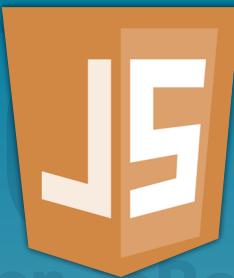


Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery



Урок № 4

Формы, проверка
достоверности форм.
Использование Cookie

Содержание

Создание элементов формы	4
Схема интернет-ресурса	4
Что такое формы.....	6
Атрибуты формы	8
Создание формы	12
Элементы формы	16
Программирование элементов формы	28
Еще раз о коллекциях элементов	28
Добавление элементов	38
Подробнее о клонировании и вставке.....	44
Удаление элементов	48
Организация проверки форм	52
Объект RegExp.	
Правила записи регулярных выражений.....	60

Методы объектов String и RegExp для работы с регулярными выражениями.....	77
Проверка достоверности данных формы	90
Что такое cookie?	107
Создание, использование и удаление Cookie.....	111
Преимущества и недостатки cookie	125
Домашнее задание.....	131
Задание 1	131
Задание 2	132
Задание 3	133

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе [Adobe Acrobat Reader](#).

Создание элементов формы

Схема интернет-ресурса

Чтобы понимать роль и место форм, особенности их атрибутов и элементов, давайте рассмотрим общую схему организации интернет-ресурса (рис. 1).



Рисунок 1

Обычно в организации ресурса выделяют три уровня: пользователь, клиент и сервер.

Уровень пользователя — это визуальный интерфейс ресурса. Именно его, как правило, и называют сайтом. Это то, что мы видим на экране браузера после успешной загрузки ресурса. Для создания интерфейса применяется язык разметки HTML.

Уровень клиента — это программное обеспечение, которое создает и прорисовывает интерфейс. Оно также принимает от пользователя введенные данные, реагирует на нажатие кнопок и ссылок, выбор списков и т.п., и передает их на сервер. Уровень клиента обеспечивается браузером и для дополнительного управления его работой используется язык JavaScript.

Уровень сервера — это «сердце» ресурса. Тут хранятся основные данные, обрабатываются запросы, проверяется авторизация, подготавливаются ответы на различные запросы клиента. На сервере располагается база данных ресурса, хранящая информацию об основных предметах ресурса (товарах, новостях, файлах), о пользователях, их истории посещений и сообщений, и многое другое. Работу сервера обеспечивают так называемые серверные технологии — PHP, ASP.NET и им подобные.

Особую роль в обмене играет канал связи как средство передачи данных от клиента серверу и обратно. Как правило, это обобщается словом «сеть» — кабели, коммутаторы, Wi-Fi-модули и т. д. Однако канал связи накладывает определенные ограничения на взаимодействие клиента и сервера. Например, канал не позволяет передавать произвольные символы, а требует их преобразования в специальный вид — транспортную кодировку. Увидеть действие транспортного кодирования несложно — введите в поиск Гугла «академия шаг» и скопируйте адресную строку браузера. Получится что-то в духе: «<https://www.google.com.ua/?q=%D0%B0%D0%BA%D0%B0%D0%B4%D0%B5%D0%BC%D0%B8%D1%8F+%D1%88%D0%B0%D0%B3>».

Это и есть транспортное кодирование, необходимое для канала связи. Обычно, кодирование совершают сам браузер и нам дополнительных действий для кодирования совершать не нужно. Тем не менее, помнить об этом надо тогда, когда мы встраиваем ссылки в наш сайт. Их желательно сразу создавать в кодированной форме, чтобы упредить возможные ошибки передачи данных.

Что такое формы

Как мы смогли увидеть в предыдущем разделе, процесс обмена данными в сетевом ресурсе довольно сложен и содержит несколько этапов. Для того чтобы разработчик сайта не вникал в каждый этап со своими особенностями, был создан специальный объект HTML — форма (**form**). Это механизм автоматизированной отправки данных, введенных пользователем, на серверную сторону сайта. Поскольку слово «форма» имеет много смыслов, в данном случае оно ближе всего по сути к форме обратной связи. Одно из самых распространенных применений формы, присутствующее практически на каждом сайте, — авторизация, форма входа на сайт (рис. 2, 3).



Рисунок 2

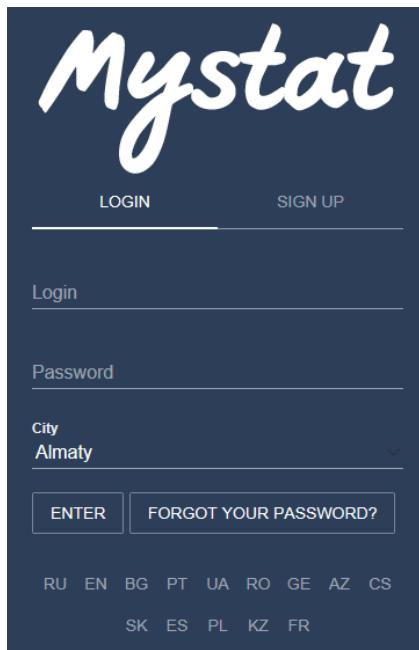


Рисунок 3

Группа элементов интерфейса может быть привязана к форме при помощи средств разметки HTML и отправлена на сервер, будучи обработанной и декодированной автоматически, без необходимости вмешательства в эти процессы со стороны разработчика.

Создается форма тегом `<form>`, закрывающий тег `</form>` обязательен. Элементы, описанные между открывающим и закрывающим тегом формы, автоматически привязываются к этой форме. По умолчанию, тег формы не вносит стилевых особенностей, то есть группировка элементов в форму не меняет способ их отображения. Но, как и любой другой объект, форма может быть дополнитель но стилизована.

Атрибуты формы

Соответственно своему предназначению, форма должна обеспечивать отправку данных на сервер. Выше мы уже отметили, что сервер — это большой программный комплекс, и отправить данные просто «на сервер» — это как просто «послать письмо» в столицу. Без указания адресата письмо до столицы дойдет, но затеряется где-то в недрах главпочтамта. Так же и с запросами к серверу необходимо указывать «адресата» — имя той из множества программ на сервере, которая примет отправленные формой данные. Имя адресата задается атрибутом формы `action: <form action="add_user.php">`.

Вторым важным атрибутом формы является метод (или способ) отправки сообщения серверу. Здесь снова нужно вспомнить, что сервер — это сложный комплекс, и он может обрабатывать запросы разного типа: запросы на чтение, запись, модификацию, удаление. Например, метод **HEAD** запрашивает у сервера только заголовочную часть сайта, без его тела, для быстрого получения метаданных наподобие названия (`title` указывается в заголовочной части). Метод **DELETE** является командой на удаление каких-то данных из ресурса. Весь перечень возможных методов запроса для действующего на сегодня протокола HTTP/1.1 приведен и детально описан в специальном стандарте RFC 2616.

Поскольку формы являются упрощенным способом автоматизированной доставки данных на сервер, для них доступны лишь два метода запроса — **GET** и **POST**. Для реализации других методов необходимо создавать собственные функции для отправки сообщений на серверы,

взамен стандартным формам. Подобные приемы будут рассмотрены далее в технологии AJAX.

С указанием метода, объявление формы будет выглядеть как `<form action="add_user.php" method="GET">`. Часто можно увидеть в учебниках, справочниках и ко-дах сайтов, что названия методов пишут то малыми, то большими буквами. С точки зрения HTML, регистр написания названия метода роли не играет (можно писать как большими, так и маленькими буквами). Однако стандарт RFC 2616 к регистру чувствителен и требует только больше буквы. Соблюдая его требования, будем указывать название метода в верхнем регистре. Будем надеяться, что при изучении более сложного материала по серверным запросам эта привычка упредит некоторые ошибки.

Различие между методами **GET** и **POST** заключается в следующем. Метод **GET** включает данные в состав URI (в адресную строку). Именно пример URI с **GET**-па-раметрами был приведен в предыдущем разделе, демон-стрирующий транспортное кодирование. К адресу сайта <https://www.google.com.ua/> прямо в адресной строке браузера дописываются данные, отправляемые на сервер. В данном случае — поисковый текст. Для разделения адреса и данных в **GET**-запросах применяется символ знака вопроса?.

В общем случае, если на сайт example.itstep.org нуж-но передать два параметра **A=1** и **B=2**, то **GET**-вариант запроса будет выглядеть как:

```
example.itstep.org?A=1&B=2
```

Символ вопроса, как уже было сказано, разделяет адрес сайта и параметры, знак **&** разделяет параметры между собой, если их несколько.

Чтобы разобраться как работает метод **POST**, необходимо рассмотреть полный состав сообщений, которыми обмениваются клиент и сервер. Не вдаваясь в глубокие подробности, все же скажем, что, кроме самого запроса, сообщение включает в себя дополнительные данные — заголовки. Рассмотрим их на следующем примере некоторого сообщения, отправляемого браузером на сервер:

```
POST /add_user.php HTTP/1.1
Host: 123.45.67.89
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64;
             rv:47.0) Gecko/20100101 Firefox/47.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
username=itstep&pass=MyItstep
```

Первая строка — это, собственно, сам запрос. Команда, которую мы передаем на сервер. Вторая указывает IP-адрес. В третьей передаются данные о типе браузера, подготовившего запрос. Далее следует указание типа содержимого запроса (заголовок **Content-Type**). Приведенное значение соответствует отправке данных из формы. Затем указывается длина сообщения (заголовок **Content-Length**) в символах, после чего следует тело сообщения — параметры, которые мы передаем из формы. Формат передачи данных такой же, как в **GET**-запросах: **имя = значение**, разделение при помощи **&**.

Следует отметить, что, во-первых, заголовков в запросе на самом деле значительно больше, здесь лишь приводится простой пример и, во-вторых, **GET**-запросы являются такими же сообщениями, только первая строка у них содержит команду **GET** и данные содержатся в ней, а не в теле сообщения.

Из-за того что передаваемые **POST**-данные включены в тело сообщения, адресная строка в браузере не меняется. Передача параметров пользователю непосредственно не видна. Однако, если страница построена с применением **POST**-данных, то есть страница появилась после отправки формы, то обновление страницы потребует повторной передачи тех же параметров. Пользователь получит предупреждение: «На странице, которую вы ищете, использовалась введенная вами информация. При возврате на эту страницу может потребоваться повторить выполненные ранее действия. Продолжить?».

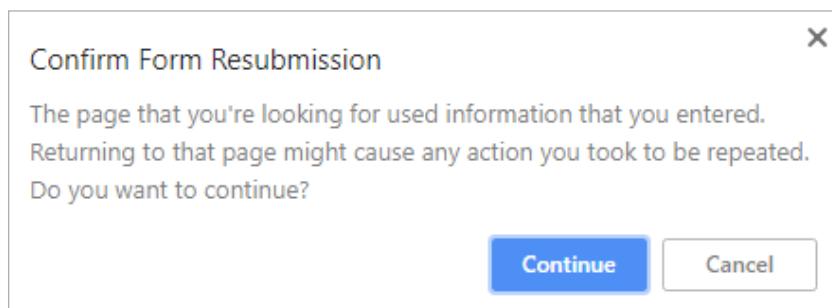


Рисунок 4

С **GET**-данными такой проблемы не возникает, т.к. данные автоматически передаются из самой адресной строки.

Краткое сравнение методов отправки формы представим в виде таблицы:

Метод	Преимущества	Недостатки	Область использования
GET	<ul style="list-style-type: none"> ■ Позволяет создать ссылку с параметрами ■ Упрощает контроль передачи 	<ul style="list-style-type: none"> ■ Данные легко просмотреть* ■ Объемные данные загромождают адрес 	Передача коротких данных, автономные ссылки
POST	<ul style="list-style-type: none"> ■ Позволяет скрыть передаваемые данные ■ Не имеет ограничений на длину данных 	<ul style="list-style-type: none"> ■ Ошибки передачи сложно заметить ■ Сообщение от браузера при обновлении страницы 	Передача изображений, файлов, больших данных

* Часто этот недостаток воспринимается как убийственный для **GET**-метода. Ради справедливости, отметим, что **POST**-данные подсмотреть не намного сложнее, открыв HTTP-сообщение полностью.

Создание формы

Для выполнения практических упражнений по работе с формами и их элементами, необходимо отметить несколько особенностей. Во-первых, форма должна отправлять данные на сервер, который передал клиенту страницу сайта, на которой расположена форма. Поскольку разработку серверной части мы еще не изучали, упражнения будем выполнять в виде отдельных html-файлов, размещенных на одном компьютере, не разделенном на *клиент* и *сервер*. Что в таком случае является сервером и куда форма будет отправлять данные?

Для ответа на эти вопросы создадим пустой файл **forms.html** в нашей рабочей папке, например, **D:\Projects\HTML**. Выбор папки произволен, вы можете создать файл

в любой папке на любом диске. Но в дальнейшем описании будем отсыльаться именно к этой папке и этому диску.

Создадим в файле простейшую форму с единственным элементом, отправляющим форму (детальнее об элементах поговорим в следующем разделе).

```
<form action='/' method='GET'>
    <input type='submit' value='Send'>
</form>
```

Как видно, форма обращается к «корню сайта», то есть установлен атрибут `action='/'`. Откроем созданный файл в браузере и увидим примерно следующее (рис. 5).



Рисунок 5

Анализируя адресную строку, можем сделать выводы:

- для работы с нашей страницей использован файловый протокол (`file://`). Для сайтов в интернете используются протоколы `http://` и `https://`;
- корнем сайта установлен диск `D:/`.

Убедимся в этом, нажав кнопку отправки формы.

В результате получим следующее см. рисунок 6.

Появление в конце адресной строки знака вопроса свидетельствует о правильном срабатывании формы. Открытие содержимого диска `D:/` подтверждает, что именно он является корневым элементом нашего «сайта».

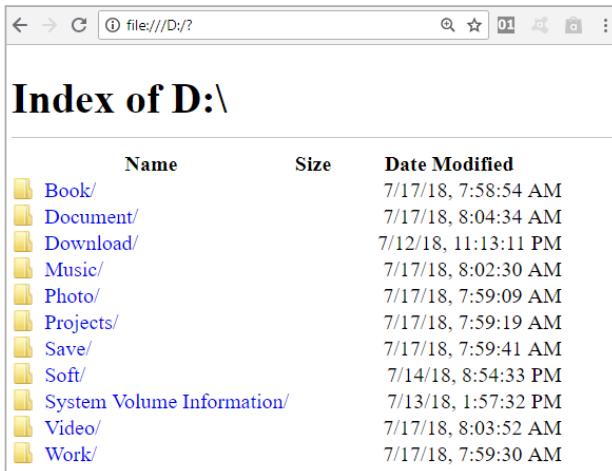


Рисунок 6

Сделаем вывод: не стоит указывать слеш в качестве адресата формы (`action='/'`). Это выражение часто можно встретить в учебниках или справочниках как простейший пример создания формы. Следует помнить, что это применяется именно для сайтов и означает «корень сайта» в котором, обычно, и располагается форма с примером.

При работе с файлами атрибут `action` следует указывать пустым (`action=""`). В таком случае данные будут передаваться по тому же адресу, по которому создана форма. Как вариант, можно вообще не указывать этот атрибут при объявлении формы. Но, в учебных целях лучше о нем не забывать, все же указывая пустые кавычки.

Замените атрибут `action` нашей формы на пустые кавычки, сохраните файл, обновите страницу в браузере и нажмите `Send`. Убедитесь, что переадресация не происходит и мы попадаем на ту же страницу. Признаком срабатывания формы будет служить появление знака вопроса в конце URI.

Для отправки формы будем всегда использовать метод **GET**, в основном благодаря возможности контроля правильности передачи данных (точнее, подготовки к передаче), так как проверить их получение со стороны сервера мы не сможем (ввиду его фактического отсутствия). На данном этапе ограничимся лишь контролем того, что данные формы правильно собраны, подготовлены и включены в состав URI.

Подытоживая, создание формы при работе с отдельным файлом, с указанием основных атрибутов будет выглядеть как:

```
<form method='GET' action=' '></form>
```

Форм на одной странице может быть несколько. У каждой может быть свой метод и свой адресат. Для работы с формами при помощи JavaScript, объект **document** имеет специальную коллекцию (массив) **document.forms**. При создании новой формы в эту коллекцию добавляется соответствующий элемент, через который формой можно управлять программным способом. Например, если на данной HTML-странице размещено две формы, то в коллекции **document.forms** будет два элемента (**document.forms[0]** и **document.forms[1]**), отвечающих каждый за «свою» форму. Порядок форм в коллекции соответствует порядку из объявления в HTML-коде.

Откройте в браузере инструменты разработчика (нажав кнопку **F12**), выберите консоль (**Console**), введите в консоли строку `«document.forms»` и нажмите **Enter**. В полученном ответе можно раскрывать детали, нажимая на треугольные символы (рис. 7).

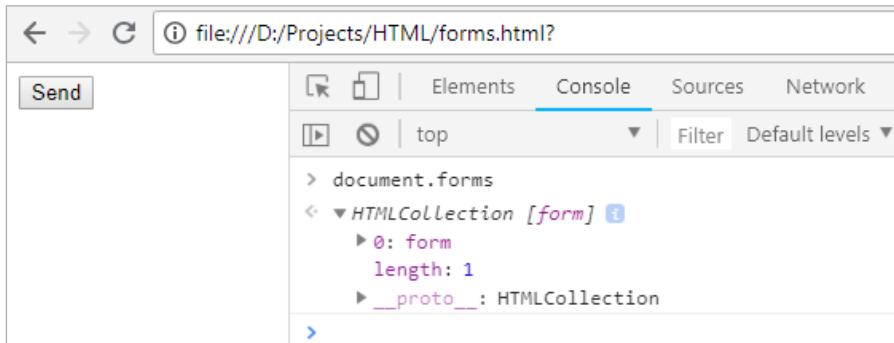


Рисунок 7

Скопируйте созданную форму (в файле [forms.html](#)) и продублируйте ее. Обратите внимание, где появляется вторая кнопка **Send**. Повторите запрос в консоли, убедитесь в расширении коллекции [document.forms](#).

Элементы формы

Элементами формы называются объекты общего интерфейса пользователя, значения из которых будут передаваться на сервер при отправке формы. Обычно, это именно те объекты, в которых можно ввести, выбрать или поменять их значение. Исключение, пожалуй, составляет специальный скрытый элемент, который добавляется к форме программным способом и пользователю для изменения недоступен.

Следует отметить, что не все то, что содержит форма, считается ее элементами. К ним не относятся «статические» объекты HTML — надписи, рисунки, блоки, разрывы строк и прочие средства разметки. Тем не менее, заключенные между тегами открытия и закрытия формы — все объекты являются дочерними для формы.

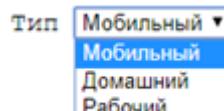
Для того чтобы отличать элементы формы (с изменямыми значениями) и все дочерние объекты, в объекте `form` предусмотрены две разные коллекции:

- коллекция `elements` содержит только элементы формы;
- коллекция `children` — все дочерние объекты.

Большинство элементов формы создаются тегом `input` и отличаются различными значениями атрибута `type`. Кроме них есть и несколько обособленных элементов.

Основные из них приведены в следующей таблице.

Тег	Описание	Примерный вид
<code><input type="button"></code>	Кнопка	
<code><input type='text'></code>	Ввод данных пользователем	
<code><input type='checkbox'></code>	«Флажок выбора»	
<code><input type='radio'></code>	Радиокнопки — один выбор из заданного набора	
<code><input type='file'></code>	Диалог открытия файла	
<code><input type='submit'></code>	Кнопка отправки формы	
<code><input type='password'></code>	Поле ввода пароля (без отображения символов)	
<code><input type='hidden'></code>	Скрытое поле	Не отображается (но передается)
<code><input type='number'></code>	Поле для ввода числа	Забронировать

Тег	Описание	Примерный вид
<code><input type='date'></code>	Выбор даты	
<code><select></code>	Выпадающий список выбора	
<code><textarea></code>	Поле для ввода большого текста	

Приведенный список в таблице не является полным справочником, а лишь отображает наиболее популярные элементы. Полный список, доступный в текущем стандарте HTML, можно узнать на сайте [WorldWideWebConsortium \(W3C\)](https://www.w3.org) по ссылке: <https://www.w3.org> или на альтернативных ресурсах справочного или учебного характера.

