использовался в цикле **for..of**:

```
> const people = ["Tom", "Bob", "Sam"];

  function reverseArrayIterator() {
      const array = this;
      let count = array.length;
      return {
          next: function(){
              if (count > 0) {
                  return {
                      value: array[--count],
                      done: false
                  };
              }
              else {
                  return {
                      value: undefined,
                      done: true
                  };
              }
          }
      }
  };
  // меняем итератор для массива people
  people[Symbol.iterator]=reverseArrayIterator;
  for(person of people){
      console.log(person);
  }
  Sam
  Bob
  Tom
```

Здесь сделано два ключевых изменения. Во-первых, нам надо внутри итератора получить текущий объект через **this**:

const array = this;

Созданную функцию итератора надо присвоить свойству Symbol.iterator:

people[Symbol.iterator]=reverseArrayIterator

Создание итерируемых объектов

Разные объекты могут иметь свою собственную реализацию итератора. И при необходимости мы можем определить объект со своим итератором. Применение итераторов предоставляет нам способ создать объект, который будет вести себя как коллекция элементов

Для создания перебираемого объекта нам надо определить в объекта метод **[Symbol.iterator]()**. Этот метод собственно и будет представлять итератор:

```
> const iterable = {
    [Symbol.iterator]() {
      return {
        next() {
              // если еще есть элементы
              return { value: ..., done: false };
              // если больше нет элементов
              return { value: undefined, done: true };
        }
      };
    }
  };
```

Метод **[Symbol.iterator]()** возвращает объект, который имеет метод **next()**. Этот метод возвращает объект с двумя свойствами value и done.

Если в нашем объекте есть элементы, то свойство value содержит собственно значение элемента, а свойство done равно false.

Если доступных элементов больше нет, то свойство done равно true.

Например, реализуем простейший объект с итератором, который возвращает некоторый набор чисел:

```
> const iterable = {
    [Symbol.iterator]() {
      return {
        current: 1,
        end: 3,
        next() {
          if (this.current <= this.end) {
            return { value: this.current++, done: false };
          }
          return { done: true };
        }
      };
    }
  };
```

Здесь итератор фактически возвращает числа от 1 до 3. Для отслеживания текущего элемента в объекте, который возвращается методом, определены два свойства: current и end.

Свойство current хранит значение текущего элемента. А свойство end задает предел. То есть в данном случае итератор возвращает числа от 1 до 3.

В методе next(), если текущее значение меньше или равно предельному значению, возвращаем объект

return { value: this.current++, done: false };

Инкремент this.current++ приведет к тому, что при следующем вызове метода next значение current будет на единицу больше.

Если достигнут предел, то возвращаем объект

return { done: true };

Это будет указывать, что объектов больше нет.

Получим из итератора возвращаемые им элементы:

```
> const myIterator = iterable[Symbol.iterator](); // получаем итератор
  console.log(myIterator.next()); // {value: 1, done: false}
  console.log(myIterator.next()); // {value: 2, done: false}
  console.log(myIterator.next()); // {value: 3, done: false}
  console.log(myIterator.next()); // {done: true}
  ▶ {value: 1, done: false}
  ▶ {value: 2, done: false}
  ▶ {value: 3, done: false}
  ▶ {done: true}
```

Здесь сначала получаем итератор в константу myIterator. Затем при обращении к ее методу next() последовательно получаем все элементы. При четвертом вызове метода next условный перебор элементов в итераторе закончен, и метод возвращает объект {done: true}.

Однако если мы хотим перебрать наш объект и получить из него его элементы, то нам не надо обращаться к методу next(). Поскольку объект iterable реализует итератор, то его можно перебрать с помощью цикла **for-of**:

```
> const iterable = {
    [Symbol.iterator]() {
      return {
        current: 1,
        end: 3,
        next() {
          if (this.current <= this.end) {
            return { value: this.current++, done: false };
          }
          return { done: true };
        }
      };
    }
  };
  for (const value of iterable) {
    console.log(value);
  }
  1
  2
  3
```

Цикл for-of автоматически обращается к методу next() и извлекает значение.

Рассмотрим еще один пример:

```
// объект-компания
const company = {
    // массив работников
    employees: [
        {name: "Tom", age: 39, position: "Senior Developer"},
        {name: "Bob", age: 43, position: "Middle Developer"},
        {name: "Sam", age: 28, position: "Junior Developer"},
    ]
};
// устанавливаем итератор
company[Symbol.iterator] = function() {
    const array = this.employees; // получаем массив работников
    let current = 0;
    return {
      next() {
        if (current < array.length) {
          return { value: array[current++].name, done: false };
        }
        return { value:undefined, done: true };
      }
    };
  };
for (const employee of company) {
  console.log(employee);
}
```

```
Tom
Bob
Sam
```

Здесь объект company представляет условную компанию, в которой есть массив работников - массив employee. Допустим, с помощью итератора мы хотим получать имя каждого работника. Для этого для объекта company устанавливаем функцию итератора, которая перебирает все элементы из массива employees.

Занятие 7. Коллекции

Множества Set

Язык JS поддерживает достаточно большое количество контейнеров, среди них словарь Map, хеш-таблица HashTable, множество Set. Рассмотрим основы работы с множеством.

Множества (sets) представляют структуру данных, которая может хранить только уникальные значения. В JavaScript функционал множества определяет объект **Set**. Для создания множества применяется конструктор этого объекта:

```
> const mySet = new Set();
```

Также можно передать в конструктор массив значений, которыми будет инициализировано множество:

```
> const arr = [1, 1, 2, 3, 4, 5, 2, 4];
  const numbers = new Set(arr);
  console.log(numbers);           // Set(5) {1, 2, 3, 4, 5}
  ▶ Set(5) {1, 2, 3, 4, 5}
```

В данном случае в множество передаются данные из массива. Однако поскольку множество может хранить только уникальные значения, то при его создании повторяющиеся значения, которые есть в массиве, удаляются.

Для упрощения создания набора мы можем сразу передать массив в конструктор Set:

```
> const numbers = new Set([1, 2, 3, 4, 5]);
  console.log(numbers);       // Set(5) {1, 2, 3, 4, 5}
  ▶ Set(5) {1, 2, 3, 4, 5}
```

Размер набора

Для проверки количества элементов можно использовать свойство **size**.

```
> const numbers = new Set([1, 1, 2, 3, 4, 5, 2, 4]);
  console.log(numbers.size);      // 5
  5
```

Добавление

Для добавления применяется метод **add()**. Его результатом является измененное множество:

```
> const numbers = new Set();
  numbers.add(1);
  numbers.add(3);
  numbers.add(5);
  numbers.add(3);     // не добавляется
  numbers.add(1);     // не добавляется
  console.log(numbers);       // Set(3) {1, 3, 5}
  ▶ Set(3) {1, 3, 5}
```

При этом, поскольку множество хранит только уникальные значения, то добавление элементов, которые уже в нем есть, не имеет смысла.

