**Web-дизайн**

Лабораторная работа №11

**Строки и методы хранения данных**

Строка как объект **String**:

const name = new String("Tom");

Свойство **length** указывает на длину строки:

const hello = "привет мир";

console.log(`В строке "${hello}"  ${hello.**length**} символов`);

Метод **repeat**() позволяет создать строку путем многократного повторения другой строки:

const hello = "hello ";

console.log(hello.**repeat**(3)); // hello hello hello

Для поиска в строке некоторой подстроки используются методы **indexOf**() (индекс первого вхождения подстроки) и **lastIndexOf**() (индекс последнего вхождения подстроки).

Параметры:

* *Подстрока*, которую надо найти.
* Необязательный параметр, *с какого символа начать поиск* подстроки в строке.

Оба этих метода возвращают индекс символа, с которого в строке начинается подстрока.

const str = "привет мир. пока мир";

const key = "мир";

console.log("Первое вхождение: ", str.**indexOf**(key));    // 7

console.log("Последнее вхождение: ", str.**lastIndexOf**(key)); // 17

Метод **includes**() возвращает *true*, если строка содержит определенную подстроку.

console.log(str.**includes**("мир")); // true

console.log(str.**includes**("миг")); // false

Метод **substring**() возвращает подстроку. Принимает два параметра:

* *индекс* символа в строке, начиная с которого надо проводить обрезку строки.
* *индекс*, до которого надо обрезать строку (необязательный параметр).

Пример:

const str = "привет мир. пока мир";

const world = str.**substring**(7, 10); // с 7-го по 10-й индекс

console.log(world); // мир

Метод **substr**() также в качестве первого параметра принимает начальный *индекс* подстроки, а в качестве второго – *длину* вырезаемой подстроки:

const bye = str.**substr**(12, 4);

console.log(bye); // пока

Метод **slice**() позволяет получить из строки какую-то ее часть. Он принимает те же параметры, что и *substring*().

const world = str.**slice**(7, 10); // с 7-го по 10-й индекс

console.log(world); // мир

Этот метод похож на метод *substring*(), тем не менее между ними есть различие. В *slice*() начальный индекс должен быть меньше чем конечный. В *substring*(), если начальный индекс больше конечного, то они меняются местами.

Другое отличие, что *slice*() позволяет использовать отрицательные индексы, которые указывают на индекс символа относительно конца строки.

const bye = str.slice(-8, -4); // с 8-го до 4-ого индекса с конца

console.log(bye); // пока

Для удаления пробелов в начале и в конце стоки используется метод **trim**():

let hello = "   Привет ";

let helloTrim = hello.**trim**();

console.log("Длина строки до: ", hello.length);     // 14

console.log("Длина строки после: ", helloTrim.length);   // 6

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

let str = "Добрый день";

str = str.**replace**("день", "вечер");

console.log(str); // Добрый вечер

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

Метод **replaceAll**() позволяет заменить все вхождения подстроки.

**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"]

**Шаблоны строк**

Шаблоны строк (*template strings* / *template literals*) позволяют вставлять в строку различные значения. Подобный прием еще называют **интерполяцией**:

const name = "Tom";

const hello = **`**Hello **${name}`**; // интерполяция строки

console.log(hello);  // Hello Tom

**html-код в шаблонах**

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

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;

**Вложенные шаблоны**

Рассмотрим пример. Создадим из элементов массива список *html*:

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;

В данном случае у массива *people* вызывается функция *map*(), которое определяет некоторое действие в виде функции для каждого элемента массива. Здесь для упрощения в качестве такой функции применяется стрелочная функция. Она получает каждый элемент массива через параметр *person* и для него формирует шаблон строки:

 `<li>${person.name}</li>`

**Передача шаблона строки в функцию**

JavaScript позволяет передать в функцию шаблон строки, причем не просто как строку, но и все ее динамчески вычисляемые фрагменты в виде отдельных параметров. Подобная возможность может применяться, например, для предобработки шаблонов и их значений.