Элементы соотносятся с формой и попадают в ее коллекции, если они создаются между открывающим (`<form>`) и закрывающим (`</form>`) тегами формы. Если элемент создается в другом месте, от может быть соотнесен с формой при помощи специального атрибута `form`. В таком случае форме необходимо присвоить идентификатор (`id`) и для элемента, созданного вне тега `<form id="telForm">`. Для привязки к форме необходимо указать указанный атрибут `<input form="telForm" ...>`.

Для иллюстрации работы с формами создадим страницу, обеспечивающую добавление новой записи (контакта)

в телефонную книгу. Она должна содержать инструменты для ввода имени абонента и номеров телефонов, с указанием их типов.

Изначально предлагается ввод одного номера, но по нажатию кнопки **One more phone** должны добавляться поля для ввода нового номера и его типа. Также предполагается наличие возможности выбора приоритетного номера.

Примерный вид страницы, которую мы хотим получить, представлен на рисунке 8.

Рисунок 8

Приведем HTML-код страницы и прокомментируем его ниже. В созданном ранее файле **forms.html** полностью замените содержимое, скопировав или набрав этот код.

Файл также доступен в папке Sources_5.1, архив которой приложен к pdf-файлу данного урока.

```
<!DOCTYPE HTML>
<html>
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8">
    <title>Phone book</title>
```

```
<style>
    body{font-family:"Courier New",
          Courier, monospace;}
</style>
</head>

<body>
    <h2>Add new contact</h2>
    <form method='GET'>
        <b>Name</b>
        <input type="text" placeholder="Enter name"/>
        <br/><br/>
        <input type="submit" value="Save"/>
        <input type="button" value="One more phone"
              onclick="add_click()"
              style="margin-left:50px" />
        <br/><br/>
        Phone number
        <input type="text" name="phone" id="ph"
              placeholder="Enter phone number" />
        Phone type
        <select name="type">
            <option value="1" selected>Cellular</option>
            <option value="2">Home</option>
            <option value="3">Work</option>
        </select>
        Priority
        <input type="radio" name="main"
              value="1" checked />
    </form>
</body>
</html>
```

Согласно поставленным условиям, изначально в форме создается одна строка для ввода данных. Поэтому в приведенном коде создается только одна группа элементов

для номера и его типа. Добавление новых номеров будет реализовано как реакция на нажатие кнопки с вызовом функции `add_click()` в следующей главе.

Сохраните изменения в файле `forms.html` и обновите соответствующую страницу в браузере. Убедитесь, что после обновления страницы ее вид соответствует приведенному на рисунке, только с одной строкой для ввода данных.

На данном этапе можно посмотреть на различие коллекций элементов формы и ее дочерних элементов. Вызовем консоль разработчика (нажав `F12`), если она была закрыта после предыдущего упражнения. Запросим первую (нулевую в массиве) форму из коллекции документа и сохраним ее в переменной `f`, введя в консоль строку:

```
f=document.forms[0]
```

Отобразим коллекции формы. Набираем в консоли `f.elements` (и нажимаем `Enter`), затем `f.children` (и снова `Enter`). Убеждаемся, что в первой коллекции 6 элементов (`5 input` и `1 select`), тогда как во второй — 11, включая разметочные элементы `b` и `br` (рис. 9).

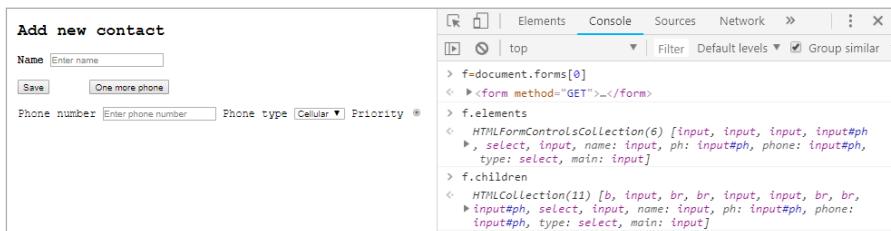


Рисунок 9

Обратите внимание, что элементы формы входят в обе коллекции, то есть доступ до них возможен при помощи

любой из них. Детальное содержание коллекций можно посмотреть, нажимая на раскрывающие «треугольники».

Нажатие на кнопку **Добавить номер** пока не приводит ни каким действиям, при этом кнопка **Save** отправляет форму (если быть точным, то это не кнопка, а элемент **submit**, автоматически отправляющий форму). При нажатии на нее в адресной строке браузера появляются данные, отделенные от адреса знаком вопроса. Заполните поля, нажмите **Сохранить** и убедитесь в переносе введенных данных в состав URI.

Если вы правильно переписали (или скопировали) код страницы и внимательно посмотрели на те данные, которые попали в адресную строку после нажатия **Save**, то, наверняка, обратили внимание, что в адресную строку не попала информация о введенном имени.

Форма передает данные только из тех элементов, которым дано программное имя (определен атрибут **name**). В нашем случае элементу ввода имени программное имя не присвоено. Поэтому, введенные данные игнорируются. Такая же ситуация с кнопками — они тоже элементы формы (как было видно из коллекций), но их значения не передаются.

Попробуйте задать атрибут **name** полю ввода имени, переписав строку в виде:

```
<b>Name</b><input type="text" placeholder="Enter name" name="name"/><br/><br/>
```

и убедитесь, что после этого данные об имени компонуются в состав URI. Также добавьте атрибут **name** кнопке и элементу **submit**, убедитесь, что в таком случае попадают

в перечень передачи и их значения, которыми являются надписи на кнопках.

Рассмотрим еще один пример, иллюстрирующий дополнительные возможности управления стилями как самой формы, так и ее элементов. Возьмем в качестве идеи форму авторизации **Logbook** (см. рис. выше) и сделаем ее значительно упрощенную версию. Создайте в вашей рабочей папке новый файл **form2.html**.

Саму форму стилизуюем как блочный элемент (**display: block**) с фоновым цветом синей гаммы (**background-color:#5599EE**). Зададим ей ширину (**width:400px**), высоту (**height:160px**) и отступ внутренних элементов от границы формы (**padding:20px**). Сохраним стилевые описания под идентификатором **#logForm**. Получим следующий HTML-документ.

```
<!doctype html>
<html>
<head>
    <style>
        #logForm{
            background-color:#5599EE;
            display:block;
            height:160px;
            padding:20px;
            width:400px;
        }
    </style>
</head>
<body>
    <form method="GET" action="" id="logForm"></form>
</body>
</html>
```

Откройте полученный документ в браузере и убедитесь, что на пустой странице отображается прямоугольник синего оттенка.

Добавим надпись **LogForm** в левую часть формы. В теле документа между открытым и закрытым тегами формы вставим блок `<div id="formText">LogForm</div>`. В стилевых определениях зададим ему белый цвет текста (`color:white`), размер шрифта (`font-size:40px`), полужирное начертание (`font-weight:bold`). Для того чтобы элементы формы могли размещаться правее надписи (обтекать), зададим ей левое плавание (`float:left`), установим ширину, близкую или равную половине ширины формы (`width:200px`) и высоту, как у формы (`height:200px`). Не забываем, что в расчет высоты также входит внутренний отступ (`padding`), добавляющийся как сверху, так и снизу. Полное стилевое описание блока примет вид:

```
#formText{
    color:white;
    float:left;
    font-size:40px;
    font-weight:bold;
    height:200px;
    width:200px;
}
```

Дальше размещаем сами элементы формы: селектор выбора города, поля ввода логина и пароля, а также кнопка отправки формы. Все они имеют схожий стиль скругленного блока, только кнопка отправки отличается цветом и выравниванием текста. Создадим стилевой класс, отвечающий за скругленную форму элементов.

Большинство стилевых определений уже повторялись и не комментируются. Основу округления составляют свойства радиуса границы (**border-radius:15px**) и отключения прорисовки самой границы (**border-width: 0**).

```
.rounded{  
    border-radius:15px;  
    border-width:0;  
    display:block;  
    height:30px;  
    margin:10px 0;  
    padding:0;  
    width:170px;  
}
```

Для выделения кнопки отправки зеленым цветом, с выравниванием по центру, создадим стилевое определение следующего содержания.

```
#enter{  
    background-color:green;  
    color:white;  
    text-align: center;  
}
```

Теперь создадим сами элементы формы. В теле формы создаем селектор выбора городов. Для примера ограничимся тремя. Следует отметить, что селектор не поддерживает подсказку (**placeholder**), поэтому применяется технология создания дополнительного поля выбора (опции) с атрибутами скрытости (**hidden**) и недоступности для выбора (**disabled**). В то же время, эта опция устанавливается как выбранная изначально (**selected**), а ее значение задается

пустым (`value=""`), что позволит в будущем проводить его контроль программными методами. Опция-подсказка размещена в селекторе на последнем месте. Итоговый код для селектора примет вид:

```
<select class="rounded" name="city">
    <option value="Kiev">Kiev</option>
    <option value="Nikolaev">Nikolaev </option>
    <option value="Odessa">Odessa</option>
    <option value="" disabled selected hidden>
        City</option>
</select>
```

Создание остальных элементов не содержит хитростей. Для всех указываем класс **rounded**, отвечающий за окружную рамку. Для кнопки отправки формы — дополнительный идентификатор (**enter**), задающий ее особенности. HTML-код элементов будет выглядеть следующим образом:

```
<input type="text" name="login" class="rounded"
       placeholder="login"/>
<input type="password" name="password"
       class="rounded" placeholder="password"/>
<input type="submit" id="enter" class="rounded"
       value="ENTER"/>
```

Полный код файла `form2.html` доступен в папке `Sources_5.1`, архив которой приложен к pdf-файлу данного урока.

После завершения оформления кода сохраните результат и откройте файл **forms2.html** в браузере. Если он уже открыт — обновите страницу, нажав клавишу **F5** или кнопку на интерфейсе браузера.

В итоге наша форма в браузере будет иметь следующий вид:

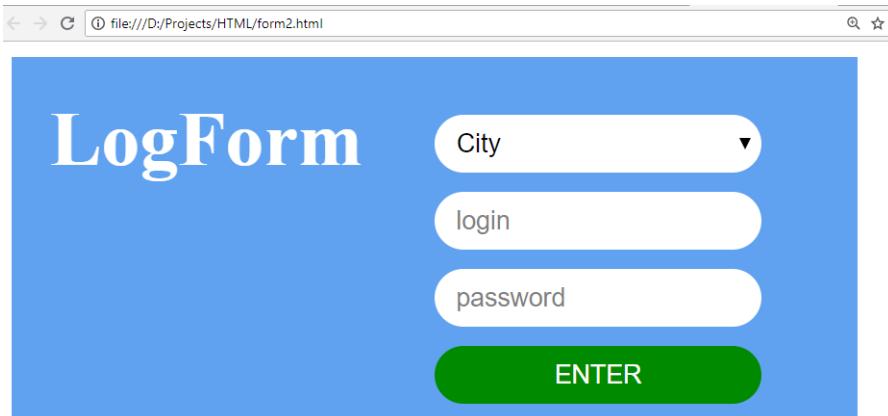


Рисунок 10

Заполните поля формы, нажмите кнопку **ENTER**. Убедитесь, что введенные данные попадают адресную строку браузера.

Включите консоль разработчика и проведите инспекцию коллекций `document.forms`, `document.forms[0].elements`, `document.forms[0].children`, как было описано выше.

Программирование элементов формы

Еще раз о коллекциях элементов

Вернемся к форме телефонной книги, созданной нами ранее, и рассмотрим более детально ее структуру с различных точек зрения. Напомним код этой формы и ее внешний вид:

```
<form method='GET'>
    <b>Name</b>
    <input type="text" placeholder= "Enter name"
           name="name" /><br/><br/>
    <input type="submit" value="Save"/>
    <input type="button" value="One more phone"
           onclick="add_click()"
           style="margin-left:50px"/>
    <br/><br/>
    Phone number
    <input type="text" name="phone" id="ph"
           placeholder="Enter phone number" />
    Phone type
    <select name="type">
        <option value="1" selected>Cellular</option>
        <option value="2">Home</option>
        <option value="3">Work</option>
    </select>
    Priority
    <input type="radio" name="main" value="1"
           checked />
</form>
```

The screenshot shows a web form with the following elements:

- Name:** A text input field labeled "Enter name".
- Save:** A button.
- One more phone:** A button.
- Phone number:** A text input field labeled "Enter phone number".
- Phone type:** A dropdown menu set to "Cellular".
- Priority:** A radio button group with one radio button selected.

Рисунок 11

Как мы уже знаем, существуют две коллекции для хранения информации о структуре формы — `elements` и `children`. Первая хранит только элементы самой формы, вторая — все дочерние элементы. Однако существует еще одна коллекция — `childNodes`, хранящая все дочерние узлы. Чтобы разобраться, в чем заключаются отличия между этими коллекциями, проведем их детальный анализ.

Для анализа коллекций создадим средства для их отображения на странице браузера. Конечно, можно проводить анализ при помощи средств консоли разработчика, как мы делали ранее, но для многократного использования удобнее создать специальные кнопки. После обновления страницы в консоли придется повторять действия по доступу к форме и ее коллекциям, тогда как нажать на кнопку будет гораздо проще.

Создадим после объявления формы дополнительную кнопку, которая будет обеспечивать анализ коллекции `elements`, а также добавим абзац (параграф) для вывода данных на страницу. Добавлять элементы надо именно после формы (после закрытого тега `</form>`), иначе интересующие нас коллекции формы будут меняться после добавления новых кнопок в саму форму. Фрагмент кода создания кнопки и абзаца приведен ниже, *полный код*

доступен в папке *Sources_5.2*, архив которой приложен к pdf-файлу данного урока (файл *form5_2-1.html*).

```
<button onclick="showElements()">Elements</button>
<p id="out"></p>
```

В разделе описания скриптов добавим функцию `showElements()`, которая будет отвечать на нажатие кнопки.

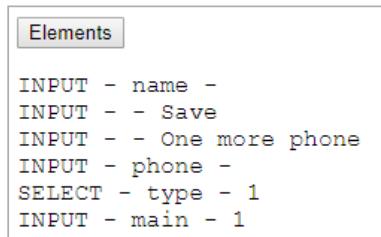
```
function showElements() {
    var e = document.forms[0].elements,
    p=document.getElementById("out");
    p.innerHTML = "";
    for ( var i=0; i<e.length; i++) {
        p.innerHTML += e[i].tagName + " — " +
        e[i].name + " — " + e[i].value + "<br>";
    }
}
```

Функция запрашивает коллекцию элементов первой формы документа (`document.forms[0].elements`), получает доступ к абзацу, предназначенному для вывода информации (`p = document.getElementById("out")`), после чего проходит циклом по коллекции и выводит на страницу данные об имени тега (`tagName`), программном имени (`name`) и значении (`value`) каждого элемента. Для разделения выводимых данных используется знак — с пробелами с двух сторон.

После нажатия на кнопку на странице, появится следующее содержимое (рис. 12).

Эти 6 элементов мы уже наблюдали в консоли в предыдущей главе. Здесь видно, что, например, первый элемент

input имеет имя **name** но не имеет значения (пустое пространство после второго «`-»`). Это пустая строка ввода для имени абонента. Можете ввести в нее произвольные данные, снова нажать на кнопку **Elements** и убедиться в правильности наших выводов.



```
Elements
INPUT - name -
INPUT - - Save
INPUT - - One more phone
INPUT - phone -
SELECT - type - 1
INPUT - main - 1
```

Рисунок 12

Второй элемент, наоборот, не имеет имени (это видно по пустому пространству между знаками «`-»`), но имеет значение **Save**. Это кнопка (точнее, элемент **submit**) отправки формы. Мы специально не давали имя, чтобы исключить этот элемент из перечня передаваемых на сервер данных.

В то же время, последние элементы имеют и имя, и значение. Это достигнуто применением атрибутов **selected** для списочного элемента (**select**) и **checked** — для радиокнопки (последний **input**).

Аналогичным образом проведем анализ коллекции **children**. Добавим соответствующую кнопку сразу после кнопки **Elements**, абзац для вывода будем использовать тот же, новый создавать не будем.

```
<button onclick="showChildren()">Children</button>
```

В секции **<script>** опишем функцию, отвечающую за нажатие кнопки.

```

function showChildren() {
    var c = document.forms[0].children,
        p = document.getElementById("out");
    p.innerHTML = "";
    for (var i=0; i<c.length; i++) {
        p.innerHTML += (i+1) + ". " +
        c[i]. tagName + "<br>"
    }
}

```

Содержание функции практически полностью повторяет предыдущую, но, во-первых, в качестве анализируемой коллекции используется **children**. Во-вторых, выводим только порядковый номер элемента и имя его тега, поскольку значения (**values**) актуальны только для элементов формы и имена (**name**) также даны не всем элементам.

Сохраним измененный файл и обновим страницу в браузере. После нажатия на кнопку **Children** появится следующий текст:

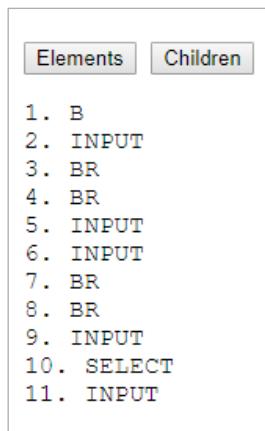


Рисунок 13

Это тоже знакомые нам 11 элементов, соответствующие структуре формы. Первый объект **B** — это выделенная жирным шрифтом надпись **Name** перед полем ввода имени. Объекты **BR** — это разрывы строк, примененные нами для оформления.

Аналогичным образом, после кнопки **Children** добавим третью кнопку, анализирующую наиболее полную коллекцию **childNodes**:

```
<button onclick="showNodes () ">Nodes</button>
```

Ее функция нажатия будет иметь вид:

```
function showNodes () {
    var n = document.forms[0].childNodes,
        p = document.getElementById("out");

    for ( var i=0; i<n.length; i++) {
        p.innerHTML += (i+1) + ". " + n[i].
                        nodeName + "<br>"
    }
}
```

В данной коллекции не все элементы имеют имя тега, поэтому для вывода использовано поле **nodeName** — имя узла, наиболее общее представление объекта DOM.

Напомним, что полный код со всеми функциями доступен в папке Sources_5.2, архив которой приложен к pdf-файлу данного урока (файл form5_2-1.html).

Сохраняем изменения, обновляем страницу и нажимаем на кнопку **Nodes**. В результате получим следующее (рис. 14).

```

1. #text
2. B
3. #text
4. INPUT
5. BR
6. BR
7. #text
8. INPUT
9. #text
10. INPUT
11. #text
12. BR
13. BR
14. #text
15. INPUT
16. #text
17. SELECT
18. #text
19. INPUT
20. #text

```

Рисунок 14

К имеющимся 11-ти элементам коллекции `children` добавлены 9 элементов с именем узла `#text`. Это текстовые элементы, связанные с реальным оформлением страницы, в том числе и не совсем преднамеренным.

Так, первый узел `#text` соответствует разделительному элементу между открывающим тегом формы и началом надписи `Name`. Обратите внимание, что в исходном коде эта надпись (`Name`) формируется с новой строки после тега `<form>` и содержит отступ в начале строки для красоты оформления кода:

```

<form method='GET'> ← разрыв строки
<b>Name</b>
↑ отступ

```

Вызовем консоль браузера (`F12`) и запрошим детализацию этого элемента. Введем `f=document.forms[0]` для получения самой формы (нажимаем `Enter`). Далее вводим

`t=f.childNodes[0]` для детализации первого (нулевого) элемента коллекции `childNodes` формы. После нажатия `Enter` увидим `#text`. Нажмем на раскрытие детальных данных (треугольник слева от надписи) и получим список полей данного объекта (см. рис. 15).

Его содержимое дублируется в нескольких полях (`data`, `nodeValue`, `textContent`, `wholeText`) и представляет собой перевод строки (символ `\n`), после которого следуют несколько пробелов, соответствующих отступу в HTML-коде формы.

Add new contact

Name

Phone number Phone type Priority

Elements Children Nodes

```

<form method="GET"></form>
> t=f.childNodes[0]
<#text>
  assignedSlot: null
  baseURI: "file:///D:/Projects/HTML/"
  childNodes: NodeList []
  data: "\n"
  firstChild: null
  isConnected: true
  lastChild: null
  length: 9
  nextElementSibling: b
  nextSibling: b
  nodeName: "#text"
  nodeType: 3
  nodeValue: "\n"
  ownerDocument: document
  parentElement: form
  
```

Рисунок 15

Как вы, наверняка, помните, любую последовательность пробельных и разрывных символов браузер воспринимает как один пробел, который отображается в режиме сбора строчных элементов и игнорируется при сборке блочных или плавающих элементов.

