STEP IT ACADEMY

Developing client-side scripts using Javascript

Lesson №5

Forms

Contents

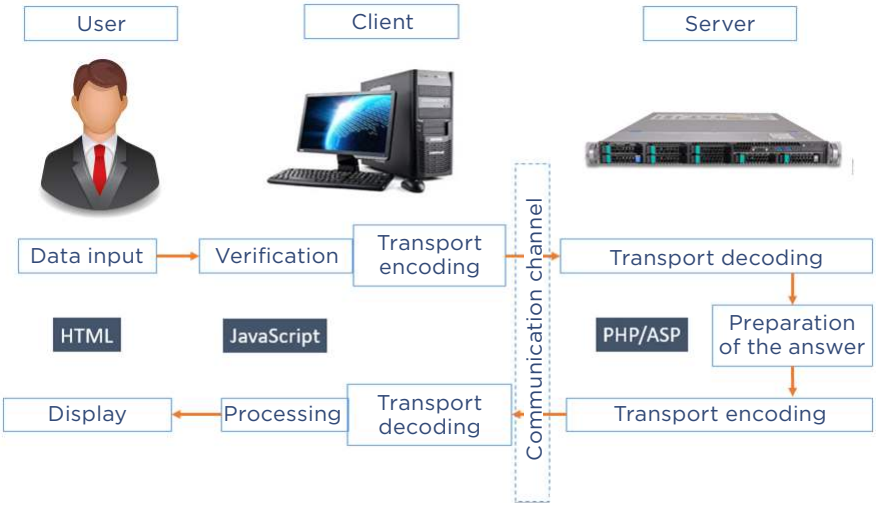
Lesson materials are attached to this PDF file.

In order to get access to the materials, open the lesson in Adobe Acrobat Reader.

Creating Form Elements

Diagram of Internet resource

In order to understand the role and place of forms, features of their attributes and elements, let's look at the general diagram of Internet resource arrangement.



Usually, the resource arrangement is divided into three levels: **user**, **client**, and **server**.

User level is visual interface of the resource. It is called the site as a rule. This is what we see in the browser after successful load of a resource. HTML is used to create the interface.

Client level is software that creates and draws interface. It also accepts input data from the user, reacts to clicking buttons and links, selecting lists, etc., and sends them to the server.

The client level is provided by the browser and JavaScript is used for additional control of its operation.

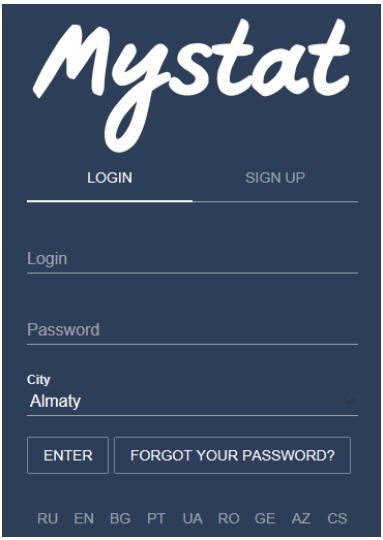
Server level is the heart of the resource. Here, the main data are stored, queries are processed, authorization is checked, and answers are prepared for various client requests. The server hosts a resource database that stores information about the main items of the resource (products, news, files), users, their browsing history and messages, and much more. Server work is provided by so-called server technologies: PHP, ASP.NET, and the like.

Communication channel plays a special role in the exchange as a mean of transferring data from client to server and back. Typically, this is summarized by the word "network": cables, switches, Wi-Fi modules, etc. However, communication channel imposes certain restrictions on communication between the client and the server. For example, a channel does not allow transfer of arbitrary characters but requires their transformation into a special form – transport encoding. It is not difficult to see the action of transport encoding; enter "*step academy*" into Google search and copy the address line of the browser. You will get something like this:
https://www.google.com.ua/search?q=step+academy-&rlz=1C1SQJL_ruUA786UA786&oq=step+academy&aqs=-chrome..69i57j0l5.4256j0j4&sourceid=chrome&ie=UTF-8

This is the transport encoding required for the communication channel. Encoding is usually done by the browser itself and we do not need additional actions for encoding. Nevertheless, we should remember this when we build links into our site. It is desirable to create them immediately in the encoded form to prevent possible errors in data transmission.

What are forms

As we could see in the previous section, the process of exchanging data in a network resource is rather complicated and contains several stages. To ensure that the developer of the site does not go into each stage with its features, a special HTML object – **form** was created. This is a mechanism for automated sending of data entered by the user to the server side of the site. Since the word "form" has many meanings, in this case it is close to a feedback form. One of the most common applications of the form, which is on almost every site, is authorization, the form of logging on to the site.



A group of interface elements can be bound to a form using HTML tools and sent to the server being processed and decoded automatically without the need for the developer to interfere in these processes.

A form is created with the `<form>` tag, the closing tag `</form>` is required. Elements described between the opening and closing tag of the form are automatically bound to this form. By default, the form tag does not introduce style features, that is, grouping elements into a form does not change the way they are displayed. But, like any other object, the form can be additionally stylized.

Attributes of the form

In accordance with its purpose, the form must ensure that data are sent to the server. We have already noted above that server is a large software complex, and sending data simply "to the server" is like simply "sending a letter" to the capital. Without the addressee's letter, the letter will reach the capital but it will be lost somewhere inside the belly of the post office. Similarly, you must specify the "destination" for requests to the server, which means name of the program on the server that will receive data sent by the form. Destination name is specified by the `action` form attribute: `<formaction="add_user.php">`.