Так как метод add возвращает ссылку на это же множество, то мы можем вызывать методы по цепочке:

```
> const numbers = new Set();
  numbers.add(1).add(3).add(5);
  console.log(numbers);       // Set(3) {1, 3, 5}
  ▶ Set(3) {1, 3, 5}
```

Удаление

Для удаления элементов применяется метод **delete()**:

```
> const numbers = new Set([1, 3, 5]);
  numbers.delete(3);
  console.log(numbers);       // Set(2) {1, 5}
  ▶ Set(2) {1, 5}
```

Причем данный метод возвращает булевое значение: true - если элемент удален и false - если удаление не произошло (например, когда удаляемого элемента нет в множестве):

```
> const numbers = new Set([1, 3, 5]);

  let isDeleted = numbers.delete(3);
  console.log(isDeleted);          // true
  isDeleted = numbers.delete(54);
  console.log(isDeleted);          // false

  true

  false
```

Если необходимо удалить вообще все элементы из множества, то применяется метод **clear()**:

```
> let numbers = new Set();
  const numbers = new Set([1, 3, 5]);
  numbers.clear();
  console.log(numbers);            // Set(0) {}
```

Проверка наличия элемента

Если нужно проверить, если ли элемент в множестве, то используется метод **has()**. Если элемент есть, то метод возвращает true, иначе возвращает false:

```
> const numbers = new Set([1, 3, 5]);
  console.log(numbers.has(3));    // true
  console.log(numbers.has(32));   // false

  true

  false
```

Перебор множества

Для перебора элементов множества применяется метод **forEach()**:

```
> const numbers = new Set([1, 2, 3, 5]);

  numbers.forEach(function(value1, value2, set){
      console.log(value1);
  })

  1

  2

  3

  5
```

Для совместимости с массивами, которые тоже имеют метод forEach, в данный метод передается функция обратного вызова, которая принимает три параметра. Непосредственно для множества

первый и второй параметры представляют текущий перебираемый элемент, а третий параметр - перебираемое множество. В данном случае применяется только первый параметр.

Также для перебора множества можно использовать цикл for...of:

```
> const numbers = new Set([1, 2, 3, 5]);

  for(n of numbers){
      console.log(n);
  }
  1
  2
  3
  5
```

Получение итератора

Также у объекта **Set** есть ряд методов, которые возвращают итератор, а точнее объект **SetIterator**. Это методы values(), keys(), entries():

```
> const numbers = new Set([1, 2, 3, 5]);

  console.log(numbers.values());  // SetIterator {1, 2, 3, 5}
  console.log(numbers.keys());    // SetIterator {1, 2, 3, 5}
  console.log(numbers.entries()); // SetIterator {1 => 1, 2 => 2, 3 => 3, 5 => 5}
  ▶ SetIterator {1, 2, 3, 5}
  ▶ SetIterator {1, 2, 3, 5}
  ▶ SetIterator {1 => 1, 2 => 2, 3 => 3, 5 => 5}
```

Соответственно возвращаемый итератор мы можем использовать для получения объектов множества:

```
> const people = new Set(["Tom", "Bob", "Sam"]);

  const iterator = people.values();

  console.log(iterator.next());   // {value: "Tom", done: false}
  console.log(iterator.next());   // {value: "Bob", done: false}
  console.log(iterator.next());   // {value: "Sam", done: false}
  console.log(iterator.next());   // {value: undefined, done: true}
  ▶ {value: 'Tom', done: false}
  ▶ {value: 'Bob', done: false}
  ▶ {value: 'Sam', done: false}
  ▶ {value: undefined, done: true}
```

Удаление из контейнера повторяющихся элементов

Ограничения объекта Set - хранения уникальных значений позволяет эффективно его применять в ряде операций. Например, удаление из массива повторяющихся элементов:

```
> const peopleArray = ["Tom", "Bob", "Sam", "Alice", "Sam", "Kate", "Tom"];
  const peopleSet = new Set(peopleArray);
  const newPeopleArray = Array.from(peopleSet);

  console.log(newPeopleArray);    // ["Tom", "Bob", "Sam", "Alice", "Kate"]
  ▶ (5) ['Tom', 'Bob', 'Sam', 'Alice', 'Kate']
```

Здесь для создания нового массива с неповторяющимися элементами применяется функция **Array.from()**, которая в качестве аргумента получает объект Set.

Задание

1. Самостоятельно изучите возможности контейнера map (например: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map).
2. Решите задачи:

Задача 1

Допустим, у нас есть массив arr.

Создайте функцию unique(arr), которая вернёт массив уникальных, не повторяющихся значений массива arr.

```
function unique(arr) {
  /* ваш код */
}

let values = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(values) ); // Hare,Krishna,:-O
```

Задача 2

Анаграммы – это слова, у которых те же буквы в том же количестве, но они располагаются в другом порядке. Например:

nap - pan

ear - are - era

cheaters - hectares - teachers

Напишите функцию aclean(arr), которая возвращает массив слов, очищенный от анаграмм.

Например:

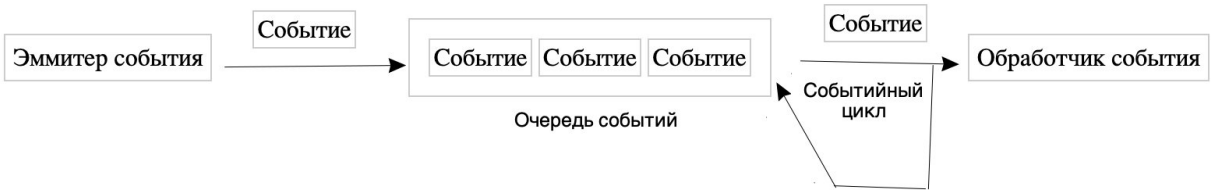
```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert( aclean(arr) ); // "nap,teachers,ear" или "PAN,cheaters,era"
```

Из каждой группы анаграмм должно остаться только одно слово, не важно какое.

Занятие 8. События мышки и клавиатуры

Для взаимодействия с пользователем в JavaScript определен механизм событий. Например, когда пользователь нажимает кнопку, то возникает событие нажатия кнопки. Аналогично, когда пользователь вводит в текстовое поле текст, возникает событие этого текстового поля. В коде JavaScript мы можем определить возникновение события и как-то его обработать.

Вкратце общий механизм выглядит следующим образом. Сначала происходит событие, например, пользователь нажал на кнопку. Объект, который сгенерировал событие, еще называется эмиттером/эмитентом события. После того как произошло событие, оно помещается в очередь событий (event queue), что гарантирует, что события, которые были сгенерированы первыми, также будут обработаны в первую очередь. Цикл событий (event loop) постоянно проверяет, есть ли новое событие в очереди событий, и если оно есть, соответствующее событие пересылается обработчикам событий (event handler). В JavaScript эти обработчики событий представляют собой простые функции, которые позволяют отреагировать на возникшее событие. Подобный подход еще называют событийным программированием (event-driven programming).



Если для события не определено обработчиков, то для него выполняется действие, которое определено браузером по умолчанию.

В JavaScript есть следующие типы событий:

- События мыши (перемещение курсора, нажатие мыши и т.д.)
- События клавиатуры (нажатие или отпускание клавиши клавиатуры)
- События жизненного цикла элементов (например, событие загрузки веб-станицы)
- События элементов форм (нажатие кнопки на форме, выбор элемента в выпадающем списке и т.д.)
- События, возникающие при изменении элементов DOM
- События, возникающие при касании на сенсорных экранах
- События, возникающие при возникновении ошибок

Стоит отметить, что применение событий не ограничены только графическим интерфейсом, однако графический интерфейс - наиболее показательная сфера применения событий.