Можем в этом убедиться, исследовав второй элемент `#text` в коллекции `childNodes`. Как видно, он располагается на 3-м месте в коллекции между элементами `B` и `INPUT`. Присмотревшись к HTML-коду формы, обратим внимание на пробел после закрывающего тега `` и открывающего

<input... в строке Name <input type="text" (для наглядности разрыв увеличен).

Уберите этот пробел в HTML-коде, чтобы теги следовали друг за другом неразрывно (<input...). Сохраните файл, обновите страницу в браузере. Убедитесь, что между надписью **Name** и полем ввода пробел исчез:



Рисунок 16

А при нажатии кнопки **Nodes** отображаются 19 элементов без исследованного нами элемента **#text** на 3-й позиции.

Можете повторить описанные действия для исключения первого элемента **#text**, удалив пробелы и разрыв строки между тегами формы и надписи **Name**. Убедитесь, что элемент из коллекции исчезает, однако внешний вид страницы не изменяется, поскольку браузер еще не перешел в режим сборки строчных элементов и пробелы игнорируются.

Элементы **#text**, кроме разметки HTML-кода, соответствуют также надписям, не оформленным в собственные теги. Например, 14-й узел **#text** (в первоначальной коллекции из 20-ти элементов) соотносится с надписью **Phone number**. Это можно проверить, повторив описанные выше действия в консоли браузера (`f=document.forms[0], t=f.childNodes[13]`).

Кроме собственно надписи, этот элемент содержит также разрыв строки и отступ перед и после текста. То есть элемент собирает в себе все, что находится между закрытым тегом (в данном случае `
`) и следующим открытым (**input** — для ввода телефона). Повторно обратите

внимание, что для надписи `Name`, заключенной в тег ``, создается не узел `#text`, а объект `B`.

Подведем некоторые итоги по исследованию структуры формы и ее коллекций.

Все средства оформления HTML-кода в виде разрыва строк, отступов (пробелы или табуляции) между тегами, в конце концов, попадают в DOM-структуру страницы. За счет них увеличиваются коллекции дочерних элементов и, естественно, количество занятой памяти и трудоемкость обновления страницы или ее части. В нашем простом примере таких элементов у формы почти половина (9 из 20-ти), что не позволяет ими пренебрегать, оправдываясь, что их не так уж и много. Конечно, отказываться от красивого оформления HTML-кода не стоит, по крайней мере, на этапе разработки. Но при выводе готового продукта можно задуматься и об оптимизации.

Да и стиль программирования может содержать оптимизированные конструкции. Например, если переместить пробел в надписи `Name` внутрь тега ``, то есть записать `Name <input...>` вместо `Name <input...>` (разрывы все так же увеличены для наглядности), то лишнего элемента `#text` создано не будет, а отступ между надписью и полем ввода сохранится.

Пробелы, внутри тегов (` Name `), а также пробелы, разделяющие их атрибуты (`<form method='GET'>`), не переносятся в структуру DOM в виде отдельных объектов. Это может быть использовано для задач оформления кода и упреждения создания новых элементов.

Надписи, оформленные без собственных тегов, попадают в специальные объекты `#text`. С одной стороны, это

может служить рекомендацией все же обрамлять надписи тегами для более предсказуемой структуры документа. С другой стороны, расширение формы путем создания дополнительных надписей в процессе выполнения программных скриптов (как говорится, «на лету») потребует от нас использование специальных узлов `#text`, не соответствующих ни одному стандартному тегу.

Добавление элементов

Разобравшись в особенностях структуры формы и ее элементов, поставим себе задачу реализовать функциональность добавления нового поля для ввода дополнительного номера телефона по нажатию кнопки **Добавить номер**. Изначально для этого мы заложили обработчик события нажатия кнопки **One more phone** в виде функции `add_click()` (`onclick="add_click()"`).

Для того чтобы добавить новый номер, необходимо создать несколько дополнительных элементов и включить их в состав формы. Проанализируем детально состав одной строки ввода номера телефона:

1. Сначала идет разрыв (`
`), обеспечивающий переход на новую строку.
2. Затем надпись: **Phone number**.
3. Далее, элемент **INPUT** для ввода номера телефона.
4. Надпись: **Phone type**.
5. Селектор выбора типа номера.
6. Надпись: **Priority**.
7. Радиокнопка для отметки основного номера.

Также следует напомнить, что данные формы отправляются только из тех элементов, которым заданы имена

(атрибут `name`). Причем для разных полей имена должны быть разными, чтобы была возможность разделить данные на стороне сервера. Исключение составляют только радиокнопки выбора основного номера, для которых имя должно быть одинаковым, чтобы обеспечить их зависимость между собой.

Чтобы формировать различные имена для новых элементов ввода, введем глобальную переменную-счетчик `phoneCounter`, значение которой будем приписывать к именам полей.

В момент первого запуска функции эта переменная должна быть инициализирована. Используем проверку `if(typeof phoneCounter == 'undefined')` для определения первого запуска функции, при котором данная переменная должна быть установлена в `1`. После этого счетчик можно увеличивать обычным способом (`phoneCounter++`). Соответственно, начало функции будет выглядеть следующим образом (*полный код со всеми функциями доступен в папке Sources_5.2, архив которой приложен к pdf-файлу данного урока (файл form5_2-2.html)*).

```
<script>
    function add_click() {
        if (typeof phoneCounter == 'undefined')
            phoneCounter = 1;
        phoneCounter++;
        var f = document.forms[0];
```

В переменной `f` сохранен первый элемент коллекции форм документа, то есть наша основная форма, к которой мы будем добавлять элементы.

Для создания новых элементов в JavaScript предусмотрено несколько функций. Функция `document.createElement` создает элемент по имени тега, например, разрыв строки можно создать вызовом `document.createElement('br')`, а элемент ввода — `document.createElement('input')`. Другая функция `document.createTextNode` служит для создания узлов типа `#text`, для которых имя тега не задано. В качестве аргумента функция принимает текст, который будет отображаться. Например, `document.createTextNode('Phone number')`.

Следует отметить, что описанные функции только создают объекты заданного типа, но не включают их в структуру документа. Для этих целей необходимо воспользоваться методом `appendChild` того объекта, к которому элементы добавляются. В нашем случае — формы, сохраненной в переменной `f`.

Таким образом, добавление разрыва строки в форму будет выглядеть следующим образом:

```
var b = document.createElement('br');
f.appendChild(b);
```

А добавление надписи `Phone number`:

```
var t = document.createTextNode('Phone number ');
f.appendChild(t);
```

Добавление поля ввода нового номера потребует дополнительных действий. Новому созданному элементу типа `input` необходимо явно указать тип `'text'` (в предыдущей главе было показано, что в зависимости от типа

элемент может вести себя совершенно по-разному). Также нужно сформировать имя с учетом переменной-счетчика и, по аналогии с уже существующим полем, добавить подсказку (**placeholder**) **Enter phone number**.

В итоге, создание поля и его добавление к форме будет иметь следующий код:

```
var phoneInput = document.createElement('input');
phoneInput.type = 'text';
phoneInput.name = 'phone' + phoneCounter;
phoneInput.placeholder = 'Enter phone number';
f.appendChild(phoneInput);
```

Добавление надписей **Phone type** и **Priority**, а также радиокнопки (разновидность **input**) полностью аналогичны описанным выше примерам.

Остановимся подробнее на добавлении списка выбора типа телефонного номера. Напомним его структуру:

```
<select name="type">
  <option value="1" selected>Cellular</option>
  <option value="2">Home</option>
  <option value="3">Work</option>
</select>
```

Для того чтобы повторить такой же элемент, необходимо создать элемент типа **select**, три элемента типа **option**, заполнить их данными и добавить к родительскому **select**. Однако, в данном случае можно воспользоваться еще одной функцией, позволяющей создавать копии (клоны) элементов — **cloneNode**. Поскольку новый элемент будет отличаться только именем, этот путь будет

более простой — создать клон и поменять имя, вместо пошагового воссоздания нового элемента с абсолютно одинаковой внутренней структурой.

Для того чтобы создать клон элемента, нужно получить к нему доступ. Сделать это можно несколькими способами, воспользуемся самой короткой коллекцией формы `elements`. Так как список имеет имя (`name='type'`), доступ к нему можно получить, обратившись к коллекции по этому имени `f.elements['type']`. Это значение можно сохранить в программную переменную (`selector`), после чего вызвать метод клонирования для создания новой переменной `var newSelector = selector.cloneNode(true)`. Аргумент `true` в вызове функции указывает, что нужно скопировать все содержимое элемента полностью. Остается поменять имя новому объекту и добавить его к форме.

Полный код функции приведен ниже.

Напомним, что полный код со всеми функциями доступен в папке Sources_5.2, архив которой приложен к pdf-файлу данного урока (файл form5_2-2.html).

```
function add_click(){
    if (typeof phoneCounter == 'undefined')
        phoneCounter = 1;
    phoneCounter++;
    var f = document.forms[0];

    var b = document.createElement('br');
    f.appendChild(b);

    var t = document.createTextNode('Phone number');
    f.appendChild(t);
```

```
var phoneInput = document.createElement('input');
phoneInput.type = 'text';
phoneInput.name = 'phone' + phoneCounter;
phoneInput.placeholder = 'Enter phone number';
f.appendChild(phoneInput);

var t2 = document.createTextNode(' Phone type ');
f.appendChild(t2);

var selector = f.elements['type'];
var newSelector = selector.cloneNode(true);
console.log(newSelector);
newSelector.name = 'type' + phoneCounter;
f.appendChild(newSelector);

var t3 = document.createTextNode(' Priority ');
f.appendChild(t3);
var mainRadio = document.createElement('input');

mainRadio.type = 'radio';
mainRadio.name = 'main';
mainRadio.value = phoneCounter;
f.appendChild(mainRadio);
}
```

Сохраните полученный код и обновите страницу браузера. Теперь при нажатии на кнопку **One more phone** появляются дополнительные «строки» ввода. Эти строки появляются перед кнопками исследования коллекций, т. к. они вынесены за форму, а строки добавляются к самой форме. Убедитесь, что при нажатии на эти кнопки новые элементы (**phone2, type2, phone3, type3,...**) отображаются в составе коллекций. Проверьте, что радиокнопки работают корректно и списки выбора не зависят друг от друга.

Add new contact

Name	Enter name				
<input type="button" value="Save"/>	<input type="button" value="One more phone"/>				
Phone number	Enter phone number	Phone type	Cellular	Priority	<input type="radio"/>
Phone number	Enter phone number	Phone type	Home	Priority	<input checked="" type="radio"/>
Phone number	Enter phone number	Phone type	Work	Priority	<input type="radio"/>
<input type="button" value="Elements"/>	<input type="button" value="Children"/>	<input type="button" value="Nodes"/>			
<pre> INPUT - name - INPUT -- Save INPUT -- One more phone INPUT - phone - SELECT - type - 1 INPUT - main - 1 INPUT - phone2 - SELECT - type2 - 1 INPUT - main - 2 INPUT - phone3 - SELECT - type3 - 1 INPUT - main - 3 </pre>					

Рисунок 17

Введите тестовые данные и нажмите [Save](#). Проверьте правильность и полноту переноса введенных данных в адресную строку браузера.

Подробнее о клонировании и вставке

Как мы увидели, создание комплексных элементов со сложной внутренней структурой можно упростить, применяя способ клонирования. Следует помнить, что клон создает полную копию объекта, в том числе имена и значения всех составных блоков. Если в новом элементе необходимо поменять все эти имена и значения, то механизм клонирования получится ничем не проще или короче, чем воссоздание объекта шаг за шагом по

одному элементу. Если же изменений в новом комплексе немного, то клонирование действительно упростит код.

В нашем случае, новая «строка» для ввода номера полностью повторяет предыдущую, за исключением новых имен полей для номера телефона и его типа. Можно ожидать, что при помощи клонирования добавление элементов упростится.

Чтобы воспользоваться клонированием, элементы, которые добавляются, надо поместить в общий контейнер (`<div>`). Поменяем определение формы:

```
<form method='GET'>
    <b>Name </b>
    <input type="text" placeholder="Enter name"
           name="name" />
    <div style="margin:10px 0;">
        Phone number
        <input type="text" name="phone" id="ph"
               placeholder="Enter phone number" />
        Phone type
        <select name="type">
            <option value="1" selected>Cellular
            </option>
            <option value="2">Home</option>
            <option value="3">Work</option>
        </select>
        Priority
        <input type="radio" name="main" value="1"
               checked />
    </div>
    <input type="button" value="One more phone"
           onclick="add_click()" />
    <input type="submit" value="Save" style=
           "margin-left:50px" />
</form>
```

Кнопки `One more phone` и `Save` перенесены вниз, «строка» с номером, типом и радиокнопкой помещена в блок `<div>`. Так как блочный элемент сам по себе создается с новой строки, разрывы `
` в новой форме не используются, а отступ между строками задается стилем блока-оболочки (`style="margin:10px 0;"`).

В новой форме еще больше проявляется разница между коллекциями `elements` и `children`. Запросите состав этих коллекций в консоли. В коллекции `elements` видны все те же 6 элементов — без группировок в отдельные блоки. Тогда как в коллекции `children` всего 5 элементов, третьим из которых идет блок `div`, в котором сгруппированы несколько элементов формы. Об этом следует помнить при работе с формами: коллекция `elements` не хранит информацию о группировке элементов в блоки, `children` — хранит.

Поменяем также функцию `add_click()`. Полный код со всеми функциями доступен в панке `Sources_5.2`, архив которой приложен к pdf-файлу данного урока (файл `form5_2-3.html`).

```
function add_click(){
    if (typeof phoneCounter == 'undefined')
        phoneCounter = 1;
    phoneCounter++;
    var f = document.forms[0];
    var line = f.children[2];
    var newLine = line.cloneNode(true);
    newLine.children[0].name = 'phone' + phoneCounter;
    newLine.children[0].value = '';
    newLine.children[1].name = 'type' + phoneCounter;
    newLine.children[2].checked = false;
    f.insertBefore(newLine,f.children[phoneCounter+1]);
}
```

Первые строки совпадают с предыдущей редакцией функции и были прокомментированы ранее.

Получаем доступ к блоку `<div>`, отвечающему за «строку» ввода номера. Как уже было проверено в консоли браузера, этот блок находится на третьей позиции коллекции `children` формы. Соответственно, получить к нему доступ можно при помощи строки `var line = f.children[2]`.

Затем создается клон этого блока (`var.newLine = line.cloneNode(true)`). После клонирования заменяем имена полей ввода с учетом счетчика, а также сбрасываем значения поля ввода имени (`newLine.children[0].value = ''`) и радиокнопки (`newLine.children[2].checked = false`), чтобы при клонировании создавались пустые поля, и радиокнопка была неотмеченной.

Последней строкой кода новый элемент добавляется к форме. Только вместо рассмотренного ранее метода `appendChild` использован метод `insertBefore`. Метод `appendChild` добавляет элемент в конец коллекции, тогда как `insertBefore` может вставить в любое место. Поскольку мы перенесли кнопки добавления и отправки вниз формы, применение `appendChild` будет добавлять строки после кнопок, что не совсем правильно с точки зрения дизайна.

Для того чтобы использовать метод `insertBefore`, нужно указать два аргумента. Первым аргументом этого метода выступает новый элемент, который добавляется к форме. Вторым — элемент, перед которым нужно вставить новый.

Определить «место», в которое необходимо вставлять новую строку помогает введенный для имен полей счетчик `phoneCounter`. Его значение позволяет нам определить количество «строк» и вставить новую стро-

ку в конце остальных, но перед кнопками добавления и отправки формы.

Сохраните изменения, обновите страницу браузера, убедитесь в работоспособности кнопок.

The screenshot shows a web-based contact form titled "Add new contact". It contains three identical rows for inputting contact details. Each row has a "Name" field with placeholder text "Enter name", a "Phone number" field with placeholder text "Enter phone number", a "Phone type" dropdown menu set to "Cellular", and a "Priority" section with a radio button that is currently selected. Below these rows is a blue rectangular button labeled "One more phone" and a white rectangular button labeled "Save".

Рисунок 18

Отказ от разрывов строк (`
`) и применение отступов (`margin`) позволило плавно регулировать расстояние между элементами. Визуально, элементы кажутся не так плотно сверстаны, как в предыдущих примерах.

Итак, механизм клонирования действительно позволил значительно упростить создание группы новых элементов. Метод `insertBefore`, в свою очередь, предоставляет более гибкий способ добавления элементов формы.

Удаление элементов

В завершение, рассмотрим процесс удаления элементов программным способом.

Для исключения элемента `elem` из дочерних узлов объекта `obj` применяется метод `obj.removeChild(elem)`. То есть для удаления элемента необходимо сначала получить программный доступ к родительскому объекту, после — к дочернему элементу, который нужно удалить и,

затем, вызвать метод `removeChild` родительского объекта с передачей в качестве параметра дочернего.

Элементы можно удалять как по одному, так и сразу блоки элементов. В последнем случае, элементы должны быть сгруппированы каким-либо образом (например, в блок `<div>`) и в метод `removeChild` должна быть передана ссылка на группу элементов.

Расширим функциональность нашей формы добавлением возможности удаления записей, которые были добавлены в процессе работы (то есть все, кроме первой). Группировка элементов формы в блоки, сделанная для задач клонирования, упростит нам и процесс удаления.

Во-первых, нужно модифицировать функцию `add_click()` в которой происходит создание новых «строк» для ввода телефонов. Добавим в состав новых строк кнопку удаления данной строки. Перед последней строкой функции `add_click()` добавим следующий код (*полный код функции доступен в папке Sources_5.2, архив которой приложен к pdf-файлу данного урока, файл form5_2-4.html*):

```
var removeButton = document.createElement('input');
removeButton.type = 'button';
removeButton.value = 'Remove';
removeButton.addEventListener('click', rem_click);
newLine.appendChild(removeButton);
```

Большинство операций уже были описаны выше: создается элемент типа `input`, устанавливается его тип `button` и значение (надпись на кнопке) `Remove`. Далее устанавливаем обработчик события нажатия кнопки (`click`), связывая его с функцией `rem_click` при помощи

метода `addEventListener`. В завершение добавляем кнопку в клонированный блок `newLine`.

Во-вторых, необходимо создать функцию `rem_click`, которая будет отвечать за нажатие кнопки и, собственно, удаление строки. Так как для всех кнопок используется одна и та же функция, различать их будем при помощи источника сообщения, то есть прототип функции-обработчика должен предусматривать аргумент: `rem_click(event)`.

При появлении события нажатия кнопки будет вызван его обработчик и передан объект события `event`. В этом событии источник полученного сообщения хранится в поле `event.target`. Именно эта конструкция даст нам возможность определить кнопку, которая была нажата.

Однако нас интересует не столько сама кнопка, сколько блок `<div>`, в котором она находится. Обратиться к этому блоку можно несколькими способами: при помощи метода `parentNode` или метода `closest`. Метод `parentNode` возвращает родительский элемент, который для кнопки и будет блоком `<div>`. Метод `closest` позволяет найти ближайший родительский элемент заданного типа. Второй метод предпочтительнее, если кнопка будет вложена в какие-либо дополнительные элементы. Соответственно, определить блок-«строку» в котором была нажата кнопка можно при помощи следующего кода:

```
var line = event.target.closest('div')
```

Родительским объектом для блока является форма. Доступ к форме мы уже неоднократно получали через коллекцию форм документа. Итоговый код функции `rem_click` примет вид:

```
function rem_click(event) {
    var line = event.target.closest('div');
    var f = document.forms[0];
    f.removeChild(line);
    phoneCounter--;
}
```

В конце функции надо не забыть уменьшить счетчик наших блоков, т. к. это значение используется для поиска последней строки.

После составления всех функций проверяем работоспособность всех функций формы экспериментальным путем.

За счет реализации удаления блоков в программе появилась потенциальная ошибка. Допустим, пользователь дважды нажал кнопку добавления номера. При этом сформируются поля с именами **phone2** и **phone3** в новых строках. После этого пользователь удаляет вторую (среднюю) строку (с **phone2**), а последнюю (с **phone3**) остается. Переменная-счетчик **phoneCounter** в функции удаления (**rem_click**) будет уменьшена (с 3 на 2). Если снова нажать кнопку добавления, счетчик увеличится (с 2 на 3) и снова сформирует имя **phone3**. В результате на форме будут две строки с одинаковыми именами полей (**phone3**). Отказаться от уменьшения счетчика нельзя, т. к. его значение используется для определения последней строки на форме.