Рассмотрим следующий пример:

const person = "Tom";

function check (parts, name){

    console.log(parts);

    return parts[0] + name + parts[1]; // Person: Tom.

}

let checkedTemplate = **check`Person: ${person}.`**; // особая передача шаблона

document.body.innerHTML = checkedTemplate;

Здесь определена 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. В массиве *parts* будет два элемента: статическая часть шаблона, которая идет до вычисляемого фрагмента, и часть шаблона, которая идет после.

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

**Регулярные выражения**

В JS определить регулярное выражение можно двумя способами:

const myExp1 = /hello/;

const myExp2 = new **RegExp**("hello");

**Методы RegExp**

Чтобы определить, соответствует ли регулярное выражение строке, в объекте *RegExp* определен метод **test**(). Возвращает *true* или *false*.

const exp = /hello/;

console.log(exp.**test**("hello world!")); // true

console.log(exp.**test**("bye world!")); // false

Метод **exec()**  проверяет, соответствует ли строка регулярному выражению, а также возвращает ту часть строки, которая соответствует выражению.

let text = "hello world!";

let exp = /hello/;

console.log(exp.**exec**(text)); // hello

text = "bye world!";

console.log(exp.**exec**(text)); // null

**Группы символов**

Один из специальных элементов синтаксиса регулярных выражений представляют группы символов, заключенные в *квадратные скобки*. Например:

let text = "арбуз";

let exp = /**[абв]**/; // строка должна иметь одну из трех букв

console.log(exp.test(text)); // true

Если нам надо определить наличие в строке буквенных символов из определенного диапазона, то можно сразу задать этот диапазон:

let exp = /**[а-я]**/;

Если, наоборот, не надо, чтобы строка имела только определенные символы, то необходимо в квадратных скобках перед перечислением символов ставить **знак ^**:

let exp = /**[^0-9]**/;

console.log(exp.test("1 2345";)); // true

console.log(exp.test("12345")); // false

При необходимости мы можем собирать комбинации выражений:

const exp = /**[дт]о[нм]**/;

console.log(exp.test("дом")); // true

console.log(exp.test("тон")); // true

**Флаги выражений**

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

* Флаг **global** позволяет найти все подстроки, которые соответствуют регулярному выражению. Символ «**g**» в выражениях.
* Флаг **ignoreCase** позволяет найти подстоки, которые соответствуют регулярному выражению, вне зависимости от регистра символов в строке. Символ «**i**».
* Флаг **multiline** позволяет найти подстроки, которые соответствуют регулярному выражению, в многострочном тексте. Символ «**m**».
* Флаг **dotAll** позволяет сопоставить точку в регулярном выражении с любым символом текста, в том числе с разделителем строки. Символ «**s**».

Примеры:

let text = "привет Мир. пока Мир";

let exp = /мир/;

console.log(exp.test(text)); // false

exp = /мир/**i**;

console.log(exp.test(text)); // true – регистр игнорируется

exp = /[мпт]и[пр]/**ig**; // можно комбинировать

text = "Мир, пир, ТИП";

console.log(exp.exec(text)); // Мир

console.log(exp.exec(text)); // пир – продолжает поиск

console.log(exp.exec(text)); // ТИП – продолжает поиск

**Регулярные выражения в методах String**

**Метод split.** Пример:

const text = "Сегодня была прекрасная погода";

const exp = /\s/; // \s - пробел

const result = text.**split**(exp); // разделитель – строки, найденные через exp

result.forEach(function(value, index, array){

    console.log(value); // слова предложения по отдельности

})

**Метод match.** Пример:

const text = "Он пришел домой и сделал домашнюю работу";

const exp = /дом[а-я]\*/gi; // \* - любое количество повторений

const result = text.**match**(exp);

result.forEach(function(value, index, array){

    console.log(value); // домой, домашнюю

})

**Метод search (поиск в строке).** Пример:

const text = "hello world";

const exp = /wor/;

console.log(text.**search**(exp)); // 6

**Метод replace.** Пример:

let menu = "Завтрак: каша, чай. Обед: суп, чай.";

const exp = /чай/gi;

menu = menu.**replace**(exp, "кофе");

console.log(menu); // Завтрак: каша, кофе. Обед: суп, кофе.