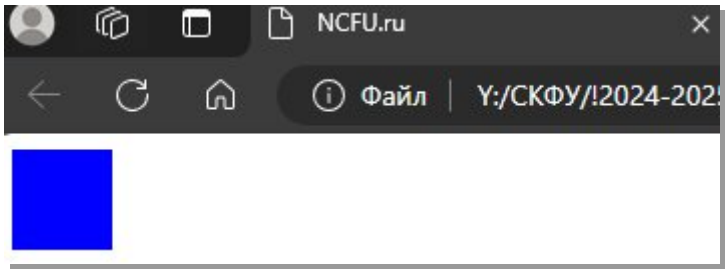
Рассмотрим простейшую обработку событий. Например, на веб-странице у нас есть следующий элемент div:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <div id="rect" onclick="console.log('Clicked!')"
     style="width:50px;height:50px;background-color:blue;"></div>
</body>
</html>
```

Здесь определен обычный блок div, который имеет атрибут **onclick**, который задает **обработчик события** нажатия на блок div. То есть, чтобы обработать какое-либо событие, нам надо определить для него обработчик. Обработчик представляет собой код на языке JavaScript. В данном случае обработчик просто выводит строку на консоль:

console.log('Clicked!')

И при нажатии на блок мы сможем увидеть в консоли браузера соответствующее сообщение:



События мыши

Событие	Описание	Объект события
click	возникает при нажатии указателем мыши на элемент	MouseEvent
dblclick	возникает при двойном нажатии указателем мыши на элемент	MouseEvent
contextmenu	возникает при открытии контекстного меню (правой кнопкой мыши)	MouseEvent
mousedown	возникает при нахождении указателя мыши на элементе, когда кнопка мыши находится в нажатом состоянии	MouseEvent
mouseup	возникает при нахождении указателя мыши на элементе во время отпускания кнопки мыши	MouseEvent
mousemove	возникает при прохождении указателя мыши над элементом	MouseEvent
mouseover	возникает при вхождении указателя мыши в границы элемента	MouseEvent
mouseout	возникает, когда указатель мыши выходит за пределы элемента	MouseEvent
mouseenter	возникает при вхождении указателя мыши в границы элемента	MouseEvent
mouseleave	возникает, когда указатель мыши выходит за пределы элемента	MouseEvent

События клавиатуры

Событие	**Описание**	**Объект события**
keydown	возникает при нажатии клавиши клавиатуры и длится, пока нажата клавиша	KeyboardEvent
keyup	возникает при отпускании клавиши клавиатуры	KeyboardEvent
keypress	возникает при нажатии клавиши клавиатуры, но после события keydown и до события keyup.	KeyboardEvent

События элементов форм

Событие	Описание	Объект события
input	возникает при изменении текста в элементах <input> и textarea или в элемент с атрибутом contenteditable	Event
change	возникает при изменении значения в списках, флажках (checkbox) или радиокнопках	Event
submit	возникает при отправке формы	Event
reset	возникает при сбросе формы (через кнопку reset)	Event

События фокусировки:

Событие	**Описание**	**Объект события**
focus	возникает при получении фокуса	FocusEvent
blur	возникает при потере фокуса	FocusEvent
focusin	возникает при получении фокуса (в отличие от события focus, это событие поднимающееся. Про поднимающиеся и опускающиеся события далее)	FocusEvent
focusout	возникает при потере фокуса(это событие поднимающееся в отличие от события blur)	FocusEvent

Занятие 9. Общие события интерфейса

Событие	Описание	Объект события
load	возникает при загрузке веб-страницы	UIEvent
unload	возникает при выгрузке веб-страницы (например, когда запрошена страница по новому адресу)	UIEvent
abort	возникает при отмене загрузки ресурса	UIEvent
Error	возникает при генерации ошибки при загрузке страницы (например, ошибка в коде JavaScript)	UIEvent
select	возникает при выделении текст на странице	UIEvent
resize	возникает при изменении размеров окна браузера	UIEvent
scroll	возникает при прокрутке	UIEvent
beforeunload	возникает непосредственно перед выгрузкой страницы	BeforeUnloadEvent
DOMContentLoaded	возникает при полной загрузке дерева DOM	Event
cut	возникает при вырезании текста из поля ввода (например, с помощью Ctrl+X)	ClipboardEvent
copy	возникает при копировании текста из поля ввода (например, с помощью Ctrl+C)	ClipboardEvent
paste	возникает при вставке текста в поле ввода (например, с помощью Ctrl+V)	ClipboardEvent
select	возникает при выделении текста в поле ввода	ClipboardEvent

События мобильных устройств и других устройств с сенсорным экраном

Событие	Описание	Объект события
orientationchange	возникает при изменении ориентации устройства	Event
deviceorientation	возникает, когда появляются новые данные об ориентации устройства	DeviceOrientationEvent
devicemotion	возникает с регулярными интервалами и указывает на силу ускорения, действующую на конечное устройство	DeviceMotionEvent
touchstart	возникает при касании дисплея	TouchEvent
touchend	возникает, когда палец убран с дисплея (касание завершилось)	TouchEvent
touchmove	возникает при движении пальцем по сенсорному дисплею	TouchEvent
touchcancel	возникает при прерывании отслеживания касаний	TouchEvent

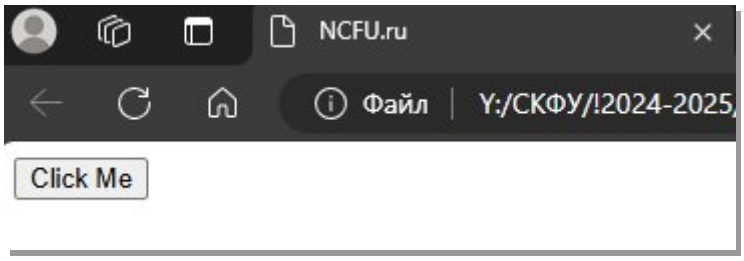
Если в коде JavaScript возникает событие, то его обрабатывает связанный с этим событием обработчик. Рассмотрим, как определять обработчики событий.

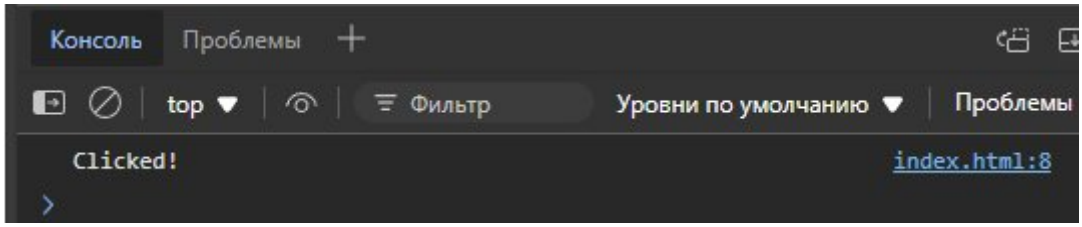
Встроенные обработчики

Самый простой способ определения обработчиков событий - их установка в коде html. Это так называемые встроенные обработчики или inline-обработчики, которые определяются в коде элемента с помощью атрибутов. Подобные атрибуты начинаются с префикса **on**. Например, у многих html-элементов есть атрибут **on**click, который определяет обработчик нажатия элемента. Посмотрим на примере кнопки:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <button onclick="console.log('Clicked!')">Click Me</div>
</body>
</html>
```

С помощью атрибута onclick="console.log('Clicked!')" к кнопке прикрепляется обработчик ее нажатия. Этот обработчик состоит из одной инструкции JavaScript - console.log("Clicked!"), которая выводит сообщение на консоль. Таким образом, при нажатии на кнопку сработает событие нажатия, и будет выполняться обработчик из атрибута onclick:





Можно даже определить несколько инструкций подобным образом:

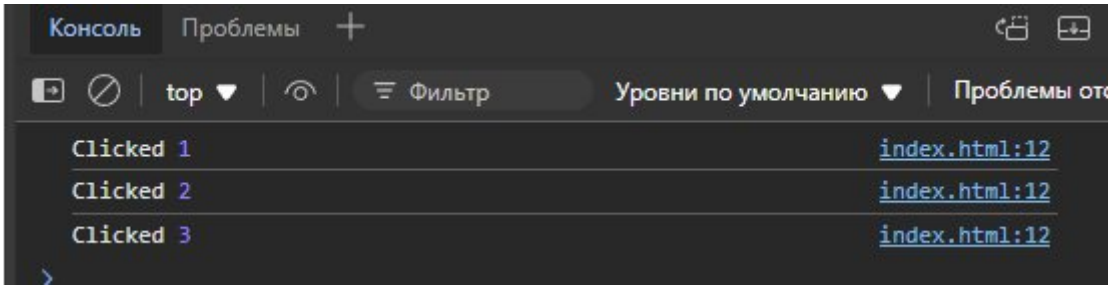
<button onclick=

"console.log('Hello');console.log('Clicked!')">Click Me

Но, очевидно, что это не самый удобный способ. Но также можно вынести все инструкции в отдельную функцию JavaScript. А атрибуту onclick передать вызов этой функции:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <button onclick="btn_click()">Click Me</div>
    <script>
    let clicks = 0;
    function btn_click(){
        console.log("Clicked", ++clicks);
    }
    </script>
</body>
</html>
```

Теперь по нажатию кнопки будет вызываться функция btn_click, которая определена в коде JavaScript.



Хотя этот подход прекрасно работает, но он имеет недостатки:

- Код html смешивается с кодом JavaScript, в связи с чем становится труднее разрабатывать, отлаживать и поддерживать приложение
- Обработчики событий можно задать только для уже созданных на веб-странице элементов. Динамически создаваемые элементы в этом случае лишаются возможности обработки событий
- К элементу для одного события может быть прикреплен только один обработчик
- Нельзя удалить обработчик без изменения кода

Свойства обработчиков событий

Проблемы, которые возникают при использовании встроенных обработчиков, были призваны решить свойства обработчиков. Подобно тому, как у html-элементов есть атрибуты для обработчиков, так и в коде javascript у элементов DOM мы можем получить свойства обработчиков, которые соответствуют атрибутам:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
<button id="btn">Click Me</div>
    <script>
    let clicks = 0;
    function btn_click(){
        console.log("Clicked", ++clicks);
    }
    document.getElementById("btn").onclick = btn_click;
    </script>
</body>
</html>
```

В итоге нам достаточно взять свойство **onclick** и присвоить ему функцию, используемую в качестве обработчика. За счет этого код html отделяется от кода javascript.

Слушатели событий

Несмотря на то, что свойства обработчиков решают ряд проблем, которые связаны с использованием атрибутов, в то же время это также не оптимальный подход. Еще один способ установки обработчиков событий представляет использование слушателей.

Для работы со слушателями событий в JavaScript есть объект **EventTarget**, который определяет методы **addEventListener()** (для добавления слушателя) и **removeEventListener()** для удаления слушателя. И поскольку html-элементы DOM тоже являются объектами EventTarget, то они также имеют эти методы. Фактически слушатели представляют те же функции обработчиков.

Метод addEventListener() принимает два параметра: название события без префикса on и функцию обработчика этого события. Например:

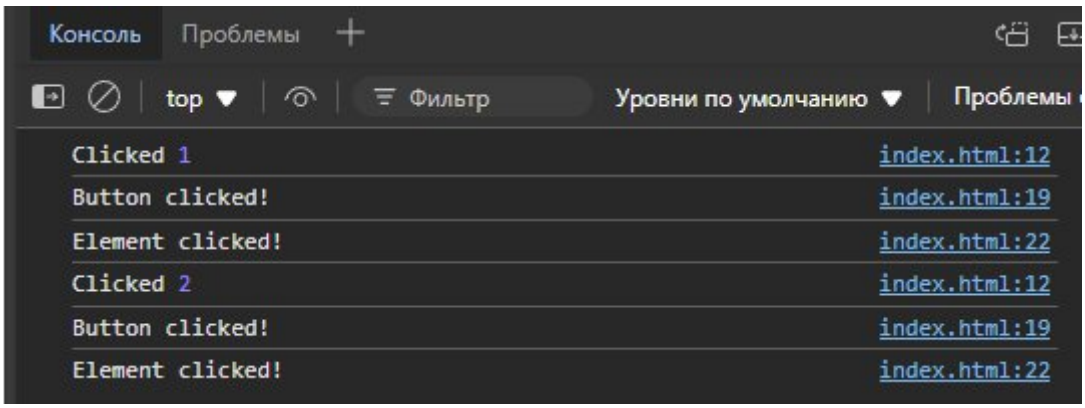
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
  <button id="btn">Click Me</div>
    <script>
    let clicks = 0;
    function btn_click(){
        console.log("Clicked", ++clicks);
    }
    const btn = document.getElementById("btn");

    btn.addEventListener("click", btn_click);
    </script>
</body>
</html>
```

То есть в данном случае опять же обрабатывается событие click. Удаление слушателя аналогично добавлению:

rect.removeEventListener("click", btn_click);

Преимуществом использования слушателей является и то, что мы можем установить для одного события несколько функций:

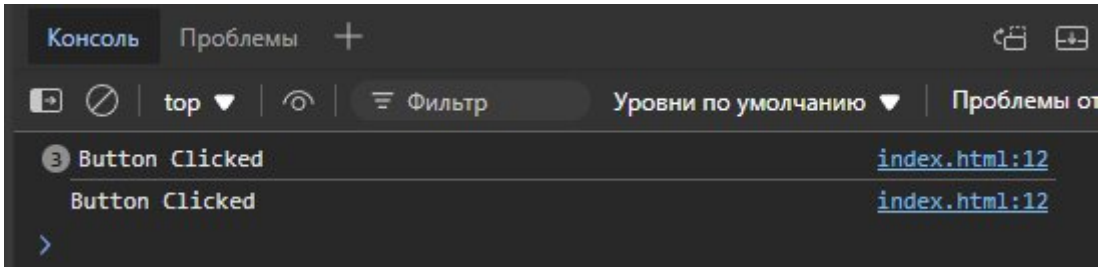
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <button id="btn">Click Me</div>
    <script>
    let clicks = 0;
    function btn_click(){
        console.log("Clicked", ++clicks);
    }
    const btn = document.getElementById("btn");

    btn.addEventListener("click", btn_click);

    btn.addEventListener("click", function(){
        console.log("Button clicked!")
    });

    btn.addEventListener("click", ()=>console.log("Element clicked!"));

    </script>
</body>
</html>
```



Передача данных в обработчик события. Объект Event

Иногда возникает необходимость передать в обработчик некоторые данные. Если обработчики событий устанавливаются с помощью атрибутов элементов, то это сделать довольно просто. Например, передадим в обработчик нажатия кнопки текст, которые будет использоваться в обработчике:

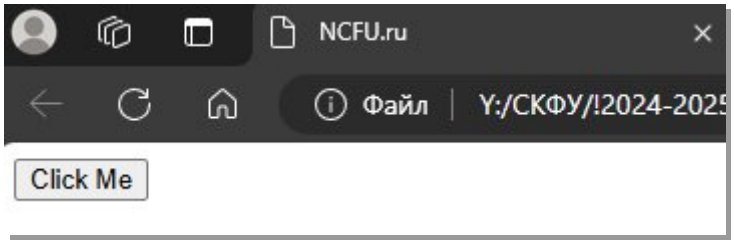
```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <button id="btn" onclick="btn_click('Button Clicked')">Click Me</button>
    <script>

    function btn_click(text){
        console.log(text);
    }
    </script>
</body>
</html>
```