Решить эту проблему можно путем разделение счетчиков для формирования имен и для оставшегося количества строк. Попробуйте сделать это самостоятельно.

Для контроля готовое решение доступно папке *Sources_5.2*, архив которой приложен к pdf-файлу данного урока, файл *form5_2-5.html*.

Организация проверки форм

Ранее мы научились создавать формы и манипулировать их элементами управления — создавать их, клонировать, удалять или преобразовывать. Далее мы рассмотрим следующий шаг при работе с формами — проверку их достоверности.

Напомним, что HTML-форма представляет собой механизм автоматизированной отправки введенных пользователем данных на сервер сайта. Обратим внимание на несколько ключевых моментов в этом механизме работе форм.

Во-первых, данные в форму вводит пользователь и ответственность за то, что в поле «Имя» будет вписано именно имя, а не номер телефона, полностью лежит на самом пользователе. Во-вторых, пользователь может ничего не вписать в некоторые поля, оставив их пустыми. Это, как мы убедились на прошлом уроке, не повлияет на работоспособность формы, и на сервер будут отправлены «пустые» данные. В-третьих, формы срабатывают автоматически, то есть отправка будет осуществлена после нажатия пользователем на соответствующую кнопку (точнее, элемент «`submit`»).

Эти особенности, заложенные в работу форм изначально, не подходят нам, если мы хотим обеспечить качественную обработку данных. Для этого надо как-то проконтролировать, что нужные поля не оставлены пустыми, постараться проверить, что в номере телефона содержатся только цифры, а адрес электронной почты

соответствует стандартному шаблону. При этом нужно будет предотвратить автоматическую отправку формы, если контрольная проверка приведет к недопустимым результатам. Другими словами, для улучшения работы с HTML-формами нам придется вмешиваться в их внутренние механизмы, заменяя их на собственные алгоритмы.

Проверка правильности введенных данных, их соответствие шаблонам или ограничениям обобщается термином «проверка достоверности форм». Это специальный сокращенный термин, используемый веб-разработчиками (так как на самом деле формы не проверяются, а проверяются введенные в их элементы данные). Проверка достоверности форм обычно проводится в несколько шагов.

Первый, простейший шаг заключается в указании правильных атрибутов для элементов формы. С вводом в действие стандарта HTML 5 перечень таких атрибутов был существенно расширен как раз с целью их более гибкого применения в формах. Например, для числовых данных следует указывать атрибут «`<input type="number">`», а не оставлять обобщенный «`<input type="text">`», в который можно ввести не только цифры. Основные атрибуты были рассмотрены на предыдущем уроке, полный их перечень можно узнать на справочных ресурсах (например [здесь](#)).

Для элементов форм, заполнение которых требуется в обязательном порядке, надо добавлять атрибут `«required»`. Например, для поля ввода имени это будет выглядеть как

```
<input type = "text" placeholder = "Your first name"  
name="name" required />.
```

Атрибут «`required`» может быть указан для элементов различного типа, не только текстовых. Элементы, у которых задан этот атрибут, проверяются на пустоту самим браузером. Если значение для элемента не введено (не вписан текст, не выбрана радиокнопка, не отмечена «галочка» и т.п.) будет выдано предупреждение «Заполните это поле» и отправка формы не начнется. Однако автоматическая проверка браузером проводится только на пустоту. Любые введенные данные позволяют пройти данный этап проверки. Пользователю достаточно нажать пробел или любую символьную клавишу, чтобы браузер посчитал, что поле заполнено (не пустое).

Второй этап реализуется путем контроля содержимого элементов формы непосредственно перед ее отправкой. Организация этого этапа может быть реализована двумя способами.

Первый способ — перехват события «`onsubmit`» формы. Сообщение об этом событии генерируется, когда пользователь нажимает кнопку отправки формы или клавишу «`Enter`» при заполнении элемента формы.

Перехват события обеспечивается следующим образом: в скриптовой части HTML кода создается функция, задачей которой будет проверка достоверности формы. Для примера, назовем ее «`checkForm`». В функцию передается объект, содержащий детали перехваченного события. Обычно его называют «`event`» (или просто «`e`» в сокращенной форме скрипта) и полное имя функции будет выглядеть как «`checkForm(event)`». Созданная функция обязательно должна возвращать значение: в случае успешной проверки «`true`», иначе — «`false`». Следующий

код иллюстрирует объявление функции, в которой происходит проверка длины текста, введенного в элемент с идентификатором «name». Если длина меньше одного символа, функция возвращает «false», иначе — «true».

```
<script>
    function checkForm(event) {
        var nameText = document.
            getElementById("name").value;
        if(nameText.length < 1) return false;
        return true;
    }
</script>
```

Для привязки созданной функции-обработчика к форме, в HTML части, в открывающем теге создания формы должен быть указан атрибут «onsubmit»:

```
<form onsubmit = "return checkForm(event)"
      method="GET" id="poll">
```

Особенностью данного способа перехвата, в отличие от многих других событий, является необходимость добавления ключевого слова «return» перед именем обработчика. Это будет транслировать (пересыпалть) результат, возвращенный функцией, дальше в механизм автоматической отправки формы. Если «return» не вписать, то даже при негативном возврате проверочной функции «checkForm» отправка формы все равно будет произведена.

В некоторых случаях, для разделения работы HTML-дизайнеров и JavaScript-программистов, обработчики событий подключаются не в HTML тегах, а программным

способом, при помощи метода «`addEventListener`» нужного элемента (обратите внимание, в определении, приведенном выше, тег формы устанавливает ее идентификатор `id="poll"`).

```
document.getElementById("poll").  
    addEventListener("submit", checkForm)
```

В таком случае исчезает возможность передать результат проверки данных в стандартный обработчик формы. Для упреждения автоматической отсылки необходимо воспользоваться методом «`preventDefault`» аргумента «`event`», передающегося в функцию «`checkForm`». То есть функция должна начинаться инструкцией «`event.preventDefault()`»:

```
function checkForm(event) {  
    event.preventDefault()  
    var nameText = document.getElementById("name") .  
                    value;  
    if(nameText.length > 0)  
        document.getElementById("poll").submit(); //  
                                         // sending form  
}
```

Поскольку стандартная обработка события отменена, для отправки формы необходимо явно вызвать ее метод «`submit()`». Возвращать значение из функции при этом уже необязательно.

Второй способ отменить автоматическую отправку формы состоит в отказе от элемента типа «`submit`» и использовании обычной кнопки. В таком случае событие «`onsubmit`» от нее не генерируется, а вызывается непо-

средственный обработчик события нажатия кнопки. Внутри этого обработчика, в случае успешной проверки, форма отправляется принудительным вызовом ее метода «`submit()`», как и в предыдущем примере.

```
<input type = "button" onclick = "checkButtonClick()"
       value = "Send" />
<script>
    function checkButtonClick() {
        var nameText = document.getElementById("name") .
                       value;
        if(nameText.length > 0)
            document.getElementById("poll").submit(); // sending form
    }
</script>
```

Как показывает практика, в некоторых случаях, если на форме нет элемента «`submit`», то его функцию берет на себя обычная кнопка. В таком случае форма будет отсылается в любом случае после нажатия кнопки, независимо от результата ее работы, т.к. перехвачено событие «`click`», а не «`submit`». Для упреждения автоматической отсылки формы можно пойти двумя путями: 1) указать

```
<form onsubmit = "return false" >
    <input type = "text" id = "name" />
    <input type = "button"
           onclick = "checkButtonClick()" value = "Send" />
</form>
```

в определении формы, либо 2) вынести определение кнопки за форму, то есть написать ее HTML код после закрытого тега формы `</form>`.

```
<form>
  <input type = "text" id = "name" />
</form>
<input type = "button" onclick = "checkButtonClick()"
       value = "Send" />
```

В первом случае автоматическая отправка формы будет прекращена возвратом всегда ложного значения `«false»`. Во втором случае кнопка не будет принадлежать форме и ее нажатие не будет иметь никакого отношения к событиям формы.

Программный вызов метода `«submit»` в проверочной функции сам-по-себе не генерирует сообщение `«onsubmit»`. Это означает, что форма отправляется без дополнительного вызова перехватчика этого события `«checkForm»` (если он есть). Вся проверка достоверности формы должна быть проведена в одной функции, а не разбита на две — одна для кнопки, другая для перехваченного события `«onsubmit»`.

Также, при замене элемента `«submit»` на кнопку не будет срабатывать отправка формы по нажатию `«Enter»`. Хорошо это или плохо — решать разработчику, но помнить об этом необходимо.

Третий этап проверки достоверности выполняется уже на стороне сервера, на который форма отсылает данные. Поскольку обработчики событий — это JavaScript функции, их можно переопределить или удалить из панели инструментов разработчика или вообще отключить поддержку JavaScript в настройках браузера. Также можно «подделать» отправку формы вручную сформировав запрос с нужными параметрами в адресной строке. На

стороне сервера не может быть уверенности в полной правильности данных, обеспеченной первыми этапами проверки. Третий этап проверки достоверности будет рассмотрен позже при изучении back-end разработки.

Объект RegExp. Правила записи регулярных выражений

Перехватив сообщение автоматической отправки формы и разобравшись, как внедрить собственный обработчик для этого события, мы переходим к следующему шагу — собственно, к проверке введенных (или не введенных) данных.

Простые способы проверки мы уже затронули в предыдущем разделе, анализируя длину введенного текста. Таких проверок, очевидно, будет недостаточно для реальных задач, тем не менее, приведенные методы являются наиболее эффективными для того чтобы проверить элемент на пустоту.

Более сложной задачей проверки достоверности данных является контроль их соответствия некоторому шаблону. Например, адрес электронной почты должен содержать символ `@`, а номер телефона — цифры. Здесь обобщить методики практически невозможно — каждые отдельные данные имеют свои шаблоны и проверку одного элемента обычно нельзя применять для проверки другого. Можно лишь выделить некоторые общие черты. Во-первых, все данные форм собираются (и позже передаются) исключительно в текстовом (строковом) виде. Помним, что HTTP — это текстовый протокол. Во-вторых, при их проверке удобно пользоваться обобщающими терминами на подобие «цифра», «пробельный символ», «буква» или

наоборот «не-цифра», «не-буква». В таком случае описание шаблона значительно упрощается, так в имени не может быть «не-буквы», а в возрасте «не-цифры».

Следуя отмеченным особенностям, был разработан один из мощнейших механизмов проверки строк на соответствие некоторому шаблону — регулярные выражения ([Regular Expressions, RegExp](#)).

Регулярные выражения представляют собой язык составления шаблонов, при помощи которых строки можно обрабатывать — проверять, разделять, заменять и т.п. В большинстве языков программирования существует поддержка регулярных выражений, некоторые даже используют их для составления программ (например, язык Perl). В данном разделе мы рассмотрим основные возможности регулярных выражений направленные на задачи проверки данных, вводимых пользователем на веб-формах.

В терминологии регулярных выражения часто используются слова «шаблон» и «маска». Слово «маска» заимствовано у системного программирования, в котором популярен термин «битовая маска». Он означает проверку значений отдельных бит числа, например, являются ли два последних бита числа единичными. Термин «шаблон» пришел из проектирования и означает некоторый образец, которому должен соответствовать объект. В области регулярных выражений эти два термина означают одно и то же. Аналогия с масками удобна, когда выражение требует: «строка должна заканчиваться на цифру», а с шаблонами, если «строка не должна содержать пробелов».

Еще одним термином регулярных выражений является слово «флаг». Корни этого слова также уходят в историю

вычислительной техники — к флаговому регистру процессора. С обычными флагами (зnamёнами) этот термин прямой связи не имеет. Смысл термина «флаг» означает некоторые дополнительные условия, при которых строка сравнивается с шаблоном. Например, считать ли одинаковыми большие и маленькие буквы одного алфавита («а» и «А»).

Шаблон или маску можно представить себе как детскую игру, в которой разные фигуры надо вставлять в разные отверстия. Кубик вставляется в квадратное отверстие, цилиндр — в круглое, клин (призма) — в треугольное. Флаги при этом указывают можно ли фигурки покрутить. Ведь при некоторых условиях цилиндр может подойти к квадратному отверстию, если вставлять его не круглой основой, а поперек — по высоте. Подобная ассоциация вполне применима к регулярным шаблонам, через которые испытуемая строка проходит или нет, в том числе, если ее немного «покрутить».

Шаблон регулярного выражения в JavaScript можно создать двумя способами — литералом и конструктором. Литерал представляет собой выражение вида «/шаблон/флаги». Литерал не берется в кавычки, признаком его начала служит прямой слеш (`«/»`). Литеральное выражение может быть использовано самостоятельно либо сохранено в переменной:

```
var template1 = /\D/
```

В приведенном примере шаблоном является выражение `«\D»`, флаги не указаны.

При помощи конструктора регулярное выражение может быть создано инструкцией

```
new RegExp ("шаблон", "флаги")
```

В таком случае и шаблон, и флаги берутся в кавычки, слеши-ограничители шаблона не нужны. Точно так же созданный шаблон может быть сохранен в переменной:

```
var template2 = new RegExp ("\D", "g") .
```

В независимости от формы создания, регулярные выражения используются одинаково. Небольшое различие между ними все же есть. Литеральные шаблоны являются константными, созданными на этапе написания программы. В то же время шаблоны, созданные при помощи конструктора можно поменять во время выполнения программы — «на лету», то есть сформировать шаблоны по результатам каких-либо проверок или вычислений. Приведенные различия касаются только возможностей создания шаблонов, в контексте их применения различий нет.

Как уже упоминалось выше, основными способами использования регулярных выражений в JavaScript являются:

- проверка строки на соответствие шаблону;
- разделение строки на части по шаблону;
- преобразование строки по шаблону (или замена по шаблону).

Начнем с рассмотрения первого способа. Проверка строки на соответствие шаблону обеспечивается методом **«test»** регулярного выражения. В качестве аргумента в метод передается сама строка, которую нужно проверить.

Результатом работы метода «`test`» является логическое значение (`true` или `false`), информирующее о положительном или отрицательном результате проверки. Соответственно, чаще всего вызов метода «`test`» сопровождается условным оператором «`if`». Например, инструкция для проверки строки «`str`» на соответствие шаблону «`template1`» может выглядеть как

```
var template1 = /\D/  
  
if( template1.test(str) ) {  
    alert("test passed");  
}  
else  
{  
    alert("test failed");  
}
```

Давайте составим простейший шаблон и посмотрим, как он работает. Для выполнения простых упражнений в несколько строк удобно воспользоваться консолью разработчика в браузере, вместо создания отдельных html-файлов. Откройте консоль (детали см. в уроке 1) и введите определения шаблона, сохранив его в переменной «`template`», в конце нажмите «`Enter`»:

```
template = /1/
```

Сам шаблон представляет собой цифру «`1`», по правилам записи шаблонов заключенную между двумя прямыми слешами. Проверка при помощи этого шаблона будет означать поиск цифры «`1`» в исследуемой строке. Если строка содержит единицу, то проверка приведет

к положительному результату. Проверим при помощи созданного шаблона несколько строк. Вводим в консоль последовательно следующие команды, нажимая «Enter» после ввода каждой строки

```
template.test("123")
template.test("234")
template.test("I am 21 years old")
```

Наблюдаем за ответами, получаемыми в консоли. В первом и третьем случае строка содержит цифру «1», т.е. тест на шаблон приводит к истинному результату (**true**). Обратите внимание, что положение цифры не играет роли. В первой строке она на первом месте, в третьей — в середине. Вторая строка не содержит единицы и, как следствие, результат проверки шаблоном ложный (**false**) (рис. 19).

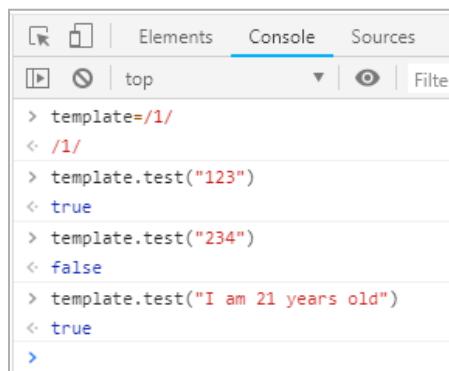


Рисунок 19

Заменим шаблон на чуть более длинный. Введем в консоль (и нажмем «Enter»)

```
template = /12/
```

Новый шаблон означает, что строка должна содержать уже две цифры — последовательность «12». Повторим тестирование ранее введенных строк (повтор ввода в консоли управляется стрелками «вверх» и «вниз» на клавиатуре) (рис. 20).

```

    Elements   Console   Sources
    top
    > template=/12/
    < /12/
    > template.test("123")
    < true
    > template.test("234")
    < false
    > template.test("I am 21 years old")
    < false
    >
  
```

Рисунок 20

В результате тест пройдет только первая строка, содержащая подстроку «12». В третьей строке, хотя и есть цифры «1» и «2», но стоят они не в заданном порядке «12», а значит, тест не проходится. Как мы видим, проверка на простые шаблоны представляет собой поиск подстроки (шаблона) внутри испытуемой строки.

Кроме простых символов шаблон может содержать некоторые специальные знаки, обобщающие в себе сразу несколько различных символов. Наиболее часто используемые выражения для специальных знаков приведены в следующей таблице. Их будет вполне достаточно для наших задач — проверки достоверности форм. Полный перечень спецзнаков можно узнать в справочниках по регулярным выражениям.

Шаблон	Описание
\d	Любая цифра
\s	Пробельный символ (пробел, табуляция, разрыв строки)
\w	Символ, допустимый в слове (англ. word—symbol): большая или маленькая буква, а также цифры или подчеркивание
. (точка)	Любой символ (кроме разрыва строки)
\D	Не цифра
\S	Не пробельный символ
\W	Символ, не являющийся \w
\x	Не x-цифра (цифры и буквы ABCDEF)

Например, использованный нами в качестве самого первого примера шаблон `/\D/` обозначает наличие знака «`\D`» в строке. Как видно из таблицы, этот специальный знак означает «не цифра». Успешную проверку таким шаблоном пройдут строки, содержащие в своем составе хоть один символ, не являющийся цифрой (в том числе пробел, запятую, точку и т.п.). Проверим новым шаблоном ранее введенные строки: поменяйте определение шаблона — введите в консоль

```
template = /\D/
```

После чего, повторяя набор команд стрелками «вверх» и «вниз» на клавиатуре, возобновим команды тестирования строк (рис. 21).

Как видно из рисунка, проверку с положительным результатом проходит только последняя строка, в которой действительно присутствуют не только цифры.

The screenshot shows a browser's developer tools with the 'Console' tab selected. The console output is as follows:

```

> template = /\D/
< /\D/
> template.test("123")
< false
> template.test("234")
< false
> template.test("I am 21 years old")
< true
>

```

Рисунок 21

Приведенные спецзнаки являются широко применяемыми для многих задач, однако, в каждом конкретном случае может возникнуть необходимость самостоятельно создать набор символов, не совпадающий со спецзнаками. Такая возможность в регулярных выражениях реализуется при помощи квадратных скобок «[]». В этих скобках перечисляются символы, входящие в желаемый набор. Обратите внимание, что наличие пробела или запятой внутри скобок будет считаться, что символ пробела или запятой также входит в набор. Если это не так, символы набираются без разделительных знаков.

Символы из набора проверяются по правилу «ИЛИ», то есть допускается совпадение с любым из перечисленных в скобках символов. Порядок перечисления символов на результаты проверки не влияет.

Пусть в качестве примера, нам нужно проверить строку на наличие цифр «1» или «2». Составляем набор допустимых символов и берем его в квадратные скобки

«[12]». Сам шаблон, ограниченный слешами, будет выглядеть следующим образом:

```
template = /[12]/
```

Проверим по этому шаблону наши ранее введенные строки (рис. 22).

The screenshot shows a browser's developer tools console window. The tabs at the top are 'Elements', 'Console' (which is selected), and 'Sources'. Below the tabs, there are several command-line entries and their results. The entries are:

- > template = /[12]/
- < /[12]/
- > template.test("123")
- < true
- > template.test("234")
- < true
- > template.test("I am 21 years old")
- < true
- >

Рисунок 22

Как видно, все они проходят проверку, так как в любой из них есть либо единица, либо двойка. Поменяйте в шаблоне порядок перечисления символов «[12]» и повторите проверки. Убедитесь, что результаты не поменяются.