**Синтаксис регулярных выражений**

**Метасимволы**

Регулярные выражения также могут использовать метасимволы:

* *«\d*» соответствует любой цифре от 0 до 9.
* *«\D*» соответствует любому символу, который не является цифрой.
* *«\w»* соответствует любой букве, цифре или знаку подчеркивания.
* *«\W»* соответствует любому символу, который не является буквой, цифрой или символом подчеркивания.
* *«\s»* соответствует пробелу.
* *«\S»* соответствует любому символу, кроме пробела.
* «.» соответствует любому символу.

Надо заметить, что метасимвол «\w» применяется только для букв латинского алфавита.

Пример:

let phoneNumbers = "11-22-33 44-55-66";

let myExp = /\d\d-\d\d-\d\d/;

document.write(phoneNumbers.**replace**(myExp, "000000")); // 000000 44-55-66

myExp = /\d\d-\d\d-\d\d/**g**;

document.write(phoneNumbers.**replaceAll**(myExp, "000000")); // 000000 000000

**Модификаторы**

Модификаторы указывают сколько раз символ или группа должны повторяться в строке:

* *{n}* соответствует *n*-ому количеству повторений символа.
* *{n,}* соответствует *n* и более количеству повторений символа.
* *{n,m}* соответствует от *n* до *m* повторений символа.
* «*?*» соответствует одному вхождению символа в подстроку или его отсутствию в подстроке.
* «*+*» соответствует одному и более повторений символа.
* «\*» соответствует любому количеству повторений или отсутствию символа.
* «^» соответствует началу строки.
* «*$*» соответствует концу строки.

Символы «?», «+», «\*» и «.» имеют особый смысл в регулярных выражениях. Чтобы их использовать в обычным значении, то символы надо экранировать с помощью слеша.

let phoneNumber = "<+7-983-111-2233>";

let myExp = /\+\d-\d{3}-\d{3}-\d{4}/;

phoneNumber = phoneNumber.replace(myExp, "80000000000");

console.log(phoneNumber); // <80000000000>

**Группы в регулярных выражениях**

Для поиска в строке более сложных соответствий применяются группы. Они заключаются в скобки. Например, у нас есть следующий код *html*, который содержит тег изображения: <img src="picture.png" />. Допустим, нам надо вытащить из этого кода:

let text = '<img src= "picture.png" />';

let exp = /[a-z]+\.**(png|jpg)**/i; // возможны варианты строки с **png** или **jpg**

let result = text.match(exp);

result.forEach(function(value, index, array){

    console.log(value); // найдет picture.png и png

})

**Получение значений групп**

Преимуществом использованием групп в регулярных выражениях является то, что мы можем получить значения каждой отдельной группы.

Например, как известно, в разных странах используются разные форматы дат. Мы хотим получаем дату в формате «*год-месяц-день*» и преобразовать ее в какой-то другой формат. Для каждого отдельного компонента мы можем определить свою группу:

const exp = /**(**\d{4}**)**-**(**\d{2}**)**-**(**\d{2}**)**/; // формат даты

const result = exp.exec("2021-09-06");

console.log(result[0]); // 2021-09-06 – главный результат поиска

console.log(result[1]); // 2021 – найдено по первой группе

console.log(result[2]); // 09 – найдено по второй группе

console.log(result[3]); // 06 – найдено по третьей группе

console.log(`${result[3]}.${result[2]}.${result[1]}`); // 06.09.2021

**Утверждения**

*Утверждения* (**assertiobs**) позвляют получить подстроку, соответствующую регулярному выражению и которая *предваряется* или *не предваряется* определенным выражением.

**Положительное утверждение** определяется с помощью выражения:

(?<=...)

После знака «=» идет выражение, которым должна предваряться подстрока.

**Отрицательное утверждение** определяется с помощью выражения:

(?<!...)