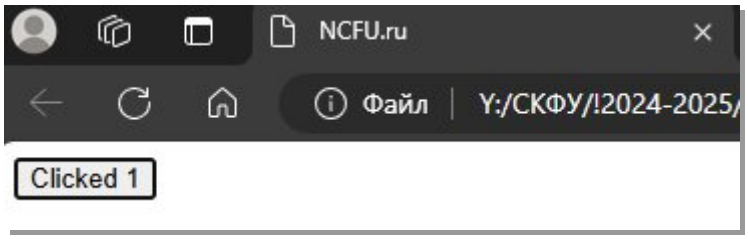


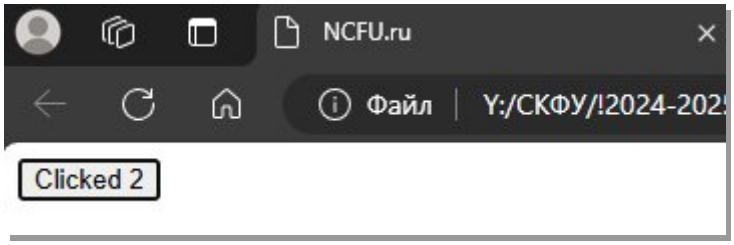
В данном случае в функцию обработчика передавалась строка, но в реальности, это может быть любой объект. Например, через значение **this** можно передать текущий объект, на котором возникает событие:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
  <button id="btn" onclick="btn_click(this)">Click Me</button>
    <script>
    let clicks = 0;

    function btn_click(btn){

        btn.textContent = `Clicked ${++clicks}`;
    }
    </script>
</body>
</html>
```

Ключевое слово this указывает на текущий объект ссылки, на которую производится нажатие. И в коде обработчика мы можем получить этот объект и обратиться к его свойствам, например, к свойству textContent и таким образом изменить текст кнопки.

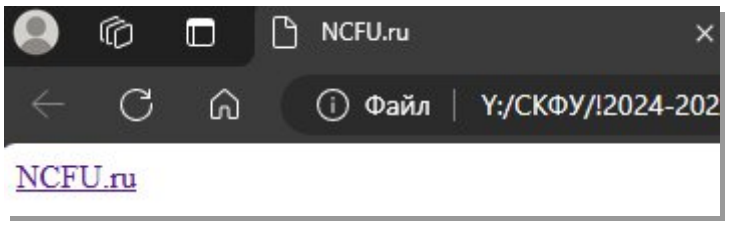






Стоит отметить, что в некоторых случаях нам может потребоваться возвращать из обработчика значение false. Например, рассмотрим следующую программу:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <a id="link" href="https://ncfu.ru" onclick="return a_click(this)">NCFU.ru</a>
    <script>

    function a_click(anchor){

        console.log(anchor.href);
        return false;
    }
    </script>
</body>
</html>
```



Здесь в атрибуте onclick ссылки - элемента <a> не просто вызывается обработчик события, а возвращается его результат:

<a id="link" href="https://ncfu.ru" onclick="return a_click(this)">

Причем функция обработчика возвращает **false**:

```
function a_click(anchor){
    console.log(anchor.href);
    return false;   // запрещаем переадресацию
}
```

Дело в том, что для некоторых обработчиков можно подтвердить или остановить обработку события. Например, нажатие на ссылку должно привести к переадресации. Но возвращая из обработчика **false**, мы можем остановить стандартный путь обработки события, и переадресации не будет. Если же возвращать значение true, то событие обрабатывается в стандартном порядке.

Если же мы вовсе уберем возвращении результата, то событие будет обрабатываться, как будто возвращается значение true:

```
<a id="link" href="https://ncfu.ru" onclick="return a_click(this)">NCFU.ru</a>
<script>

function a_click(anchor){

    console.log(anchor.href);
    return false;
}
</script>
```

Объект Event

При обработке события браузер автоматически передает в функцию обработчика в качестве параметра объект **Event**, который инкапсулирует всю информацию о событии. И с помощью его свойств мы можем получить эту информацию:

- **bubbles**: возвращает true, если событие является восходящим. Например, если событие возникло на вложенном элементе, то оно может быть обработано на родительском элементе.
- **cancelable**: возвращает true, если можно отменить стандартную обработку события
- **currentTarget**: определяет элемент, к которому прикреплен обработчик события
- **defaultPrevented**: возвращает true, если был вызван у объекта Event метод preventDefault()
- **eventPhase**: хранит число, которое представляет стадию обработки события. Возможные значения:
    - 0 (Event.NONE)
    - 1 (Event.CAPTURING_PHASE)
    - 2 (Event.AT_TARGET)
    - 3 (Event.BUBBLING_PHASE)
- **target**: указывает на элемент, на котором было вызвано событие
- **timeStamp**: хранит время возникновения события
- **type**: указывает на имя события
- **isTrusted**: указывает, событие было сгенерировано элементами веб-страницы или кодом JavaScript

Например:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>NCFU.ru</title>
</head>
<body>
    <button onclick="btn_click(event)">Click Me</button>
    <script>
    function btn_click(e){
        console.log(e);
    }
    </script>
</body>
</html>
```

При вызове функции-обработчика информация о событии доступна через объект event. Этот объект не определяется разработчиком, это просто аргумент функции обработчика, который хранит всю информацию о событии:

<button onclick="btn_click(event)">

В коде JavaScript этот объект можно получить через параметр функции:

```
function btn_click(e){
    console.log(e);
}
```

В данном случае просто выводим объект на консоль.

Остановка выполнения события

С помощью метода **preventDefault()** объекта Event мы можем остановить дальнейшее выполнение события. В ряде случаев этот метод не играет большой роли. Однако в некоторых ситуаций он может быть полезен. Например, при нажатии на ссылку мы можем с помощью дополнительной обработки определить, надо ли переходить по ссылке или надо запретить переход. Или другой пример: пользователь отправляет данные формы, но в ходе обработки в обработчике события мы определили, что поля формы заполнены неправильно, и в этом случае мы также можем запретить отправку.

Например, остановим переход по ссылке:

index.html *

```html
<a id="link" href="https://ncfu.ru">ncfu.ru</a>
<script>
function linkHandler(e){
    console.log("Link has been clicked");
    e.preventDefault();     // останавливаем переход по ссылке
}
const link = document.getElementById("link");
link.addEventListener("click", linkHandler);
</script>
```

Здесь по нажатию на ссылку будет срабатывать метод linkHandler. И, поскольку в этом методе с помощью вызова e.preventDefault() предупреждаем переход по ссылке, то перехода не будет. Данный подход, к примеру, часто используется при ajax-запросах, когда надо обработать нажатие на

ссылку, но при этом не выполнять перехода на другой ресурс, а сделать к нему запрос из кода javascript без перезагрузки страницы.

Распространение событий

Когда мы нажимаем на какой-либо элемент на станице и генерируется событие нажатия, то это событие может распространяться от элемента к элементу. Например, если мы нажимаем на блок div, то также мы нажимаем и на элемент body, в котором блок div находится. То есть происходит распространение события.

Есть несколько форм распространения событий:

- Восходящие: событие распространяется вверх по дереву DOM от дочерних узлов к родительским
- Нисходящие: событие распространяется вниз по дереву DOM от родительских узлов к дочерним, пока не достигнет того элемента, на котором это событие и возникло

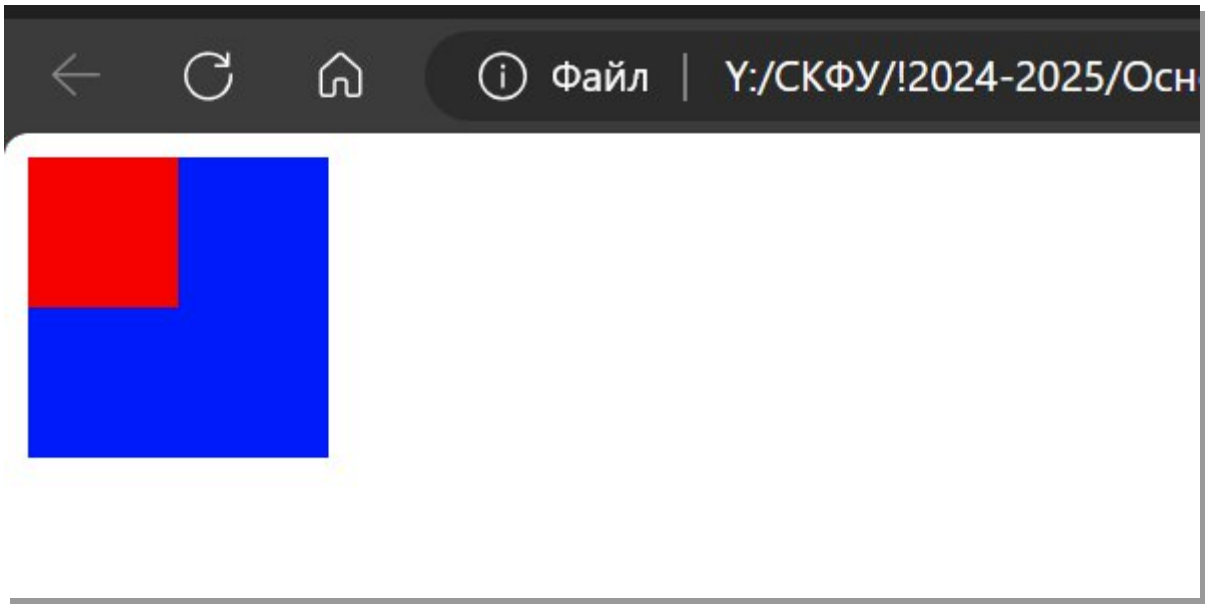
Восходящие события

Рассмотрим восходящие (bubbling) события, которые распространяются в верх по дереву DOM. Допустим, у нас есть следующая веб-страница:

index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ncfu.ru</title>
    <style>
    #blueRect{
        width:100px;
        height:100px;
        background-color:blue;
    }
    #redRect{
        width:50px;
        height:50px;
        background-color:red;
    }
    </style>
</head>
<body>
    <div id="blueRect">
        <div id="redRect"></div>
    </div>
    <script>
    const redRect = document.getElementById("redRect");
    redRect.addEventListener("click", () => console.log("Событие на redRect"));

    const blueRect = document.getElementById("blueRect");
    blueRect.addEventListener("click", ()=>console.log("Событие на blueRect"));

    document.body.addEventListener("click", () => console.log("Событие на body"));
    </script>
</body>
</html>
```