В множестве допустимых символов можно указывать интервалы «от и до». Для этого применяется символ «-» (десфис). Несколько примеров приведено в следующей таблице:

Выражение	Значение
[1-4]	символы (цифры) от «1» до «4»
[a-z]	символы (буквы) от «а» до «z» (маленькие)
[A-Z]	символы (буквы) от «A» до «Z» (большие)
[г-ж]	символы (буквы) от «г» до «ж»
[0-9A-F]	шестнадцатеричные цифры 0123456789ABCDEF

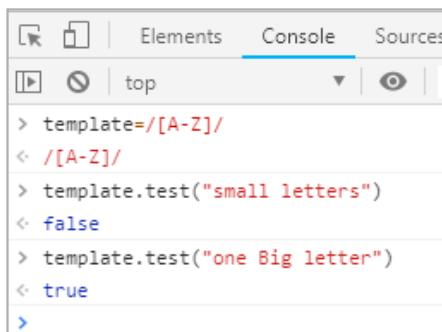
Для примера составим шаблон с набором символов «[A-Z]». В этот набор входят все большие буквы английского алфавита. Проверка этим шаблоном будет положительной, если строка содержит хоть одну большую английскую букву. Убедимся в этом на практике: введем определение шаблона в консоли браузера

```
template=/[A-Z]/
```

и затем проведем проверку строк, содержащих и не содержащих большие английские буквы. Введите следующие команды:

```
template.test("small letters")
template.test("one Big letter")
```

Как несложно убедиться, проверку с положительным результатом проходит только вторая строка, в которой есть одна большая буква «B». В первой строке все буквы маленькие и , как результат, проверка возвращает «false» (рис. 23).



The screenshot shows a browser's developer tools console tab labeled "Console". The console output is as follows:

```

Elements Console Sources
top
> template=/[A-Z]/
< /[A-Z]/
> template.test("small letters")
< false
> template.test("one Big letter")
< true
>

```

Рисунок 23

В случае, если перечисленные в наборе символы НЕ должны находиться в строке, используется символ отрицания «`^`» в начале перечисления. Например, `«[^12]»` — символы, кроме `«1»` или `«2»`, `«[^a-z]»` — всё, кроме символов от `«a»` до `«Z»`.

Введем новый шаблон, содержащий недопустимый набор символов

```
template = /[^123]/
```

Проверка данным шаблоном будет искать в строке символы кроме `«1»`, `«2»` или `«3»`. Повторим тест некоторых из предыдущих строк:

The screenshot shows a browser's developer tools console with the 'Console' tab selected. It displays the following interaction:

```

Elements   Console   Sources
top
> template = /[^123]/
< /[^123]/
> template.test("123")
< false
> template.test("234")
< true
> template.test("I am 21 years old")
< true
>

```

Рисунок 24

По данному шаблону тест не проходит только первая строка. Она содержит исключительно «запрещенные» символы (`123`) и других символов в ней нет. В остальных строках есть символы, кроме «запрещенных», поэтому проверка шаблоном заканчивается положительным результатом.

Если в строке ожидается несколько идущих подряд одинаковых символов или символов из одного набора,

то в шаблоне применяются указатели количества — квантификаторы (англ. *quantifiers*). После записи допустимого множества (заключенного в скобки «[]») или отдельного символа, записывается один из следующих квантификаторов:

Выражение	Значение
?	0 или 1 раз
+	1 и более раз
*	0 и более раз
{2}	ровно 2 повтора (число произвольное)
{2,}	2 и более повторов (число произвольное)
{2,7}	от 2 до 7 повторов (числа произвольные)

Указатель количества действует только на тот символ (или группу), после которого указывается и не влияет на предыдущие или последующие символы.

Например: шаблон «\d{7}» означает «семь цифр подряд», этот шаблон может использоваться для проверки 7-значных номеров телефонов. Шаблон «\s+» можно прочитать как «один или более пробельных символов», этот шаблон можно применять, если данные разделены несколькими символами, относящимися к группе пробельных.

Шаблон «https?» демонстрирует ограниченное действие указателей количества — модификатор «?» относится только к ближайшему символу «s» и не влияет на предыдущие. Значит, последовательность «http» обязательна, а символ «s» может быть 0 или 1 раз, по такому шаблону пройдут строки, содержащие либо «http», либо «https».

В регулярных выражениях могут применяться символы-якоря (англ. *anchors*). Эти символы указывают

на определенные места в строке, например, `^` — начало строки, `$` — конец строки.

Без применения якорей шаблон не привязывается к какому-либо месту строки и проводит поиск от начала до конца. Символы-якоря могут ограничить места поиска. Например, если нужно убедиться, что строка начинается с символа «`1`», то применим шаблон

```
template = /^1/
```

В данном шаблоне использован якорь начала строки `^`, после которого следует символ `1`. Значит, будет осуществлен поиск именно такой комбинации: начало строки, затем единица. Проверим работу этого шаблона — введем его определение в консоль и повторим тестирование строк (рис. 25).

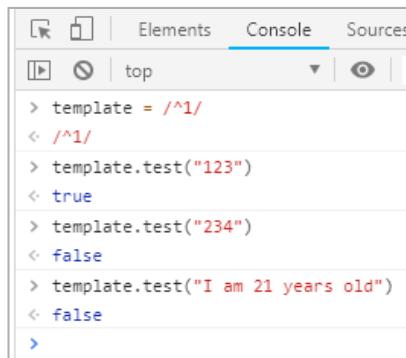


Рисунок 25

Как и ожидалось, тест проходит только первая строка, начинающаяся с единицы. Остальные строки тест про-валивают, хотя третья строка содержит единицу, но она находится не в начале строки.

Обратите внимание: символ «`\^`» имеет двойное значение. Если он стоит в начале группы `[\^...]`, то он «запрещает» символы, то есть его можно прочитать как «НЕ» или «КРОМЕ». Если же символ `\^` находится не в квадратных скобках, то он означает «начало строки» и является символом-якорем.

Составьте самостоятельно регулярный шаблон для проверки того, что

- а) строка заканчивается символом `\4`,
- б) строка заканчивается символом `\d`,
- в) строка заканчивается любой цифрой.

Проверьте работу шаблонов на используемых нами ранее строках.

Завершая обзор основных правил построения регулярных выражений отметим, что некоторые символы в них играют особую роль, указывая спецзнаки, группы, якоря и т.п. Для того чтобы включать подобные символы в шаблоны, подразумевая сами символы, а не их специальное назначение, их необходимо «экранировать» — добавлять обратный слеш `\` перед ними. Символы, которые необходимо экранировать следующие:

```
^ $ [ ] ( ) { } . * + ? < > | \
```

Например, если в строке нужно найти символ `[$]`, то шаблон должен содержать экранированный символ `\\$`, иначе он будет восприниматься как «конец строки» и алгоритм поиска изменится.

Введите в консоли браузера выражение для шаблона

```
template=/$/
```

И затем проверьте этим шаблоном строку «\$20 is better than \$10»

```
template.test("$20 is better than $10")
```

Результат проверки положительный и может показаться, что шаблон создан правильно. Повторите ввод команды (стрелка «вверх» на клавиатуре) и уберите символы «\$» из строки. Нажмите «Enter»

```
template.test("20 is better than 10")
```

Результат проверки снова положительный, хотя символов «\$» в строке уже нет.

The screenshot shows the browser's developer tools with the 'Console' tab selected. It displays the following command history:

```

> template=/$/           // Shows the template as /$/ in red
< $/
> template.test("$20 is better than $10") // Shows the test result as true in red
< true
> template.test("20 is better than 10")      // Shows the test result as true in red
< true
>

```

Рисунок 26

Почему же результат второй проверки оказался положительным? Потому что символ «\$» без экранирования (без обратного слеша) означает конец строки. А так как конец есть в любой строке, они все будут проходить положительный тест данным шаблоном.

Для исправления ситуации следует поменять шаблон

```
template=/\$/
```

Повторите тест ранее введенных строк и убедитесь, что вторая проверка возвращает «`false`», чего и требовалось добиться.

The screenshot shows a browser's developer tools with the 'Console' tab selected. The console output is as follows:

```

> template=/\$/ 
< /\$/
> template.test("$20 is better than $10")
< true
> template.test("20 is better than 10")
< false
>

```

Рисунок 27

Найдите ошибку: мы хотим проверить строку на предмет номера телефона в международном формате, наподобие «`+380512670053`» — 12 цифр (специальный символ «`\d`») и знак «`+`» в начале строки («`^+`»), кроме номера, других символов не допускается, то есть после цифр должен следовать конец строки («`$`»). Правильным ли будет следующий шаблон:

```
template = /^+\d{12}$/
```

Ответ: шаблон неправильный, так как знак «`+`» является специальным квантификатором, означающим «один и более раз». Его необходимо экранировать. «`\+`». Правильная запись шаблона будет выглядеть как

```
template = /^\\+\\d{12}$/
```

Методы объектов String и RegExp для работы с регулярными выражениями

В предыдущем разделе мы рассмотрели основные правила формирования регулярных выражений. Далее рассмотрим несколько вариантов их применения и отметим роль флагов в регулярных шаблонах.

Работу с регулярными шаблонами обеспечивают как объекты типа `RegExp`, так и объекты типа `String`. С одним из методов объекта `RegExp` «`test`» мы уже познакомились. Напомним — этот метод позволяет проверить, подходит ли строка под данный шаблон, и возвращает результат этой проверки.

Еще один метод объекта `RegExp` «`object/null exec(str)`» имеет расширенные возможности по сравнению с методом «`test`». Параметром метода также является строка, обрабатываемая регулярным выражением. Кроме проверки строки на соответствие шаблону, метод «`exec`» обеспечивает поиск тех фрагментов исходной строки, которые совпали с шаблоном. В качестве результата своей работы метод возвращает объект с описанием найденного фрагмента либо значение «», если совпадений с шаблоном найдено не было.

Давайте рассмотрим работу метода «`exec`» на примере. Введем в консоль строку (можно скопировать из данного текста и вставить в консоль браузера)

```
str = "20% of population owning 80% income"
```

и шаблон для поиска чисел (цифра «\d», повторяющаяся один или более раз «+»)

```
template = /\d+/
```

Выполним метод «exec» шаблона «template» над строкой «str». Введем в консоль следующую команду, нажав «Enter» в конце ее набора.

```
template.exec(str)
```

В качестве ответа получим некоторый объект. Раскроем его содержимое, нажав на символ «▶» слева от полученного в консоли результата. Под индексом «[0]» в ответе содержится первый найденный фрагмент, совпавший с шаблоном, то есть первое число «20». Поле «index» содержит информацию о позиции найденного числа в строке. В данном случае это значение «0», то есть число найдено в самом начале строки.

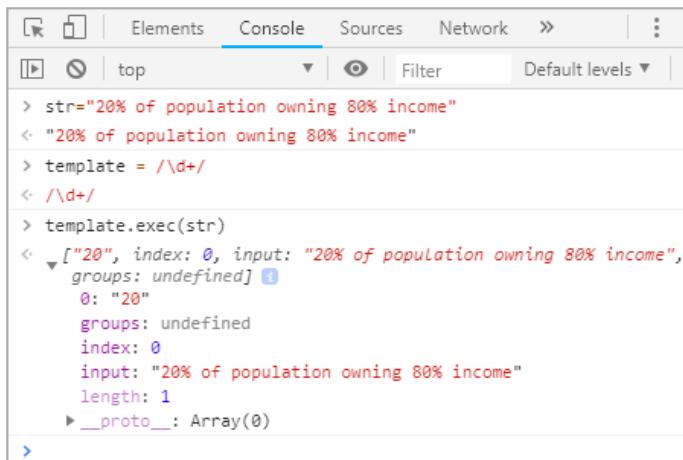


Рисунок 28

Попробуем повторно выполнить команду «`template.exec(str)`» — получим тот же самый ответ с найденным числом «`20`» в начале строки. Сделаем вывод — метод «`exec`» останавливается после обнаружения первого совпадения с шаблоном.

Вполне логичный вопрос «А как же получить следующие числа в строке?» приведет нас к использованию дополнительного условия поиска, которое устанавливается флагами. В нашем случае следует применить флаг «`g`», отвечающий за глобальный поиск (*от англ. global*). С использованием флага шаблон примет вид

```
template = /\d+/g
```

При наличии флага «`g`» метод «`exec`» можно вызывать несколько раз, получая новые совпадения с шаблоном. В случае, если очередное совпадение не будет найдено, метод «`exec`» вернет значение «`null`».

Ведите в консоль новое определение шаблона и вызовите его метод «`exec`» с ранее введенной строкой «`str`».

```
template.exec(str)
```

В результате, после первого запуска получится тот же ответ, что и в прошлый раз. Однако повторный вызов метода «`exec`» приведет к новому результату, содержащему следующее найденное число — `80`. Поле «`index`» содержит значение «`25`», отвечающее за позицию этого числа в исходной строке. Третий вызов метода закончится возвратом «`null`», что означает невозможность найти следующее совпадение с шаблоном.

The screenshot shows the Chrome DevTools console interface. The 'Console' tab is selected. The command `template = /\d+/g` creates a regular expression object. The command `template.exec(str)` is run twice on the string `str = "20% of population owning 80% income"`. The first execution returns an array with index 0 containing the value '20'. The second execution returns an array with index 25 containing the value '80'. Both executions return objects with properties: `index` (the position in the string), `input` (the original string), `groups` (undefined), and `length` (1). The `__proto__` property is also shown. Finally, `template.exec(str)` is run again, returning `null` because there are no more matches.

```

> template = /\d+/g
< /\d+/g
> template.exec(str)
< ["20", index: 0, input: "20% of population owning 80% income",
  groups: undefined]
  0: "20"
  groups: undefined
  index: 0
  input: "20% of population owning 80% income"
  length: 1
  > __proto__: Array(0)
> template.exec(str)
< ["80", index: 25, input: "20% of population owning 80% income",
  groups: undefined]
  0: "80"
  groups: undefined
  index: 25
  input: "20% of population owning 80% income"
  length: 1
  > __proto__: Array(0)
> template.exec(str)
< null
>

```

Рисунок 29

Задание: создайте инструкцию поиска всех чисел в строке при помощи условного цикла.

Подобной с методом «exec» функциональностью обладает метод «object/null match(RegExp)» у объекта String. Напомним, метод «exec» принадлежит объекту «RegExp». Метод «match» также проверяет строку на соответствие шаблону и, в зависимости от результатов проверки, возвращает объект с найденными фрагментами, либо «null», если совпадений с шаблоном не найдены. В отличие от метода «exec» результаты работы метода «match» будут разными для шаблонов с флагом «g» и без него.

При записи этого метода программная инструкция как бы «переворачивается» — метод «[match](#)» вызывается у строковой переменной ([str](#)), а в качестве аргумента ему передается регулярный шаблон ([template](#)):

```
str.match(template)
```

Посмотрим, как работает данный метод и как он реагирует на наличие флага глобального поиска. Введем определение шаблона без флага и вызовем метод [«match»](#).

```
template = /\d+/
str.match(template)
```

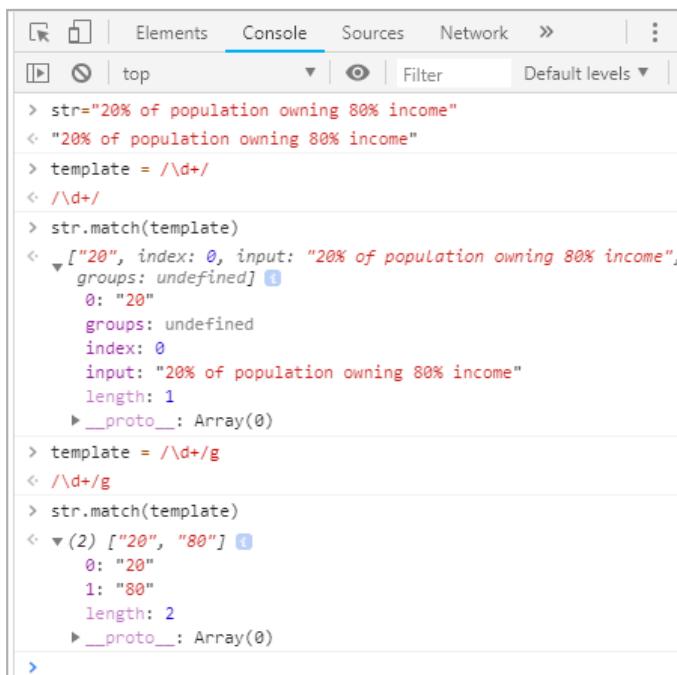


Рисунок 30

Раскроем детали полученного в консоли ответа (см. рис. 30). Затем введем шаблон с флагом глобального поиска и повторим вызов метода

```
template = /\d+/g  
str.match(template)
```

В результате должно получиться так, как приведено на рисунке 12.

Из анализа результатов можно сделать следующие выводы. Без флага глобального поиска метод [«match»](#) работает так же, как и рассмотренный выше метод [«RegExp.exec»](#). Однако при наличии флага результат получается другой — в итоговом ответе сразу собраны все совпадения с шаблоном: оба числа [«20»](#) и [«80»](#) попали в итоговый массив. Правда при этом отсутствует информация об индексах этих чисел в строке.

Данные отличия позволяют выбрать один из рассмотренных методов в зависимости от желаемых результатов проверки. Если нужны сами числа, то удобнее будет получить их за один запрос методом [«String.match»](#) не создавая циклы. Если же нужны данные о положении этих чисел в строке, то предпочтение можно отдать методу [«RegExp.exec»](#), который эту информацию предоставляет в явном виде. Всё зависит от конкретной задачи, стоящей перед программистом.

Еще одним методом объекта [«String»](#), работающим с регулярными выражениями, является [«replace»](#). Этот метод позволяет выполнить замену найденных совпадений на другие значения.

Например, мы допускаем, что номер телефона может быть введен пользователем с использованием тире [«67-](#)

00-53». Для ввода это допустимо, но для хранения в базе данных эти тире следует убрать. Используем для этого «`replace`»: введем в консоль новое определение строки «`str`» и вызовем ее метод «`replace`»:

```
str="67-00-53"
str.replace("-", "")
```

В метод «`replace`» передается два параметра. Первый — это то, что нужно заменить. Второй — строка для замены. В нашем случае заменяется символ тире (" - ") на пустую строку (""), это означает, что символы тире просто будут исключены из строки.

Выполним эти команды и проанализируем результат:

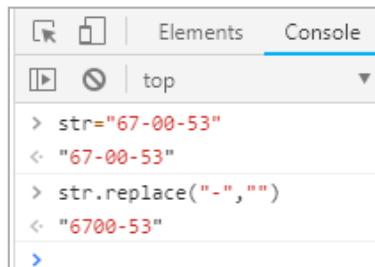


Рисунок 31

Как видно, исчез только первый символ тире, второй остался без изменений. Отмечаем для себя, что метод «`replace`» именно так и работает — заменяет только первое найденное совпадение (с подобным поведением мы уже сталкивались чуть раньше, не так ли?).

Вы уже наверняка догадались, что для поиска всех совпадений следует использовать регулярное выражение с флагом глобального поиска. Проверим нашу догадку —

заменим шаблон поиска на регулярный (не забудьте, что для регулярного шаблона кавычки не нужны, а для строки замены кавычки должны остаться). Добавьте в консоль следующую команду

```
str.replace(/-/g, "")
```

Ее результатом будет полная замена всех тире в строке.

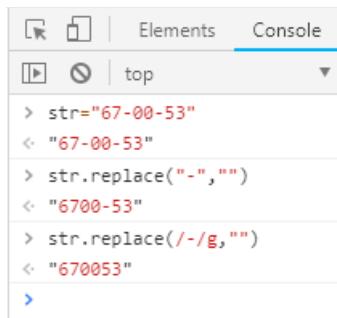


Рисунок 32

Расширим условие задачи — в номере телефона при вводе допускаются, кроме тире, еще и пробелы. А в ко- нечном результате их нужно исключить.

Поскольку нам нужно искать множество символов, используем их группировку квадратными скобками. Итоговый шаблон примет вид `«/[-]/g»` (возможно, в до- кументе это не очень хорошо видно, но после знака «-» набирается пробел). Проверим работу этого шаблона. Заменим строку `«str»`, добавив в ее определение пробелы, и вызовем метод `«replace»` с новым шаблоном

```
str = "67 - 00 - 53"
str.replace(/[- ]/g, "")
```

Убедимся, что из строки исчезли и тире, и пробелы.