После знака «!» идет выражение, которым **не** должна предваряться подстрока.

Например:

const text = "All costs: $10.53, €15.65";

const exp = /**(?<=\$)**\d+(\.\d\*)?/; // перед числом должен идти символ $

const result = exp.exec(text);

console.log(result);   // ["10.53", ".53", …]

**Введение в JSON**

JSON (*JavaScript Object Notation*) представляет легковесный формат хранения данных. Поддерживает три типа данных: *примитивные значения*, *объекты* и *массивы*.

**Примитивные значения** представляют стандартные строки, числа, значение *null*, логические значения *true* и *false*.

**Объекты** представляют набор простейших данных, других *объектов* и *массивов*. Например, типичный объект JSON:

{

    "name": "Tom",

    "age": 30

}

В JS этому объекту соответствовал бы следующий:

var user = {

    name: "Tom",

    age: 30

}

В JSON названия свойств **обязательно** заключаются в кавычки, как строки.

Объекты могут быть сложными:

{

    "name": "Tom",

    "age": 30,

    "company": {

        "name": "Google",

        "address": "USA, Mountain View"

    }

}

**Массивы** в JSON похожи на массивы JS:

["Tom", true, 30]

К примеру, массив объектов:

[{

    "name": "Tom",

    "age": 30

},{

    "name": "Alice",

    "age": 23

}]

**Сериализация и десериализация JSON**

Для сериализации объекта JS в JSON применяется функция **JSON.stringify():**

var user = {

    name: "Tom",

    age: 30

};

var serializedUser = JSON.stringify(user);

document.write(serializedUser); // {"name":"Tom","age":30}

Для обратной операции – десериализации или парсинга json-строки в объект JS применяется метод **JSON.parse()**:

var user = {

    name: "Tom",

    age: 30

};

// сериализация

var serializedUser = JSON.stringify(user); // в JSON

// десериализация

var tomUser = JSON.parse(serializedUser); // обратно в объект

document.write(tomUser.name); // Tom

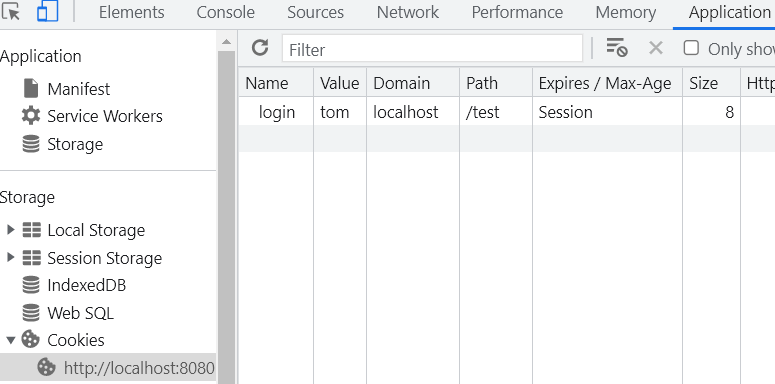
**Локальное хранение данных**

**Cookie**

Для работы с *cookie* в объекте *document* предназначено свойство **cookie**. Для установки *cookie* достаточно свойству **document.cookie** присвоить строку с *cookie*:

document.cookie = "login=tom;";

В данном случае устанавливается *cookie*, которая называется «*login*» и которая имеет значение «*tom*». И в большинстве браузеров мы можем посмотреть ее, узнать всю информацию о ней и в дальнейшем ее можно использовать в приложении:



Строка *cookie* принимает до шести различных параметров: имя , значение, срок окончания действия (*expires*), путь (*path*), домен (*domain*) и *secure*. В предыдущем примере использовались только два параметра: *имя* и *значение*.

Подобная *cookie* имеет очень ограниченный срок жизни: если явным образом не установить срок действия, то она будет удалена с закрытием браузера.