Если мы нажмем на вложенный (красный) div, то событие пойдет к родительскому элементу div и далее к элементу body:



Надо сказать, что подобное поведение не всегда является желательным. И в этом случае мы можем остановить распространение событие с помощью метода **stopPropagation()** объекта Event:

```
31      const redRect = document.getElementById("redRect");
32  redRect.addEventListener("click", function(e){
33      console.log("Событие на redRect");
34      e.stopPropagation();
35  });
```

Нисходящие события

События также могут быть нисходящими (capturing). Для их использования в метод **addEventListener()** в качестве третьего необязательного параметра передается логическое значение true или false. Значение **true** указывает, что событие нисходящим. По умолчанию все события восходящие.

Возьмем ту же веб-станицу, только изменим ее код javascript:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ncfu.ru</title>
    <style>
    #blueRect{
        width:100px;
        height:100px;
        background-color:blue;
    }
    #redRect{
        width:50px;
        height:50px;
        background-color:red;
    }
    </style>
</head>Z
<body>
    <div id="blueRect">
        <div id="redRect"></div>
    </div>
    <script>
    const redRect = document.getElementById("redRect");
        redRect.addEventListener("click", function(){
            console.log("Событие на redRect");
            }, true);

        const blueRect = document.getElementById("blueRect");
            blueRect.addEventListener("click", function(){
            console.log("Событие на blueRect");
            }, true);

    document.body.addEventListener("click", function(){
            console.log("Событие на body");
            }, true);
    </script>
</body>
</html>
```

События мыши

Одну из наиболее часто используемых событий составляют события мыши:

- click: возникает при нажатии указателем мыши на элемент
- dblclick: возникает при двойном нажатии указателем мыши на элемент
- contextmenu: возникает при открытии контекстного меню (правой кнопкой мыши)
- mousedown: возникает при нахождении указателя мыши на элементе, когда кнопка мыши находится в нажатом состоянии
- mouseup: возникает при нахождении указателя мыши на элементе во время отпускания кнопки мыши
- mousemove: возникает при прохождении указателя мыши над элементом
- mouseover: возникает при вхождении указателя мыши в границы элемента
- mouseout: возникает, когда указатель мыши выходит за пределы элемента
- mouseenter: возникает при вхождении указателя мыши в границы элемента

- mouseleave: возникает, когда указатель мыши выходит за пределы элемента

Отдельно стоит сказать про разницу между последними четырьмя событиями. mouseenter и mouseleave срабатывают только тогда, когда пересекается внешний край соответствующего элемента. А события mouseover и mouseout также срабатывают, когда другой элемент находится внутри соответствующего элемента и курсор мыши перемещается во внутренний элемент (т.е. уходит от внешнего элемента) или покидает внутренний элемент (то есть перемещается на внешний элемент).

Например, обработаем события mouseover и mouseout:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>METANIT.COM</title>
    <style>
    #blueRect{
        width:100px;
        height:100px;
        background-color:blue;
    }
    </style>
</head>
<body>
<div id="blueRect"></div>

<script>
function setColor(e){
    if(e.type==="mouseover")
        e.target.style.backgroundColor = "red";
    else if(e.type==="mouseout")
        e.target.style.backgroundColor = "blue";
}
const blueRect = document.getElementById("blueRect");
blueRect.addEventListener("mouseover", setColor);
blueRect.addEventListener("mouseout", setColor);
</script>
```

28</body>

29</html>

Теперь при наведении указателя мыши на блок blueRect он будет окрашиваться в красный цвет, а при уходе указателя мыши - блок будет обратно окрашиваться в синий цвет.