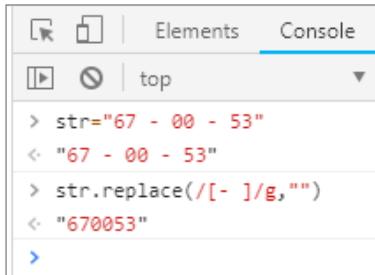


Рисунок 33

Задание: создайте шаблон и проверьте его работоспособность для ситуации, когда в номере телефона допускаются еще и круглые скобки. Для проверки можете использовать телефон в формате [«\(0512\) 67-00-53»](#).

Третьим методом объекта [«String»](#) (после [«match»](#) и [«replace»](#)), работающим с регулярными выражениями, является метод [«split»](#). Этот метод предназначен для разбиения строки по определенному шаблону.

В простейшем случае метод [«split»](#) можно использовать без регулярных выражений с обычными строками-разделителями. Например, инструкция

```
str.split(",")
```

Разделит строку на части, разделенные запятыми [\(","\)](#). Результатом работы метода будет массив, состоящий из полученных частей строки.

В более сложных случаях в качестве разделителя можно указывать регулярное выражение. Например, пусть в некоторое поле формы вводится фамилия и имя

пользователя. Наша задача — разделить их. При этом мы предусматриваем, что пользователь при вводе может поставить несколько пробелов («Smith John») или отделить фамилию от имени запятой («Smith, John»).

Поскольку в данном примере мы работаем со словами, в качестве разделителя нам удобно будет воспользоваться спецсимволом «\W», означающий символы, не применяемые в словах. Именно эти символы и отделяют слова друг от друга. С учетом того, что этих символов может быть сразу несколько подряд (например, запятая и два пробела), укажем для них квантификатор «+». В итоге, шаблон примет вид «\W+».

Проверим работу этого шаблона с методом «split». Создадим две строки с разными вариантами ввода фамилии и имени

```
str1 = "Smith John"  
str2 = "Smith, John"
```

После чего зададим шаблон и вызовем методы «split» у каждой из строк

```
template = /\W+/  
str1.split(template)  
str2.split(template)
```

В результате каждого вызова мы получаем одинаковые массивы, состоящие из двух разделенных элементов — фамилии и имени. В самих разделенных строках нет ни пробелов, ни запятых.

```

Elements Console
top
> str1="Smith John"
< "Smith John"
> str2="Smith, John"
< "Smith, John"
> template = /\W+/
< /\W+/
> str1.split(template)
< ▶ (2) ["Smith", "John"]
> str2.split(template)
< ▶ (2) ["Smith", "John"]
>

```

Рисунок 34

Задание: в поле ввода телефона допускается вписывать несколько номеров, разделяя их запятой. Естественно, перед и после нее возможны пробелы в любом количестве. Напишите регулярное выражение и проверьте его методом `«split»` для разделения строки на отдельные номера телефонов. Для тестирования можете использовать строку `«67-00-53 , 670053 , +380512670053»`.

В завершение знакомства с работой регулярных выражений рассмотрим еще один часто используемый на практике флаг `«i»`, означающий регистронезависимый анализ шаблона. Напомним, что регистронезависимость означает проведение сравнений, одинаковых как для больших букв (символов верхнего регистра), так и для маленьких (нижнего регистра). Обычное сравнение (без флага `«i»`) является регистрозависимым, то есть большие и маленькие буквы считаются разными символами.

Пусть, в качестве примера, нам нужно получить от пользователя подтверждение, содержащее в себе слово «yes». На этапе проектирования предположим несколько вариантов ввода, которые мы будем считать допустимыми:

```
str1="Yes, of course"  
str2="well, let it be yes"  
str3="I said YES!"
```

Нам нужно убедиться, что строка содержит слово «yes», написанное маленькими или большими буквами, а также их комбинацией. Для таких целей как раз и подходит флаг «i». Зададим шаблон

```
template = /yes/i
```

и выполним проверку строк по этому шаблону, вводя последовательно следующие команды

```
template.test(str1)  
template.test(str2)  
template.test(str3)
```

Убедимся, что все они приводят к истинному результату. Для того чтобы убедиться в том, что другие строки будут приводить к отрицательным проверкам, введем несколько проверок, не содержащих слова «yes»

```
template.test("good")  
template.test("OK")
```

Увидим, что в таком случае результат ложный.



The screenshot shows a browser's developer tools with the 'Console' tab selected. The session log displays the following interactions:

```
> str1="Yes, of course"
< "Yes, of course"
> str2="well, let it be yes"
< "well, let it be yes"
> str3="I said YES!"
< "I said YES!"
> template = /yes/i
< /yes/i
> template.test(str1)
< true
> template.test(str2)
< true
> template.test(str3)
< true
> template.test("good")
< false
> template.test("OK")
< false
>
```

Рисунок 35

Проверка достоверности данных формы

В данном разделе рассмотрим несколько практических примеров применения регулярных выражений для проверки достоверности данных форм.

Для этого создадим форму-анкету, предназначенную для добавления нового пользователя в некоторую систему. Введем поля для наиболее популярных данных — реального имени, логина/пароля, телефона и электронной почты.

Создайте новый HTML-документ, наберите или скопируйте в него следующий код (*пример также доступен в папке «Sources_6», файл «JS6_1.html»*).

```
<!DOCTYPE HTML>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;
        charset=UTF-8">
    <title>Registration form</title>
    <style>
        #regForm b{
            display:inline-block;
            padding-top:5px;
            width:100px;
        }
        #regForm input[type="submit"]{
            margin: 15px 150px;
        }
    </style>
</head>
```

```
<body>
<form method = "GET" id = "regForm"
      onsubmit = "return checkForm()" >

    <b>First name:</b>
    <input type = "text" id = "name1"
           name = "name1" required /><br/>

    <b>Last name:</b>
    <input type="text" id="name2" name="name2"
           required /><br/>

    <b>Login:</b>
    <input type="text" id="login" name="login"
           required /><br/>

    <b>Password:</b>
    <input type="password" id="pas1"
           name="pass" required /><br/>

    <b>Confirm password:</b>
    <input type="password" id="pas2" required /><br/>

    <b>Email:</b>
    <input type="email" id="email" name="email"
           required /><br/>

    <b>Phone number:</b>
    <input type="phone" id="phone" name="phone"
           required /><br/>

    <b>Code:</b>
    <input type="number" id="code" name="code"
           required /><br/>

    <b></b><label>
    <input type="checkbox" name="accept" required />
    I accept the rules</label><br/>
    <input type="submit" value="Register" />
</form>
```

```
<script>
    function checkForm() {
        alert("Analyzing...") ;
        return false;
    }
</script>
</body>
</html>
```

Сохраните файл и откройте его в браузере. Страница должна соответствовать следующему рисунку

The form consists of eight input fields and two buttons. The fields are labeled: First name, Last name, Login, Password, Confirm password, Email, Phone number, and Code. Each label is followed by a horizontal input field. Below these fields is a checkbox labeled "I accept the rules". At the bottom right is a blue "Register" button.

First name:	<input type="text"/>
Last name:	<input type="text"/>
Login:	<input type="text"/>
Password:	<input type="text"/>
Confirm password:	<input type="text"/>
Email:	<input type="text"/>
Phone number:	<input type="text"/>
Code:	<input type="text"/>
<input type="checkbox"/> I accept the rules	
Register	

Рисунок 36

Обратите внимание: для всех элементов формы (кроме кнопки submit) указан атрибут «**required**». Это означает, что браузер будет автоматически проверять наличие данных в этих полях. Попробуйте, не вводя никаких данных нажать кнопку «**Register**». Появится предупреждающая надпись, и отправка формы не произойдет (это можно заметить по отсутствию данных в адресной строке браузера).

The screenshot shows a registration form with the following fields and validation message:

- First name:** (Input field)
- Last name:** (Input field)
- Login:** (Input field)
- Password:** (Input field)
- Confirm password:** (Input field)
- Email:** (Input field)
- Phone number:** (Input field)
- Code:** (Input field)

A validation message is displayed above the input fields: **Please fill out this field.** This message is associated with the "First name" field, which is highlighted with a blue border.

Below the input fields is a checkbox labeled **I accept the rules**. At the bottom right is a **Register** button.

Рисунок 37

Как уже было отмечено выше, контролируется только пустота полей, если в них ввести произвольные данные, то проверка будет пройдена. Можете в этом убедиться, нажав пробел в полях для ввода имен, логина, пароля.

Следующий момент, на который мы обращаем внимание, это выбор наиболее подходящих типов для элементов формы. Так, для ввода пароля применен тип **«type="password»**, для электронной почты **type="email"** и т.д. Это позволяет некоторую часть работы также переложить на браузер. Например, пароли автоматически скрываются, а контроль ввода электронной почты уже произвольной кнопкой не пройдешь. Попробуйте вписать некоторый символ, например **«1»** в поле для почты и нажмите кнопку **«Register»**. Браузер выдаст предупреждение о некорректном адресе и форма не отправится (рис. 38).

First name:

Last name:

Login:

Password:

Confirm password:

Email:

! Please include an '@' in the email address. '1' is missing an '@'.

I accept the rules

Register

Рисунок 38

Следующим шагом при проверке достоверности данных является перехват события отправки формы «`onsubmit`». Для этого мы в открывающем теге формы указываем атрибут `onsubmit="return checkForm()"`, а в скриптовой части создаем функцию `«checkForm()`», ответственную за обработку формы.

На данном этапе функция просто выводит сообщение о начале анализа данных и возвращает `«false»` в любом случае. Это должно предотвратить отправку формы. Для того чтобы убедиться в этом заполните все поля и нажмите кнопку `«Register»`. На странице появится сообщение, после закрытия которого адресная строка браузера не меняется и данные из полей формы не исчезают. Это признаки того, что форма отправлена не была и функция `«checkForm()`» выполнилась успешно (рис. 39).

На данном этапе мы отменили автоматическую отправку формы, получив возможность провести более

детальный анализ введенных пользователем данных. Не забывайте проверять работоспособность каждого этапа при создании собственного проекта.

The screenshot shows a registration form with the following fields and their values:

- First name:** Web
- Last name:** Programmer
- Login:** WP
- Password:**
- Confirm password:**
- Email:** wp@it.step
- Phone number:** 670053
- Code:** 3257

Below the form is a checkbox labeled "I accept the rules" with a checked state. At the bottom center is a "Register" button.

A modal dialog box is displayed on the right side of the form, containing the text "This page says" and "Analyzing...". In the top right corner of the dialog box is a blue "OK" button.

Рисунок 39

Теперь перейдем, собственно, к анализу данных. Воспользуемся для этого механизмом регулярных выражений.

Для того чтобы правильно составить регулярный шаблон для проверки каких-либо данных необходимо четко и полностью сформулировать ограничения, которые на эти данные накладываются. Любые дальнейшие решения, на подобие «а давайте еще вот это добавим» вполне вероятно могут потребовать существенной переработки шаблона, а, возможно, и всего алгоритма проверки. Составить универсальный шаблон на все случаи жизни нет смысла, его нужно создавать для конкретных правил.

Рассмотрим на конкретном примере, как изменяющиеся требования могут повлиять на структуру шаблонов. Будем составлять правила для проверки имен и фамилий.

Правило 1. В имени (и фамилии) могут быть только буквы. Для простоты ограничимся буквами английского алфавита. Поначалу кажется, что данных ограничений вполне достаточно для анализа имени.

Заданное правило проще всего проверять «от обратного» — если в строке есть что-то, кроме букв, то проверка считается не пройденной. Создаем шаблон `«/[^\wedge a-z]/i»`, означающий любой символ кроме набора `«a-z»`. Флаг `«i»` означает сравнение без учета регистра, допуская в имени как большие, так и маленькие буквы. Сохраняем шаблон в локальной переменной `«t1»`. Также в функции `«checkForm()»` получаем данные из поля `«name1»`, сохраняем их в локальной переменной `«n1»` и организовываем проверку шаблоном `«t1»`. Функция примет вид:

```
function checkForm() {
    var n1 = document.getElementById("name1").value;
    var t1 = /[^\wedge a-z]/i;
    if(t1.test(n1))
        alert("First name is incorrect");
        return false;
}
```

Сохраните измененный файл и обновите страницу в браузере. Проверьте работу новой функции проверки — вводите в поле `«First name»` различные имена правильного и неправильного написания (с цифрами, например). Во втором случае будет выдаваться предупреждение о некорректном имени.

Правило 2. Всё то же самое, что и в первом правиле, только первая буква имени должна быть большой. Такое

незначительное изменение в правилах проверки приводит к коренному пересмотру всего алгоритма обработки имени. Во-первых, придется отказаться от регистронезависимой проверки, т.к. надо разделять большие и маленькие буквы. Во-вторых, появляется привязка к началу строки, поскольку речь идет именно о первой букве. Как следствие, тест надо строить не «от противного», а при прямом соответствии регулярной маске.

Сам шаблон примет вид `/^[A-Z][a-z]*$/` — начало строки (`^`), далее один из символов `[A-Z]`, далее символы `[a-z]` в произвольном количестве (квантификатор `*`). Функция `«checkForm()»` преобразуется следующим образом: (обратите внимание, теперь проверка `«t1.test(n1)»` инвертируется знаком `«!»`)

```
function checkForm() {
    var n1 = document.getElementById("name1").value;
    var t1 = /^[A-Z][a-z]*$/;
    if(!t1.test(n1))
        alert("First name is incorrect");
    return false;
}
```

Сохраните файл с новой функцией, обновите страницу браузера и проверьте работу нового алгоритма для различных написаний имен с большой и с маленькой буквой в начале, а также с недопустимыми символами в своем составе. Тем временем, в службу техподдержки поступила жалоба от пользователя по имени Jean-Paul (Жан-Поль). Разобрав ее, было принято решение, что программистам необходимо заменить алгоритм анализа имен и фамилий. Ведь и фамилии тоже часто бывают составными.

Правило 3. Составные имена нужно проверять каждое отдельно по шаблону «Большая буква» далее «маленькие» потом «тире» и снова «Большая», затем «маленькие». Причем составной части может и не быть, а может быть и большее количество составных частей.

Новый шаблон примет вид `/^ [A-Z] [a-z]* (- [A-Z] [a-z]*)* $/`. Первая его часть совпадает с предыдущим шаблоном. Далее в круглых скобках мы ставим тире и снова повторяем шаблон — это вторая часть имени. После круглых скобок ставим квантификатор `*`, это означает, что второй части, взятой в скобки, может не быть, а может быть и несколько повторов для сложносоставных имен.

Замените шаблон в проверочной функции и проведите проверку его работоспособности.

Для формирования следующего правила вспомним о таких всемирно известных именах (и фамилиях) как d'Artagnan или O'Brian. Первая буква в них может не быть большой и после нее возможен апостроф. Будем допускать, что правильным также будет написание имени без апострофа (dArtagnan или OBrian). Итак, правило 4: первая буква имени может быть маленькой, вторая может быть большой, после первой буквы возможен апостроф (одинарная кавычка).

Снова меняем шаблон. Если первая буква может быть как большой, так и маленькой, набор символов следует расширить `«/^ [A-Za-z]»`, дальше возможен апостроф, то есть продолжаем шаблон апострофом с квантификатором «один или ноль» (знак вопроса) `«'?»`. Затем снова следует символ, который может быть большим либо маленьким `«[A-Za-z]»` и далее — произвольный набор малых симво-

лов до конца строки «`[a-z]*$`». Соединив части шаблона вместе, получим следующее регулярное выражение

```
/^ [A-Za-z] ' ? [A-Za-z] [a-z]*$/
```

Замените проверочный шаблон в функции `«checkForm()`» и проверьте его работоспособность аналогично тому, как это делалось ранее.

Задание: дополните последний шаблон возможностью составной части имени (или фамилии) по такому же образцу.

В завершение анализа различных правил для построения шаблонов проверки имен и фамилий отметим, что построенные шаблоны все равно не охватывают весь спектр различных имен. Чем больше мы будем усложнять шаблон, тем с большей вероятностью шаблон «пропустит» неправильное имя. Например, по третьему шаблону правильным будет считаться имя «`A-A-A`». Можете убедиться в этом, введя это имя в поле формы или отдельно в консоли, набрав (или скопировав из урока) строку

```
/^ [A-Z] [a-z]* (- [A-Z] [a-z]*) *$/ .test ("A-A-A")
```



Рисунок 40

Напомним еще раз основные выводы:

- универсальные шаблоны не существуют, для них всегда можно найти исключение, когда либо правильная строка не пройдет шаблон, либо неправильная будет успешно проверена.
- для составления шаблона нужно заранее четко сформулировать правила, иначе работа над шаблоном может превратиться в самостоятельную бесконечную задачу.

Разобравшись с особенностями и сложностями проверки имен и фамилий, перейдем к анализу паролей. Следуя приведенным рекомендациям, сразу сформулируем правила и не будем их переделывать на новые в течение работы. К тому же, в отличие от имен, правила для паролей — это полностью в нашей власти.

Будем требовать, чтобы пароль содержал хотя бы один символ нижнего регистра, один — верхнего и одну цифру. Других ограничений накладывать не будем.

Поскольку требования задаются на содержание определенных символов и не ограничивают наличие других символов, их проверку будем проводить «от обратного». Если строка не проходит тест шаблоном `\d` (цифра), значит, в строке нет ни одной цифры. Таким образом, мы можем проверить введенный пароль три раза на наличие каждого из необходимых символов.

Дополним проверочную функцию кодом анализа пароля

```
var p1 = document.getElementById("pas1").value;
if(! /\d/.test(p1))
    alert("Password has no digit");
```

```
if(! /[A-Z]/.test(p1))
    alert("Password has no big letters");
if(! /[a-z]/.test(p1))
    alert("Password has no small letters");
```

В переменной «`p1`» сохраняется введенное пользователем значение, затем проводятся три проверки разными шаблонами. В данном примере шаблоны не сохраняются в отдельные переменные, а применяются непосредственно в условных инструкциях.

Проверку повторного ввода пароля можно провести обычным сравнением без использования регулярных выражений. Реализуйте ее самостоятельно.

Дальше перейдем к проверке достоверности электронной почты. Следует отметить, что правила проверки электронных адресов вызывают наиболее горячие дискуссии и споры в среде веб-разработчиков. Причин этому несколько.

Во-первых, стандарты составления адресов постоянно обновляются, дополняются или упраздняются. Перечень стандартов можно посмотреть на справочных ресурсах Интернета, например [Википедия](#). От этого и сами адреса могут приобрести или потерять валидность.

Во-вторых, имеются различия между внутренними и внешними адресами. Например, для внутренних нужд может применяться адрес «`user@localhost`», тогда как для внешних адресов нужны составные имена доменов (с точкой) «`user@itstep.org`». Значит, правила проверки также следует корректировать для разных сетевых проектов.

В-третьих, не все почтовые серверы поддерживают полный набор стандартизованных правил, особенно их нововведений. В частности, согласно стандартам, вполне легальным почтовым именем является один пробел «`@ itstep.org`». Можете попробовать создать такой ящик на каком-либо домене. Также стандарты предусматривают регистрозависимость адресов, то есть различие больших и маленьких символов. Значит, адреса `«user@itstep.org»` и `«User@itstep.org»` должны считаться разными. Тем не менее, большинство почтовых серверов их не будут различать между собой.

Перечень различных регулярных выражений для адресов электронной почты в соответствие с разными стандартами можно посмотреть в Интернете, например, на сайте <https://emailregex.com/>. Разберем одно из них, имеющее вид

```
^\w+([-+.']\w+)*@\w+([-.\ ]\w+)*.\w+([-.\ ]\w+)*$
```

Его основу составляет «символ слова» `\w`. Описание этого символа приведено в таблице второго раздела урока.

Адрес должен начинаться хотя бы одним из таких символов `«^|\w+»` (квантификатор `«+»` предусматривает «один и более» символов). В дальнейшем, могут добавиться символы `«-+.」`, но эти символы не могут быть последними в имени. Значит, после них должен снова появиться хотя бы один словарный символ `«|\w+»`. Эта комбинация взята в группу круглыми скобками `«([-.]|\w+)»`, после группы указан квантификатор `«*»`, допуская как наличие, так и отсутствие этой группы. Замет следует символ `«@»`. Далее приведено выражение, совпадающее с именем, только из дополнительных

символов остаются только два «[.-]». Затем символ точки, он должен быть экранирован «\.» в регулярном шаблоне. И снова то же выражение, что и до точки.