Параметр **expires** позволяет установить срок действия *cookie*:

document.cookie = "login=tom;expires=Sun, 06 Mar 2022 00:00:00 GMT;";

Формат параметра *expires* очень важен. Этот срок можно получить использовав метод **toUTCString()** объекта *Date*:

var expire = new Date();

expire.setHours(expire.getHours() + 4); // +4 часа от текущего времени

document.cookie = "login=tom;expires=" + expire.toUTCString() + ";";

Если нам надо установить *cookie* для какого-то определенного пути на сайте, то мы можем использовать параметр **path**:

document.cookie="login=tom;expires=Sun, 06 Mar 2022 00:00:00 GMT;path=/home;"

Если на нашем сайте есть несколько доменов, и мы хотим установить *cookie* непосредственно для определенного домена, тогда можно использовать параметр **domain**:

document.cookie = "login=tom; Sun, 06 Mar 2022 00:00:00 GMT;path=/;"+

"domain=blog.mysite.com;"

Параметр **secure** задает использование *SSL* (*SecureSockets Layer*) и подходит для сайтов, использующих протокол *https*. Если значение этого параметра равно *true*, то *cookie* будут использоваться только при установке защищенного соединения SSL.

**Получение cookie**

Для простейшего извлечения *cookie* из браузера достаточно обратиться к свойству **document.cookie**:

var expire = new Date();

expire.setHours(expire.getHours() + 4);

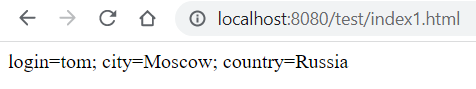
document.cookie = "city=Moscow;expires="+expire.toUTCString()+";";

document.cookie = "country=Russia;expires="+expire.toUTCString()+";";

document.cookie = "login=tom;";

document.write(document.cookie); // выводим все cookie

Здесь были установлены три *cookie*, и браузер выведет нам результат:



*Cookie* разделяются точкой с запятой, поэтому нужно еще провести некоторые преобразования, чтобы получить отдельно их имя и значение. К примеру:

var cookies = document.cookie.split(";"); // разделяем cookie

for(var i=0; i<cookies.length; i++){

    var parts = cookies[i].split("=") // делим на имя и значение

    var name = parts[0]

    var value = parts[1]

    document.write("Имя куки: " + name + "<br>");

    document.write("Значение: " + value + "<br>");

}

**Web Storage**

В *HTML5* была внедрена новая концепция для хранения данных – **web storage**.

*Web storage* состоит из двух компонентов: *session storage* и *local storage*.

**Session storage** представляет временное хранилище информации, которая удаляется после закрытия браузера.

**Local storage** представляет хранилище для данных на постоянной основе. Данные из *local storage* автоматически не удаляются и не имеют срока действия. Эти данные не передаются на сервер в запросе HTTP.

Все данные в *web storage* представляют набор пар **ключ-значение**.

Для работы с *local storage* в JS используется объект **localStorage**, а для *session storage* – объект **sessionStorage**.

Для сохранения данных надо передать в метод **setItem()** объекта *localStorage*:

localStorage.setItem("login", "tom@gmail.com");

В этот метод передаются два значения: ключ и значение сохраняемого объекта. Если в *localStorage* уже есть объект с ключом «*login*», то его значение заменяется новым.

Для получения сохраненных данных надо вызвать метод **getItem()**. В этот метод передается ключ объекта:

var login = localStorage.getItem("login"); //tom@gmail.com

Чтобы удалить объект, применяется метод **removeItem()**:

localStorage.removeItem("login");

Для полного удаления всех объектов из **localStorage**, относящихся к этой странице, можно использовать метод **clear()**:

localStorage.clear();

Надо учитывать, что данные в *localStorage* сохраняются в виде строки:

localStorage.setItem("age", 23);

var age = localStorage.getItem("age");

age = parseInt(age) + 10; // для сложения преобразуем в число

document.write(age); //33

Чтобы сохранить сложный объект можно использовать сериализацию в формат JSON:

var user ={

    name: "Tom",

    age: 23

};

localStorage.setItem("user", JSON.**stringify**(user));

var savedUser = JSON.**parse**(localStorage.getItem("user"));

document.write(savedUser.name + " " + savedUser.age); // Tom 23

Самоучитель JS - <https://learn.javascript.ru/>

Самоучитель JS - <https://metanit.com/web/javascript/>

Справочник JS - <https://javascript.ru/manual>

Справочник JS - <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference>

**Общие задания**

Выполните следующие задания с использованием JS:

1. На странице должны быть поле для ввода текста и кнопка «Добаить». Пользователь должен вводить предложения в поле. При нажатии на кнопку предложение должно добавляться на страницу как элмент списка. Для этого используйте шаблон `<li>${str}</li>`, где *str* – введенное предложение. При этом перед выводом на страницу шаблон необходимо передать в функцию как аргумент. Функция должна следующим образом преобразовать предложение:

* Удалите лишние пробелы с начала и конца.
* Все цифры замените на слова (к примеру, «1» замените на «один»).
* Если слово начинается с большой буквы, то все слово переведите в верхний регистр.

В итоге на странице будут отображаться все добавленые пользователем и преобразованные предложения.

1. На странице должно быть поле для ввода текста (используйте <textarea>). В это поле пользователь будет вводить текст для дальнейшего анализа.

Создайте кнопки выполнящие следующие действия с помощью **регулярных выражений**:

* Выводит отдельные «правильные» предложения (начинаются с большой буквы, заканчиваются знаком припенания).
* Находит все числа (могут состоять из нескольких цифр) и заменяет их на слово «NUMBER».
* Выводит диалоговое окно, куда пользователь вводит набор символов. Находит все слова, содержащие в себе только эти символы. Для создания регулярного выражения из строки можно использовать конструктор **RegExp.**
* Находит первые слова всех предложений (здесь могут помочь утверждения и символ «^», который соответствует началу строки).

1. Пользователь должен вводить текст в виде JSON-объекта (можно использовать <textarea> для ввода). Добавьте кнопку, которая запускает парсинг JSON в объект JS. Проверьте, является ли результат парсинга массивом или простым объектом. Если массив, то отобразите его в виде списка. Если объект, то отобразите его в таблице, где каждая строка состоит из двух полей: свойство и значение.
2. Создайте форму для ввода логина и пароля пользователя и кнопку «Войти». При нажатии на кнопку, если поля не пустые, должно проверяться есть ли данные в **local storage** по пользователю с таким логином. Если такой пользователь есть, то совпадает ли введенный пароль. Если пароль не совпадает, то вывести сообщение об этом и на этом закончить выполнение. Если же пароль совпадает, то отобразить данные пользователя в форме с полями для ввода (форму изначать можно сделать скрытой). Эти поля должны быть заполнены данными пользоваетля. Также должна появится кнопка «Сохранить», с помощью которой должно быть возможно сохранить внесенные в поля изменения в **local storage.** Добавьте кнопку «Удалить», чтобы иметь возможность удалить пользователя с его данными.

Данные пользователя: логин, пароль, должность, описание.

Если пользователя с введенным логином нет, то создать нового пользователя и также отобразить ему форму (ту же самую, что описана выше) для внесения и сохранения данных нового пользователя.

Данные пользователя должны быть представлены в виде объектов, так что для сохранения и загрузки из **local storage** используйте работу с JSON.

**Индивидуальное задание**

Дополните сайт, разработанный по выбранной теме по индивидуальному заданию с прошлых лабораторных работ.

В шапку (или в меню) добавьте поле для ввода текста. Это поле должно принимать регулярное выражение. Сделайте, чтобы при нажатии на Enter, осуществлялся поиск подходящей страницы на основе поиска совпадения регулярного выражения с описанием страниц сайта (которое было добавлено в прошлой лабораторной работе). Отобразите ссылку на найденную страницу.

На страницу с таблицей добавьте следующее:

Сохраняйте данные таблицы при изменении (добавлении) данных, используя **local sorage**. При загрузке страницы извлекайте данные из **local sorage** в таблицу. Так как данные должны быть представлены в виде объектов, при сохранении и загрузке используйте сериализацию и десериализацию JSON.