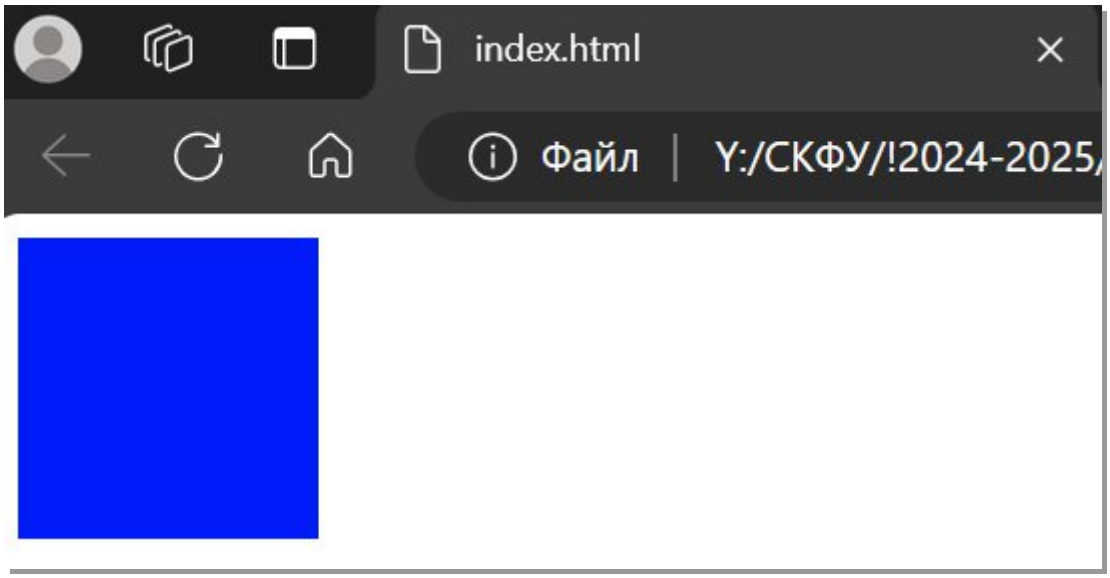
MouseEvent

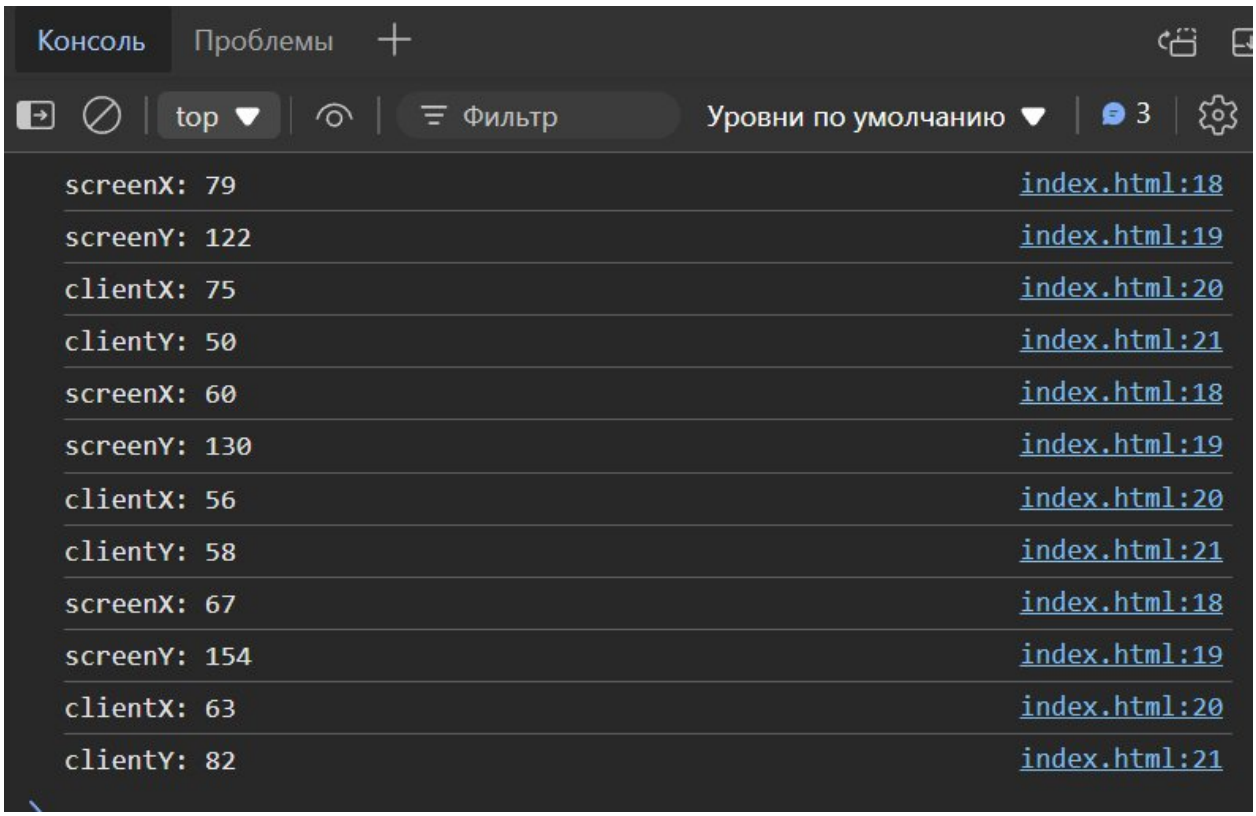
Объект Event является общим для всех событий. Однако для разных типов событий существуют также свои объекты событий, которые добавляют ряд своих свойств. Так, для работы с событиями указателя мыши определен объект **MouseEvent**, который добавляет следующие свойства:

- **altKey**: возвращает true, если была нажата клавиша Alt во время генерации события
- **button**: содержит номер нажатой кнопки мыши
- **buttons**: содержит номер, который представляет нажатую кнопку мыши. 1 обозначает левую кнопку мыши, 2 — правую кнопку мыши, 4 — колесо мыши или среднюю кнопку мыши, 8 — четвертую кнопку мыши, а 16 — пятую кнопку мыши. Если при срабатывании события было нажато несколько кнопок, это свойство содержат сумму соответствующих чисел.
- **clientX**: определяет координату X окна браузера, на которой находился указатель мыши во время генерации события
- **clientY**: определяет координату Y окна браузера, на которой находился указатель мыши во время генерации события
- **ctrlKey**: возвращает true, если была нажата клавиша Ctrl во время генерации события
- **movementX**: содержит координату X относительно предыдущей координаты X при последнем событии перемещения мыши
- **movementY**: содержит координату Y относительно предыдущей координаты Y при последнем событии перемещения мыши
- **metaKey**: возвращает true, если была нажата во время генерации события метаклавиша клавиатуры
- **region**: содержит идентификатор области или элемента, которая относится к событию
- **relatedTarget**: определяет вторичный источник возникновения события
- **screenX**: определяет координату X относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события
- **screenY**: определяет координату Y относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события
- **shiftKey**: возвращает true, если была нажата клавиша Shift во время генерации события

Определим координаты клика:

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <style>
    #blueRect{
        width:100px;
        height:100px;
        background-color:blue;
    }
    </style>
</head>
<body>
<div id="blueRect"></div>