По правилам JavaScript регулярное выражение должно быть ограничено в начале и конце прямыми слешами «/>. Соответственно, дополним нашу функцию «`checkForm()`» средствами проверки электронного адреса:

```
var eMail = document.getElementById("email").value;
var te = /^\\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*$/ ;
if(! te.test(eMail) )
    alert("e-Mail is incorrect");
```

Некоторые проверки правильности почтового адреса браузер возьмет на себя, поэтому полноценное испытание шаблона можно провести, поменяв тип элемента формы на обычный текстовый. Также можно попробовать шаблон на примерах в консоли. Если же ничего не менять, попробуйте ввести в поле для почтового адреса выражение «`user*2@itstep.org`». Согласно стандартам, это легальное выражение и первичную проверку должно пройти. В наших правилах мы не допускаем символа «`*`» и должно появиться предупреждение «`e-Mail is incorrect`» из функции «`checkForm()`».

Примеры шаблонов для проверки номера телефона мы уже разбирали в предыдущем разделе. Сформулируйте самостоятельно набор правил для проверки и составьте регулярное выражение для них.

Простейшей в нашем случае проверкой будет ситуация с «кодом». Считаем, что это некоторый код подтверждения

и в нем могут быть только цифры. Такой пример мы также рассматривали выше при анализе пароля, используйте его для проверки ввода пользователя.

Последняя проверка на наличие отметки о согласии с корпоративными правилами не требует применения регулярных выражений. Достаточно обычным образом проверить значение этого элемента формы на величину «`on`», устанавливаемую в случае наличия отметки на нем.

После введения и проверки работоспособности всех шаблонов можно разрешить отправку формы. Это реализуется возвратом значения «`true`» из функции проверки «`checkForm()`».

Полное описание скриптовой части для проверки валидности созданной формы приводится далее (*код также доступен в папке «Sources_6», файл «JS6_2.html»*).

```
<script>
function checkForm() {
    var n1 = document.getElementById("name1").value;
    var t1 = /^[A-Z][a-z]*(-[A-Z][a-z]*)*$/;
    if(!t1.test(n1)){
        alert("First name is incorrect");
        return false;
    }
    var n2 = document.getElementById("name2").value;
    if(!t1.test(n2)){
        alert("Last name is incorrect");
        return false;
    }
    var p1 = document.getElementById("pas1").value;
    if(! /\d/.test(p1)){
        alert("Password has no digit");
        return false;
    }
}
```

```
if(! /[A-Z]/.test(p1)) {
    alert("Password has no big letters");
    return false;
}
if(! /[a-z]/.test(p1)) {
    alert("Password has no small letters");
    return false;
}
var p2 = document.getElementById("pas2").value;
if(p1 != p2) {
    alert("Passwords mismatch");
    return false;
}
var eMail = document.getElementById("email").value;
var te = /^\\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+
        ([-.]\\w+)*$/ ;
if(! te.test(eMail) ){
    alert("e-Mail is incorrect");
    return false;
}
var phone = document.getElementById("phone").
            value;
if(! ^\\+?\\d{6,12}$.test(phone) ){
    alert("Phone is incorrect");
    return false;
}
var cod = document.getElementById("code").value;
if(/\\D/.test(cod) ){
    alert("Code is incorrect");
    return false;
}
var agree = document.getElementById("accept").
            value;
if(agree!="on"){
    alert("You should accept rules");
    return false;
}
```

```
    return true;  
}  
</script>
```

Обратите внимание, что в случае успешной отправки формы страница перезагружается, из-за чего введенные в поля данные исчезают, а в адресной строке браузера появляются имена передаваемых параметров и их значения.

Если же проверка данных не приводит к положительному результату, то появляется предупреждение либо от браузера, либо от проверочной функции, введенные данные не очищаются, адресная строка браузера не изменяется. Проверьте работу созданной нами формы в различных ситуациях.

Что такое cookie?

Перейдем к рассмотрению второй части процесса обмена данными между клиентом и сервером. Напомним, что первая часть состояла в отправке информации от клиента на сервер, для этого были предназначены HTML-формы. Следующий этап — получение клиентом ответа от сервера.

Самой главной особенностью, которая влияет на подготовку сервером ответа клиенту, заключается в том, что сервер, сам по себе, не хранит историю взаимодействия с разными клиентами. «Общение» клиента и сервера всегда носит одноразовый характер. В момент установки соединения клиент и сервер обмениваются данными, но после разрыва соединения ни клиент, ни сервер, по умолчанию, не имеют данных о предыдущих соединениях (сеансах).

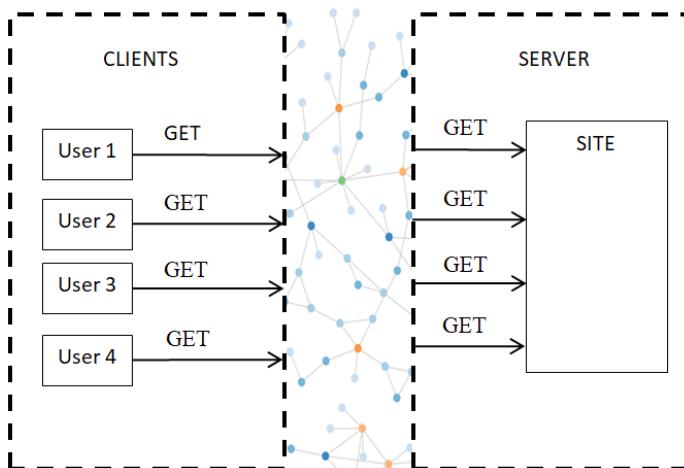


Рисунок 41

С учетом того факта, что запросы от различных клиентов идут к серверу различными путями, проходя множество промежуточных узлов и коммутаторов, у сервера нет практически никаких способов сопоставить запросы, которые к нему приходят, с клиентами. На приведенном рисунке это означает, что невозможно однозначно и правильно сопоставить стрелки, исходящие от клиентов, и стрелки, приходящие к серверу.

Одной из возможностей хоть как-то различать клиентов со стороны сервера является анализ IP-адреса, но это ненадежно как минимум по двум причинами. Во-первых, у клиента может поменяться IP, например, если сотовый телефон переходит из зоны действия одной сотовой в другую. У стационарных компьютеров такое случается реже, но тоже возможно.

Во-вторых, если несколько компьютеров включены в локальную сеть одного предприятия, то для внешнего сервера все они будут иметь один и тот же IP-адрес, установленный прокси-сервером предприятия. Если анализировать только IP, то действия, совершенные на одном компьютере, могут повлиять на ответ сервера для другого компьютера.

Для того чтобы как-то «пометить» различных клиентов сервер может добавить к своему ответу некоторые данные, уникальные для каждого клиента, а клиент, при обращении к серверу, пересыпает эти данные обратно. Так реализуется возможность «помнить» о предыдущих сессиях, даже если будут изменения в IP-адресах. Следующий рисунок иллюстрирует, что даже при изменении порядка, в котором приходят запросы от различных пользователей,

их можно идентифицировать по дополнительным данным, то есть сопоставить начало и конец стрелки.

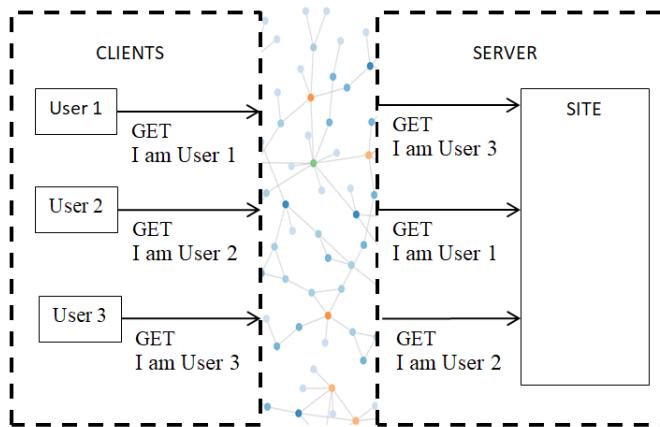


Рисунок 42

Эти дополнительные данные получили название Cookie (от англ. «выпечка, печенье», произносится как «куки» и не переводится). Можно сравнить Cookie с теми пирожками, которые несла в корзинке (HTTP-пакете) Красная Шапочка от Матери (сервера) к Бабушке (клиенту).

По формализму, Cookie похожи на POST-данные, включаемые формой с соответствующим методом отправки. Они не видны в адресной строке браузера, для того чтобы их просмотреть нужно проанализировать сам HTTP пакет.

Cookie автоматически обрабатываются браузером — сохраняются на компьютере пользователя при загрузке страницы и добавляются в пакет, когда формируется новый запрос к серверу. Участие пользователя в этом процессе не требуется. На внешний вид страницы Cookie, сами по себе, никак не влияют.

Установить, добавить или удалить Cookie может как сервер, так и клиент. Вне зависимости от того, кто первый установил Cookie, они будут автоматически включаться во все пакеты, которыми обмениваются клиент и сервер при дальнейших коммуникациях.

Создание, использование и удаление Cookie

Перед тем, как приступить к детальному рассмотрению способов использования Cookie, отметим некоторые их особенности, отличающие Cookie от других объектов JavaScript, и отражающиеся на их описании. Начнем с того, что Cookie передается в составе HTTP пакета, то есть является текстовым элементом, как и все остальное в этом пакете. Это означает, что описание Cookie является строкой, создание и удаление Cookie также логично организовывать в строковом виде.

Для одного домена (сайта) может быть установлено несколько Cookie. Соответственно, должен существовать способ отличать Cookie между собой. Для того чтобы была возможность разделять Cookie, им были присвоены имена. Подобно переменным в программировании, Cookie устанавливаются инструкцией «имя=значение». Такой же инструкцией можно поменять значение уже установленных Cookie, использовав их имена.

Следующей особенностью Cookie, на этот раз в отличие от переменных, является время их действия ([expires](#)). Например, в задачах авторизации, в которых Cookie находят широкое применение, часто бывает необходимо ограничивать продолжительность сеанса, требуя повторного ввода пароля. Для простых систем, наподобие форума или электронной почты, этот срок может быть около дня или больше. В то же время для банковских операций срок

ограничен часами или даже минутами. По прошествии срока действия, указанного в атрибуте «`expires`», Cookie автоматически удаляется.

В больших сайтах для лучшей их организации обычно создают несколько поддоменов, предназначенных для различных задач. Например, для некоторого сайта (домена) «`mystep.org`» задачи авторизации, регистрации и восстановления доступа можно реализовать на поддомене «`reg.mystep.org`». Для просмотра перечня предоставляемых услуг можно создать поддомен «`service.mystep.org`». Это значительно упрощает разработку сайта и делает возможным работу нескольких разработчиков одновременно над разными задачами. При этом может возникнуть ситуация, когда Cookie устанавливается одним поддоменом (например, `reg.mystep.org`), а доступ к ней требуется из другого поддомена (`service.mystep.org`). А возможно, наоборот, доступ к Cookie из других поддоменов нужно ограничить. Это регулируется свойством «`path`», устанавливаемым для каждой Cookie отдельно. Наиболее общим является выражение «`path=/`», что означает доступ из всех страниц сайта.

С точки зрения модели DOM, все Cookie хранятся в одной переменной «`document.cookie`». Обратите внимание, эта переменная является строкой, а не массивом или объектом, хотя в ней может храниться несколько значений различных Cookie. Однако работа с этой строкой имеет ряд особенностей, отличающих ее от других строк.

В переменную «`document.cookie`» автоматически попадают все Cookie, переданные сервером в составе HTTP пакета при загрузке страницы. При этом эта переменная доступна для изменений и со стороны клиента, то

есть добавить или удалить Cookie можно при помощи JavaScript. Добавленные клиентом Cookie будут точно также пересыпаться между клиентом и сервером при каждом последующем обмене пакетами, пока не истекут их сроки действия.

Для добавления (создания) Cookie нужно присвоить переменной «`document.cookie`» новое значение. Особенностью является то, что вместо оператора «`+=`», применяемого для добавления строк, нужно использовать простое присваивание «`=`». Браузер сам обработает эту команду как создание Cookie, предыдущие значения не «сотрутся»:

```
document.cookie = "registered=User; expires=Thu,  
14 Feb 2019 08:12:40 GMT; path=/"
```

В строке добавления указывается имя Cookie (в приведенном примере это «`registered`»), затем знак «`=`» и ее значение («`User`»), дополнительные кавычки для значения не нужны. Через разделитель «`;`» указывается ключевое слово «`expires=`» и записывается срок действия Cookie по стандарту RFC 2822, например, «`Thu, 14 Feb 2019 08:12:40 GMT`». Затем снова разделитель «`;`», ключевое слово «`path=`» и область действия Cookie. В данном примере — глобальная область «`/`».

Значение Cookie («`User`») может быть выбрано произвольным образом. Имя («`registered`») ограничено правилами именования (см. урок 1), но тоже предусматривает возможность изменения. Ключевые слова «`expires=`» и «`path=`» должны быть набраны именно в таком виде, вариации не разрешаются. Формат даты-времени также

должен соответствовать указанному формату, другие форматы, даже стандартные, не допускаются. Например, записи «2019-02-14 08:12:40» или «02/14/2019 08:12:40», являющимися правильными по другим стандартам, в данном случае будут приводить к ошибкам.

Дополнительно в атрибутах Cookie может быть указан домен, для которого устанавливается данное значение «`domain=mystep.org`». По умолчанию значение домена устанавливается в имя сайта (сервера), отправившего данную страницу, и в переназначении не нуждается.

Атрибут «`secure`» используется без значения и указывает, что данные Cookie могут передаваться только через защищенный протокол HTTPS. При использовании обычного протокола HTTP передача происходит не будет.

Атрибуты «`domain`» и «`secure`» не являются обязательными, можно создавать Cookie без них, применяя инструкцию, приведенную выше. С указанием всех перечисленных атрибутов команда добавления Cookie будет выглядеть следующим образом:

```
document.cookie = "registered=User; expires=Thu,  
14 Feb 2019 08:12:40 GMT; path=/;  
domain=mystep.org; secure"
```

Если нам нужно получить полный перечень установленных Cookie, мы обращаемся к переменной «`document.cookie`» для чтения. В ответ мы получим строку, в которой действующие Cookie представлены парами «имя=значение», разделенными символами «`;`» (точка с запятой и пробел). Данные о сроке и области видимости не возвращаются.

Еще одной особенностью Cookie является то, что они устанавливаются для сайтов (доменов) и, в большинстве случаев, не могут быть установлены для локальных файлов, с которыми мы раньше выполняли все практические работы.

Для того чтобы выполнить упражнения по работе с Cookie необходимо либо создать настоящий домен (локальный или сетевой), либо воспользоваться онлайн HTML — редакторами, например, «[Tryit Editor](#)» от «[w3schools.com](#)», который доступен по [ссылке](#). Подобных редакторов много и можно выбрать произвольный из них, но дальнейшее описание урока ориентировано именно на этот редактор.

Перейдите по указанной ссылке и введите или скопируйте в левую часть (в редактор) следующий код, поменяйте даты для срока действия Cookie на актуальные:

```
<!DOCTYPE html>
<html>
<body>
    <p id="out"></p>
    <script>
        document.cookie = "registered=User;
                           expires=Thu, 14 Feb 2019
                           08:12:40 GMT; path=/";
        document.cookie = "organization=Step;
                           expires=Thu, 14 Feb 2019
                           08:12:40 GMT; path=/";
        window.out.innerHTML = document.cookie;
    </script>
</body>
</html>
```

Инструкция `<p id="out"></p>` создает пустой абзац (параграф), предназначенный для дальнейшего заполнения из скриптовой части (для вывода информации).

В скрипте устанавливаются два значения для двух новых Cookie: «`registered=User`», и «`organization=Step`», указываем для них одинаковый срок и область действия.

Затем выводим в подготовленный абзац `«out»` значение строки `«document.cookie»`.

Нажмите на зеленую кнопку `«Run»`. В правой части редактора должен получиться результат, подобный приведенному на рисунке:

```

<!DOCTYPE html>
<html>
<body>
<p id="out"></p>
<script>
document.cookie = "registered=User; expires=Thu, 14 Feb 2019 08:12:40 GMT; path=/";
document.cookie = "organization=Step; expires=Thu, 14 Feb 2019 08:12:40 GMT; path=/"
window.out.innerHTML = document.cookie;
</script>
</body>
</html>

```

Result Size: 457 x 718

```

ASPSESSIONIDSQTTDACQ=MHHLKDJMFBCAEPKHOOCOC
_ga=GA1.2.1386818016.1546604075;
_gads=ID=e28eb8c0ad48363a:T=1546604074:S=ALNI_MaSFTL-
EP177UIpEVaUc2j9l3a1A;_gid=GA1.2.581304514.1549626874;
G_ENABLED_IDPS=google; snhfrFromEEA=false;
registered=User; organization=Step

```

Рисунок 43

Как видно, в конце выведенной строки действительно присутствуют установленные нами Cookie (возможно, они появятся в других местах, не обязательно в конце). При этом также видны другие Cookie, установленные без нашего участия.

Эти значения получены от сервера и используются для служебных целей, например, для поддержания работы редактора с разными пользователями, сохраняя введенные ранее коды.

Обрабатывать такую строку достаточно неудобно, поэтому напишем команды, которые преобразуют ее в объект:

```
var x = document.cookie;
var s = x.split('; ');
var cookieObject = {};
var c;
for(var i=0; i < s.length; i++){
    c = s[i].split('=');
    cookieObject[c[0]] = c[1];
}
```

В первой строке получаем данные о Cookie и сохраняем их в переменной «**x**». Далее разбиваем строку методом «**split**» по разделителю «; » (не забываем пробел). В результате получаем массив «**s**» из строк в виде «имя=значение».

Проходим циклом по полученному массиву и разбиваем каждую его запись по разделительному символу «=», сохраняя результат в переменной «**c**» (**c** = **s[i].split('=')**). Полученные данные об имени и значении Cookie сохраняем в объекте «**cookieObject**» как пару «ключ-значение». Ключом выступает имя Cookie, сохраненное в первой части разделенной строки «**c[0]**», значением — вторая часть «**c[1]**» (**cookieObject[c[0]] = c[1]**).

Итоговый объект «**cookieObject**» облегчит нам работу с Cookie. Например, выведем полученные данные отдельно, по одной записи в строке. Для этого используем цикл «**for-in**», предназначенный для обхода объектов

```
for(var co in cookieObject)
    window.out.innerHTML += co + " = " +
    cookieObject[co] + "<br/>";
```

В теле цикла добавляем к содержимому абзаца «`out`» данные об имени Cookie «`co`», знак «`=`», значение Cookie «`cookieObject[co]`» и разрыв строки «`"
"`». Со строкой «`<innerHTML>`» работа ведется обычным образом, используя для добавления оператор `+ =`.

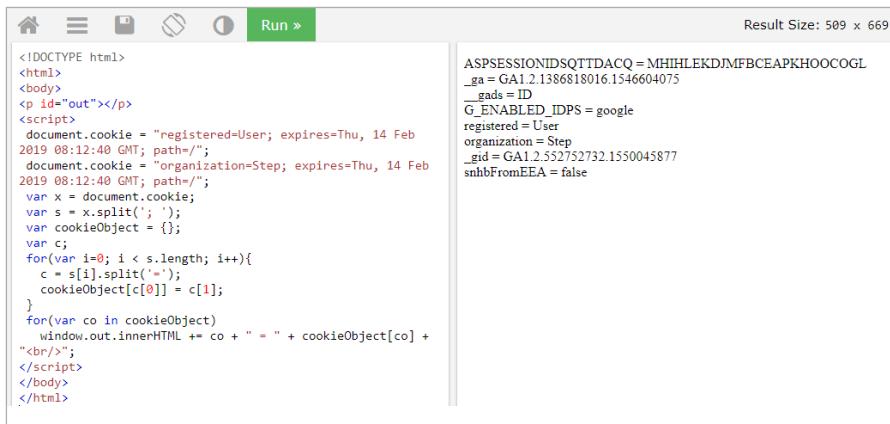
С учетом описанного преобразования, код примет следующий вид.

```
<!DOCTYPE html>
<html>

<body>
    <p id="out"></p>
    <script>
        document.cookie = "registered=User;
                            expires=Thu, 14 Feb 2019
                            08:12:40 GMT; path=/";
        document.cookie = "organization=Step;
                            expires=Thu, 14 Feb 2019
                            08:12:40 GMT; path=/";
        var x = document.cookie;
        var s = x.split('; ');
        var cookieObject = {};
        var c;
        for(var i=0; i < s.length; i++){
            c = s[i].split('=');
            cookieObject[c[0]] = c[1];
        }
        for(var co in cookieObject)
            window.out.innerHTML += co + " = " +
            cookieObject[co] + "<br/>";
    </script>
</body>

</html>
```

Обновите его в редакторе и снова нажмите кнопку «Run». В результате каждое отдельное определение Cookie должно начинаться с новой строки



```

<!DOCTYPE html>
<html>
<body>
<p id="out"></p>
<script>
document.cookie = "registered=User; expires=Thu, 14 Feb
2019 08:12:40 GMT; path=/";
document.cookie = "organization=Step; expires=Thu, 14 Feb
2019 08:12:40 GMT; path=/";
var x = document.cookie;
var s = x.split(' ');
var cookieObject = {};
var c;
for(var i=0; i < s.length; i++){
  c = s[i].split('=');
  cookieObject[c[0]] = c[1];
}
for(var co in cookieObject)
window.out.innerHTML += co + " = " + cookieObject[co] +
"<br/>";
</script>
</body>
</html>

```

ASPSESSIONIDSQTTDACQ = MHIHLEKDJMFBCAEPKHOOCOGL
`_ga = GA1.2.1386818016.1546604075`
`_gads = ID`
`G_ENABLED_IDPS = google`
`registered = User`
`organization = Step`
`_gid = GA1.2.552752732.1550045877`
`snhbFromEEA = false`

Рисунок 44

Как видно на приведенном рисунке, порядок Cookie может измениться — установленные нами значения теперь отображаются не в конце списка.

Давайте создадим простейшую систему, имитирующую авторизацию пользователей. Для этого нам понадобится

1. Перехватить событие загрузки страницы «**onload**», установив для него собственный обработчик. Это позволит нам по-разному отображать страницу для зарегистрированных и незарегистрированных пользователей.
2. В обработчике события нужно проверить установлено ли значение для Cookie с именем «**register**» (имя может быть произвольным, мы его сами выбираем. Не используйте использованное ранее в примерах имя

«**registered**», т.к. его значение уже установлено в предыдущих упражнениях).

3. Если указанное значение присутствует, то выдаем приветствие пользователю. Его имя хранится как значение Cookie.
4. Если значения нет, выводим предложение авторизоваться.

В левую часть окна (в редактор) напишите или скопируйте следующий код, далее проведем его анализ.

```
<!DOCTYPE html>
<html>

<body onload="checkReg()">
    <div id='regDiv'></div>

    <script>
        function checkReg() {
            var x = document.cookie;
            var rd = document.getElementById("regDiv");
            var s = x.split('; ');
            var cookieObject = {};
            var c;

            for(var i=0; i < s.length; i++) {
                c = s[i].split('=');
                cookieObject[c[0]] = c[1];
            }

            if('register' in cookieObject){
                rd.innerHTML = "Hello, " +
                    cookieObject['register'];
            }
            else
            {
```

```

        rd.innerHTML =
            "Name: <input type='text' id='name' />
             <input type='button' value='Register'
            onclick='regClick()' />";
    }
}

function regClick() {
    var inputName = document.
        getElementById("name");
    var name = inputName.value;
    var expDate = new Date;
    expDate.setTime((new Date).
        getTime() + 60*1000);
    document.cookie =
        "register="+name+";expires="+expDate.
        toGMTString()+";path=/";
    checkReg();
}
</script>

</html>

```

При объявлении тела документа `<body onload = "checkReg()">` указываем, что к событию загрузки документа `«onload»` подключена функция `«checkReg»`.

Далее создаем блок `<div id='regDiv'></div>`, который будет служить нам для формирования содержания страницы (взамен используемому ранее абзацу `«out»`).

В скриптовой части создаем функцию `«checkReg»`. Ее начало мы уже разбирали, происходит преобразование строки Cookie в объект. Далее совершается проверка, установлено ли значение для имени `«register»` инструкцией `if('register' in cookieObject)`.

В случае успешного результата проверки выводим в блок «`regDiv`» приветствие, иначе формируем поле ввода и кнопку авторизации, для которой указываем новую функцию «`regClick`».

Далее описываем эту функцию. В ее теле получаем введенное в поле «`name`» имя. Далее формируем срок действия Cookie: создаем объект для работы с датой и временем «`var expDate = new Date`». Получаем текущую дату и время инструкцией «`((new Date).getTime()`», добавляем к ней значение «`60*1000`», что соответствует 60 000 миллисекунд, то есть 60 секунд, то есть 1 минута. Полученное значение сохраняем в объекте «`expDate`». Все это записано одной строкой

```
expDate.setTime((new Date).getTime() + 60*1000);
```

Затем устанавливаем новое значение для Cookie по описанному ранее в уроке формализму. Для указания срока действия используем метод «`expDate.toGMTString()`» у сформированного объекта.

В конце функции вызываем первую функцию «`checkReg`» для перехода в авторизованный режим. Более правильный вариант — инициировать перезагрузку страницы, но в данном HTML редакторе это может не сработать, в силу внутренних ограничений. Если Вы работаете с собственным сайтом лучше сделайте перезагрузку страницы, вызвав метод «`location.reload()`» вместо вызова функции «`checkReg`».

Запустите скрипты, нажав кнопку «`Run`». Поскольку это первый запуск, мы должны увидеть предложение авторизоваться (рис. 45).

The screenshot shows a browser developer tools interface. On the left, there is a code editor window containing the provided JavaScript code. On the right, there is a results panel with the title "Result Size: 338 x 712". Inside the results panel, there is an input field labeled "Name: User" and a button labeled "Register".

```
<!DOCTYPE html>
<html>
<body onload="checkReg()">
<div id="regDiv"></div>

<script>
function checkReg(){
    var x = document.cookie;
    var rd = document.getElementById("regDiv");
    var s = x.split('; ');
    var cookieObject = {};
    var c;
    for(var i=0; i < s.length; i++){
        c = s[i].split('=');
        cookieObject[c[0]] = c[1];
    }
    if('register' in cookieObject){
        rd.innerHTML = "Hello, "+cookieObject['register'];
    }else{
        rd.innerHTML = "Name: <input type='text' id='name' /><input type='button' value='Register' onclick='regClick()' />";
    }
}
function regClick(){
    var inputName = document.getElementById("name");
    var name = inputName.value;
    var expDate = new Date();
    expDate.setTime((new Date).getTime() + 60*1000);
    document.cookie = "register="+name+";expires="+expDate.toGMTString()+";path=/";
    checkReg();
}
</script>
</html>
```

Рисунок 45

Ведите имя в поле ввода и нажмите кнопку «Register». Убедитесь, что результат сменился на приветствие.

The screenshot shows a browser developer tools interface. On the left, there is a code editor window containing the provided JavaScript code. On the right, there is a results panel with the title "Result Size: 338 x 712". Inside the results panel, the text "Hello, User" is displayed.

```
<!DOCTYPE html>
<html>
<body onload="checkReg()">
<div id="regDiv"></div>
```

Рисунок 46

Перезапустите скрипт, нажав кнопку «Run» еще раз — приветствие не меняется, поскольку значение Cookie «register» все еще установлено.

Подождите минуту, в течение которой должен истечь срок действия Cookie, и снова инициируйте перезапуск. Приветствие сменится на форму регистрации, что означает исчезновение имени «register» из `document.cookie`.

Исчезновение Cookie после истечения срока действия также используется для удаления ранее установленных

Cookie. Отдельных команд удаления Cookie нет, а если удалить их все же необходимо, то нужно установить им «просроченный» период действия. По сути, можно выполнить те же действия, что и в функции `regClick()`, только вместо добавления 1 минуты, отнять ее. Такая «установка» Cookie на самом деле приведет к удалению.

Преимущества и недостатки cookie

Как мы уже отметили, Cookie нужны для разделения клиентов, а значит для хранения данных, которые можно назвать персональными, уникальными для разных пользователей. Применение механизмов Cookie можно сравнить с использованием читательского билета в библиотеке или удостоверения личности в банке. Если посетитель библиотеки придет без читательского или клиент банка придет без документов, то его, скорее всего, не обслужат, по крайней мере, должным образом. В то же время потеря удостоверения или его кража могут привести к негативным последствиям, т.к. им могут воспользоваться злоумышленники.

Полностью аналогично, основные преимущества и недостатки Cookie так или иначе связаны с вопросами использования и конфиденциальности персональных данных. Удобства подобны удостоверению личности — Вас можно идентифицировать и оказать полный перечень услуг. Опасность заключается в том, что это удостоверение может быть использовано в нелегальных целях.

Одной из наиболее популярных областей применение Cookie, как уже отмечалось выше, является авторизация. Когда пользователь вводит логин и пароль, сервер их анализирует и, в случае достоверности, устанавливает Cookie для этого клиента. В дальнейших сессиях сервер проверяет, есть ли данного клиента установленные Cookie. Если есть, то повторная авторизация не нужна. Добавив

тот факт, что Cookie обрабатывается и пересылается браузером без участия пользователя, получаем удобный способ поддерживать авторизированное соединение.

Главное правило информационной безопасности гласит: «чем удобнее, тем опаснее». Сохранение надолго данных авторизации может привести к тому, что другой пользователь того же компьютера, включив его после Вашей работы, получит доступ к Вашим данным. Так же потенциально опасно то, что другой пользователь может найти сохраненные после Вашей работы Cookie и воспользоваться ими для авторизации. И уж тем более опасно вводить данные авторизации, пользуясь чужим компьютером или смартфоном.

Другое популярное применение Cookie заключается в персонализированной (таргетинговой) рекламе. Когда пользователь посещает определенные страницы сайта, просматривает конкретные новости или товары в сетевом магазине, для этого сайта могут быть установлены Cookie, запоминающие историю просмотров. Эта информация может быть использована для подготовки рекламных объявлений персонально для Вас.

С одной стороны, это делает рекламу более эффективной и менее раздражительной. С другой стороны, анализ Вашей истории просмотров может быть использован и для сбора нежелательной информации — времени начала работы, продолжительности работы, посещенных страницах и т.п. Да и постоянно смотреть рекламу товара, который Вы уже купили, может быть неприятно.

Для того чтобы информация оставалась доступной после перезапуска браузера, Cookie сохраняются в файлах

на жестком диске компьютера. Соответственно, свободное дисковое пространство будет уменьшаться при использовании Cookie, что часто относится к их недостаткам. В то же время, в браузерах устанавливаются ограничения:

- всего может храниться до 300 значений Cookie;
- каждый Cookie не может превышать 4Кб;
- с одного домена может храниться до 20 значений Cookie.

Если установка новых Cookie приведет к превышению ограничений, то браузер удалит наиболее старую (по времени установки) Cookie либо «обрежет» размер до лимитированных 4Кб. Умножив 4 Кб на 300 Cookie, получим максимальный размер хранимых данных в объеме 1200 Кб или чуть больше 1 Мб. Даже для смартфона этот объем сложно назвать значительным. Особенно по сравнению с удобством восстановления данных при перезапуске браузера.

В старых браузерах файл с Cookie хранился в папке с самим браузером. В современных системах, предназначенных для работы разных пользователей, файлы хранятся в персональных папках. Браузеры имеют множество настроек, позволяющих управлять файлами Cookie как всеми сразу, так и для отдельных сайтов. Лучше использовать инструментарий, предоставляемый браузерами, чем искать сами файлы Cookie и управлять ими вручную. Альтернативные возможности имеют антивирусные программы и мониторы.

В итоге, Cookie имеют как преимущества, так и недостатки. Как и любое другое средство, Cookie могут быть использованы как для упрощения работы и повышения ее качества, так и для злоумышленных действий. Полное отключение Cookie приведет к необходимости повторной

авторизации при каждом обновлении страницы, переходе по ссылке или перезапуске браузера. Неосторожное использование Cookie, в свою очередь, может привести к утечке персональных данных. Порекомендуем не отказываться от применения Cookie, но стараться не забывать очищать их, когда пользуетесь не личным устройством.

Осознавая потенциальную опасность, которую несут в себе Cookie и, в то же время, удобство их использования Европейский Союз принял директиву «[DIRECTIVE 2009/136/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL](#)», регулирующую правила электронных коммуникаций и предоставления услуг. В пункте 66 этой директивы указано, что сайты обязаны предоставить пользователям четкую и исчерпывающую информацию о любой деятельности, которая может привести к сохранению данных или получению к ним доступа (в т.ч. Cookie). Сайт обязан предоставить право на отказ от сохранения данных максимально удобным для пользователя способом.

Директива была принята в 2011 году, получив название «Закон о Cookie» (*Cookie law*). Один год был установлен как переходной период. С 2012 года сайты, использующие Cookie, обязаны сообщать об этом пользователю. В противном случае при работе в Европейском Союзе организация, нарушающая этот закон, будет оштрафована. Поэтому большинство популярных сайтов обязательно предупреждают пользователя о Cookie и дают возможность отказаться от них.

Для примера можно зайти на сайт объявлений [eBay](#). В нижней части страницы после загрузки появится предупреждение об использовании Cookie

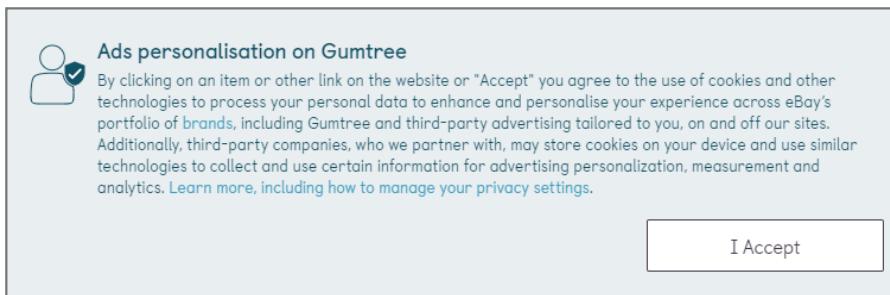


Рисунок 47

В предупреждении содержится короткая информация о Cookie, а также ссылка страницу с более детальным их описанием и настройками использования.

Современные браузеры позволяют быстро проверить наличие Cookie для сайта, выводя кнопку в левой части адресной строки. Например, для браузера Chrome она выглядит следующим образом

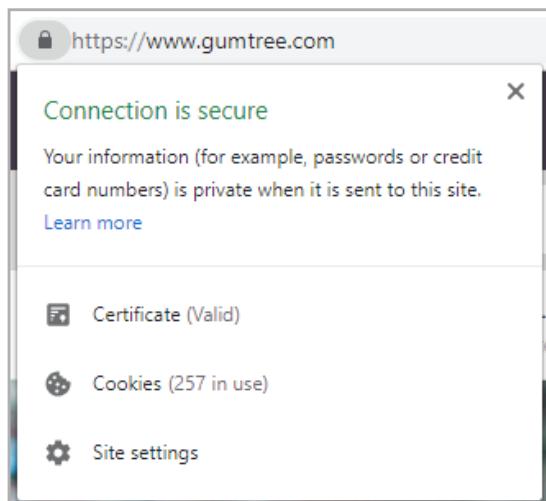


Рисунок 48

Перейдя в настройки сайта ([Site settings](#)) можно задать ограничения на использование Cookie, JavaScript и другие параметры.

Естественно, что сайты, честно предупреждающие об использовании Cookie, на самом деле не собираются использовать их в злоумышленных целях. Для них важнее позитивный имидж и отказ пользователя от Cookie лишь усложнит ему работу с сайтом. Настораживать должна ситуация, когда сайт использует Cookie, но не предупреждает об этом. А как проверить наличие у сайта Cookie и определить их значения Вы уже знаете из этого урока.

Домашнее задание

Задание 1

1. Дополните созданную форму телефонного справочника возможностью:
 - ввода даты рождения (`input type="date"`);
 - выбора файла с фотографией (`input type="file"`);
 - ввода электронной почты (`input type="email"`);
 - указания сайта (`input type="url"`);
 - ввода «секретного слова» (`input type="password"`);
 - выбора любимого цвета (`input type="color"`);
 - отметки **Напоминать о дне рождения** (`input type="checkbox"`);
 - сброса введенных данных (`input type="reset"`).
2. Обратите внимание, в каком формате передаются данные от каждого из элементов формы (в адресной строке браузера).
3. Примените стилевые определения для элементов формы на свой вкус.

Вопросы к домашнему заданию

1. Что такое HTML-формы? Для каких целей они нужны?
2. Как получить программный доступ к форме? Ко всем формам?
3. Что такое элементы формы? Каким образом можно получить к ним доступ?

4. Какими способами можно указать, что элемент относится к форме?
5. При каких условиях данные из элемента передаются с отправкой формы?

Задание 2

1. Реализуйте возможность добавлять к контакту несколько дат с выбором типа: день рождения, день профессии, день первой встречи (можете дополнить или поменять перечень). Используйте для выбора типа даты элемент `<select>`.
2. Реализуйте возможность загружать несколько файлов с подписями каждого из них. Содержание подписи не ограничивается (`<textarea>`).
3. Реализуйте возможность удалять добавленные элементы (пп.1-2) каждый по отдельности.
4. Проверьте функциональность формы, убедитесь, что все данные, в т. ч. от добавленных элементов, передаются при отправке формы (попадают в адресную строку браузера).

Вопросы к домашнему заданию

1. Раскройте последовательность действий для добавления нового элемента в форму?
2. Чем отличаются методы `appendChild` и `insertBefore`?
3. Для чего нужен механизм клонирования элементов? Какой порядок создания клона?

4. Как задать или поменять имя нового созданного элемента? Как установить ему обработчик события?
5. Как исключить (удалить) элемент из формы?

Задание 3

1. Напишите регулярные выражения для проверки того, что переменная «`code`» содержит:
 - a) ровно 4 цифры;
 - b) от 4 до 6 цифр;
 - c) от 4 до 6 символов, которыми могут быть цифры либо маленькие латинские буквы;
 - d) ровно 7 шестнадцатеричных цифр (`0123456789 ABCDEF`), допускается использование как больших, так и маленьких букв (`0123456789abcdef`);
 - e) начинается с цифры и содержит произвольное количество цифр и латинских букв любого регистра (большие и маленькие).
2. Напишите инструкцию для создания Cookie с именем «`status`» и значением «`active`», действие которой будет ограничено 15 минутами от момента создания.
3. Напишите инструкцию для удаления Cookie с именем «`status`».
4. Напишите скрипт для подсчета количества установленных Cookie для данного сайта.
5. Дополните созданную в разделе 6 форму авторизации полями, рассмотренными в разделе 4 данного урока, обеспечьте проверку их достоверности. При первом запуске форма должна выглядеть следующим образом (рис. 49).

Run »

First name:

Last name:

Login:

Password:

Confirm password:

Email:

Phone number:

Code:

I accept the rules

Register

Рисунок 49

Используйте механизм Cookie для запоминания введенных пользователем данных (кроме пароля и кода), отобразите их на странице приветствия. Необходимые поля должны проверяться на пустоту браузером, а также анализироваться средствами JavaScript

Run »

First name:

Last name:

Login:

Password:

Confirm password:

Email:

Phone number:

Code:

I accept the rules

Register

Рисунок 50

Run »

First name:

Last name: ! Please fill out this field.

Login:

Password:

Confirm password:

Email:

Phone number:

Code:

I accept the rules

Register

Рисунок 51

An embedded page on this page says

Password has no big letters

OK

First name:

Last name:

Login:

Password:

Confirm password:

Email:

Phone number:

Code:

I accept the rules

Register

Рисунок 52

После того, как форма будет заполнена данными и нажата кнопка «**Register**» на странице должны

появиться введенные данные, примерный образец приведен на рисунке



Рисунок 53

Убедитесь, что после обновления страницы (повторного нажатия на кнопку «Run») приветствие сохраняется, а после истечения срока действия Cookie снова появляется форма регистрации.

6. Дополните страницу приветствия кнопкой «Sign out», удаляющей все Cookie, созданные при регистрации. При нажатии на эту кнопку сразу должна отобразиться регистрационная форма, без необходимости ожидания времени истечения срока Cookie.

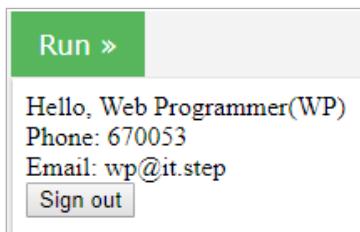


Рисунок 54



Unit 4.

Формы, проверка достоверности форм. Использование Cookie

© Компьютерная Академия «Шаг», www.itstep.org.

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.