<script>
function handleClick(e){
    console.log("screenX: " + e.screenX);
    console.log("screenY: " + e.screenY);
    console.log("clientX: " + e.clientX);
    console.log("clientY: " + e.clientY);
}
const blueRect = document.getElementById("blueRect");
blueRect.addEventListener("click", handleClick);
</script>
</body>
</html>
```





События клавиатуры

Другим распространенным типом событий являются события клавиатуры.

- keydown: возникает при нажатии клавиши клавиатуры и длится, пока нажата клавиша
- keyup: возникает при отпускании клавиши клавиатуры
- keypress: возникает при нажатии клавиши клавиатуры, но после события keydown и до события keyup. Надо учитывать, что данное событие генерируется только для тех клавиш, которые формируют вывод в виде символов, например, при печати символов. Нажатия на остальные клавиши, например, на Alt, не учитываются.

Для работы с событиями клавиатуры определен объект **KeyboardEvent**, который добавляет к свойствам объекта Event ряд специфичных для клавиатуры свойств:

- **altKey**: возвращает true, если была нажата клавиша Alt во время генерации события
- **key**: возвращает символ нажатой клавиши, например, при нажатии на клавишу "T" это свойство будет содержать "T". А если нажата клавиша "Я", то это свойство будет содержать "Я"
- **code**: возвращает строковое представление нажатой клавиши физической клавиатуры QWERTY, например, при нажатии на клавишу "T" это свойство будет содержать "KeyT", а при нажатии на клавишу ";" (точка запятой), то свойство возвратит "Semicolon".

При использовании этого свойства следует учитывать ряд момент. Прежде всего используется клавиатура QWERTY. То есть мы переключим раскладку, к примеру, на русскоязычную и нажмем на клавишу "Я", то значением будет "KeyZ" - на клавиатуре QWERTY клавиша Z представляет ту же клавишу, что и на русскоязычной раскладке "Я"

Другой момент - учитывается именно физическая клавиатура. Если нажата клавиша на виртуальной клавиатуре, то возвращаемое значение будет устанавливаться браузером исходя из того, какой клавише на физической клавиатуре соответствовало нажатие.

- **ctrlKey**: возвращает true, если была нажата клавиша Ctrl во время генерации события
- **metaKey**: возвращает true, если была нажата во время генерации события метаклавиша клавиатуры
- **shiftKey**: возвращает true, если была нажата клавиша Shift во время генерации события

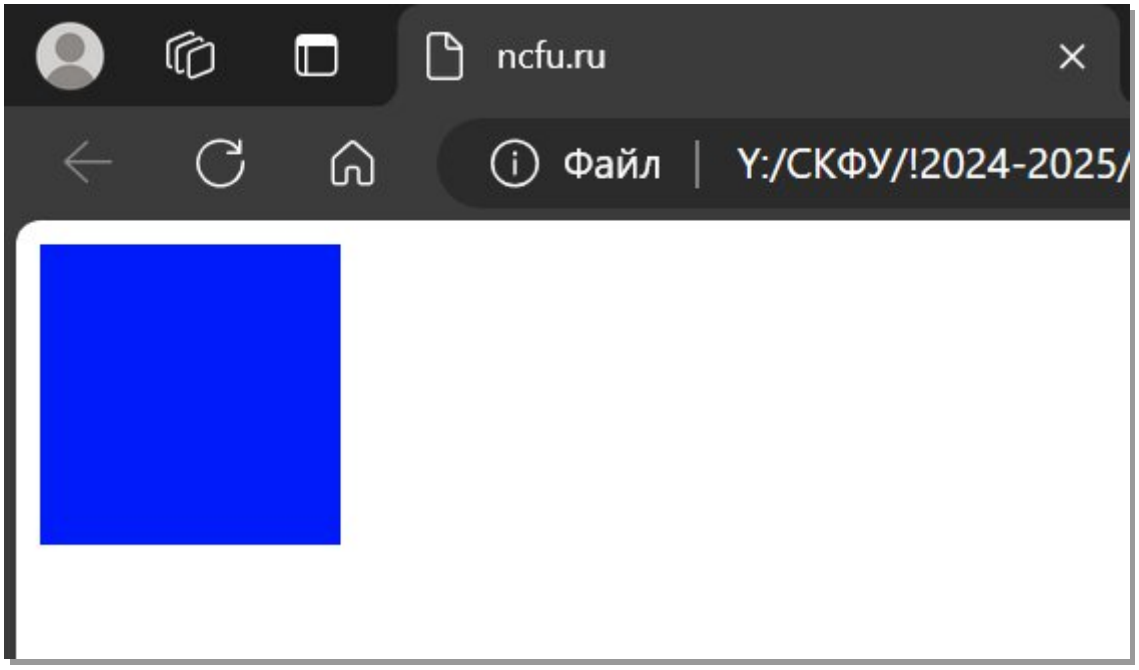
Например, мы можем с помощью клавиш клавиатуры перемещать элемент на веб-странице:

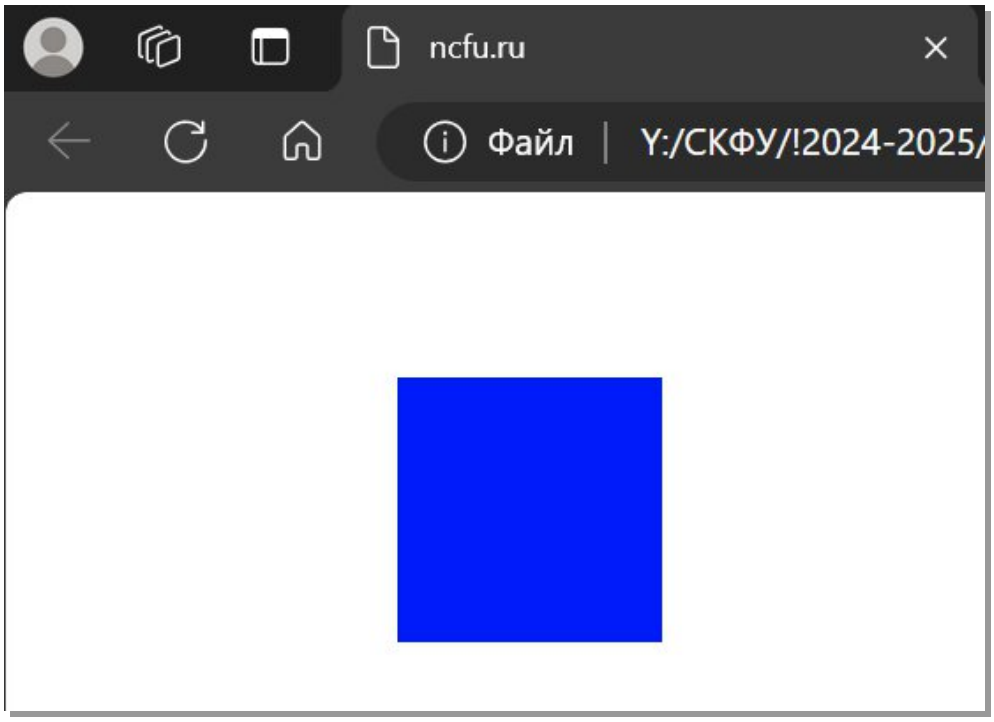
index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ncfu.ru</title>
    <style>
    #blueRect{
        width:100px;
        height:100px;
        background-color:blue;
    }
    </style>
</head>
<body>
<div id="blueRect"></div>

<script>
const blueRect = document.getElementById("blueRect");
// получаем стиль для blueRect
const blueRectStyle = window.getComputedStyle(blueRect);
// устанавливаем обработчик нажатия клавиши
window.addEventListener("keydown", moveRect);

function moveRect(e){
    const left = parseInt(blueRectStyle.marginLeft); //смещение от левого края
    const top = parseInt(blueRectStyle.marginTop);  // смещения от левой границы

    switch(e.key){

        case "ArrowLeft":  // если нажата клавиша влево
            if(left>0)
                blueRect.style.marginLeft = left - 10 + "px";
            break;
        case "ArrowUp":    // если нажата клавиша вверх
            if(top>0)
                blueRect.style.marginTop = top - 10 + "px";
            break;
        case "ArrowRight":    // если нажата клавиша вправо
            if(left < document.documentElement.clientWidth - 100)
                blueRect.style.marginLeft = left + 10 + "px";
            break;
        case "ArrowDown":    // если нажата клавиша вниз
            if(top < document.documentElement.clientHeight - 100)
                blueRect.style.marginTop = top + 10 + "px";
            break;
    }
}
</script>
</body>
</html>
```

В данном случае обрабатывается событие keydown, в обработчке которого управляем стилевыми свойствами элемента blueRect. Так как при прикреплении обработчика стиль элемента может быть не установлен, то явным образом вычисляем его с помощью метода window.getComputedStyle().

Список кодов клавиш клавиатуры можно посмотреть на странице https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key/Key_Values.





Здесь нам интересуют четыре клавиши: вверх, вниз, влево, вправо. Им соответственно будут соответствовать названия "ArrowUp", "ArrowDown", "ArrowLeft" и "ArrowRight". Если одна из них нажата, производим действия: увеличение или уменьшение отступа элемента от верхней или левой границы. Ну и чтобы элемент не выходил за границы окна, проверяем предельные значения с помощью document.documentElement.clientWidth (ширина корневого элемента) и document.documentElement.clientHeight (высота корневого элемента).

Программный вызов событий

События могут возникать не только в следствие действий пользователя на веб-странице. События также можно вызывать из кода JS.

Чтобы программно вызвать событие, у элемента на веб-странице можно вызвать метод **dispatchEvent()**, в который передается экземпляр объекта **Event** (либо его производные типа MouseEvent или KeybordEvent).

```
//  определяем объект события

const event = new Event(имя_события, config);

//   вызываем событие для элемента element

element.dispatchEvent(event);
```

Первый аргумент, передаваемый конструктору Event, представляет собой строку - тип события. Дополнительно в качестве второго параметра можно передать объект конфигурации. В частности, с помощью объекта конфигурации можно определить следующие свойства:

- cancelable: можно ли событие отменить (если true, то отменяемое событие, false - неотменяемое)
- bubbles: должно ли событие быть восходящим (если true, то восходящее)

Например, “нажмем” на ссылку:

index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ncfu.ru</title>
</head>
<body>
    <a id="link" href="https://ncfu.ru">NCFU.ru</a>
    <script>
    const link = document.getElementById("link");    // получаем ссылку
    const event = new MouseEvent("click");
    link.dispatchEvent(event);
    </script>
</body>
</html>
```