The second important form attribute is a method of sending a message to the server. Here again we need to remember that the server is a complex, and it can handle requests of various types: requests for reading, writing, modifying, deleting. For example, the `HEAD` method asks the server for only the header portion of the site without its body to quickly get

meta data like title (`title` is indicated in the header section). The `DELETE` method is a command to delete some data from the resource. The entire list of possible query methods for the current HTTP/1.1 protocol is given and described in detail in the special RFC 2616 standard.

Because forms are a simplified way to automate delivery of data to the server, only two request methods are available for them: `GET` and `POST`. To implement other methods, you need to create your own functions to send messages to servers instead of standard forms. Similar techniques will be considered further in the AJAX technology.

With the method specified, the form declaration will look as follows: `<formaction="add_user.php"method="GET">`. You can often see in textbooks, guides, and site codes that method names are written in capital or small letters. From the point of view of HTML, the register of name of a role method does not matter (you can write it in both capital and small letters). However, RFC 2616 is case sensitive and requires only capital letters. Following these requirements, we will indicate method name in uppercase. Let's hope that when studying more complex material on server requests, this habit will prevent some errors.

The difference between `GET` and `POST` methods is as follows. The **`GET` method** includes data in the URI (in the address bar). The example of URI and `GET` parameters was given in the previous section where transport encoding was demonstrated. Data that are sent to the server are written right in the address bar of the browser for the site address https://www.google.com.ua/. In this case, the search query.

To separate addresses and data in `GET` requests, question mark `?` is used.

In general, if you need to pass two parameters – `A=1` and `B=2` – to example.itstep.org, then the `GET` request will look as follows:

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

The question mark, as it was already mentioned, separates site address and parameters, the `&` sign separates parameters among themselves if there are multiple.

To understand how the **`POST` method** works, it is necessary to consider the complete composition of messages exchanged between the client and the server. Without going into details, we can say that (besides the request itself) the message includes additional data: headers.

Let's consider them in the following example of a message sent by the browser to the server:

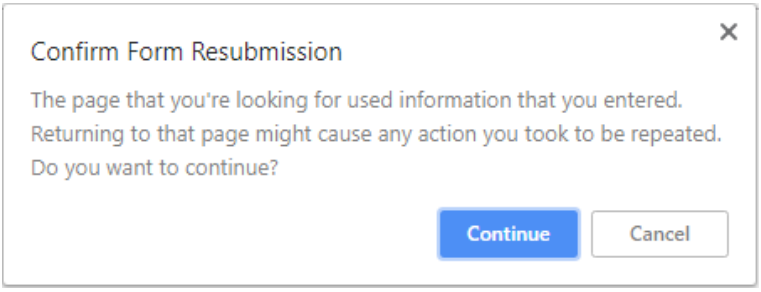
```
POST /add_user.php HTTP/1.1
Host: 123.45.67.89
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64;
rv:47.0) Gecko/20100101 Firefox/47.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
username=itstep&pass=MyItstep
```

The first line is the request itself. The command that we send to the server. The second one indicates IP address. In the third one, data about the browser type that prepared the request

is transmitted. The following is an indication of the content type of the request (`Content-Type` header). The value given corresponds to sending data from the form. Then, the length of the message (the `Content-Length` header) is indicated in characters followed by the message body: parameters that we pass from the form. The format of data transfer is the same as in the `GET` requests: `name = value`, division by `&`.

It should be noted that, first, headers in the request are actually much larger, only a simple example is given here and, second, `GET` requests are the same messages, but their first line contains the `GET` command and data are contained in it, not in the message body.

Due to the fact that the transmitted `POST` data are included in the message body, the address bar in the browser does not change. The transfer of parameters to the user is not visible. However, if the page is built using the `POST` data, that is, the page appeared after sending the form, then updating the page will require resending the same parameters. The user will receive a warning: The page that you're looking for used information that you entered. Returning to that page might cause any action you took to be repeated. Do you want to continue?



There are no such problems with GET because data are automatically transferred from the address bar itself.

A brief comparison of methods for form submission is presented in the form of a table:

Method	Advantages	Disadvantages	Field of use
GET	▪ Allows creating a URL parameters ▪ Eases transfer control	▪ It's easy to peep data* ▪ Large data overload address	Sending short data, autonomous links
POST	▪ Allows hiding transferred data ▪ Doesn't have limitation to data length	▪ It's hard to notice transfer errors ▪ Browser message at page update	Sending images, files, large data

* This disadvantage is often perceived as disastrous for the GET method. To be fair, it's not that difficult to see POST data by opening the full HTTP message.

Create a form

To perform practical exercises on working with forms and their elements, several features should be noted. First, the form should send data to the server that passed the page of the site, where the form is located, to the client. Since we have not studied the development of the server part yet, the exercises will be performed as separate html files located on a single computer, not divided into *client* and *server*.

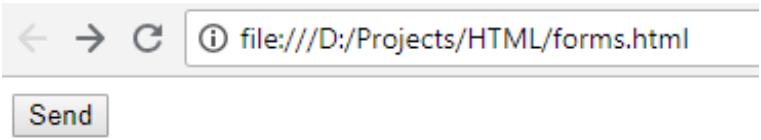
What then is the server and where will the form send data?

To answer these questions, let's create a file `forms.html` in our working folder, for example, `D:\Projects\HTML`. Folder selection is arbitrary, you can create a file in any folder on any disk. But in the following description, we will refer to this folder and this disk.

Let's create the simplest form in the file with the only element sending the form (we'll talk in more detail about elements in the next section).

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

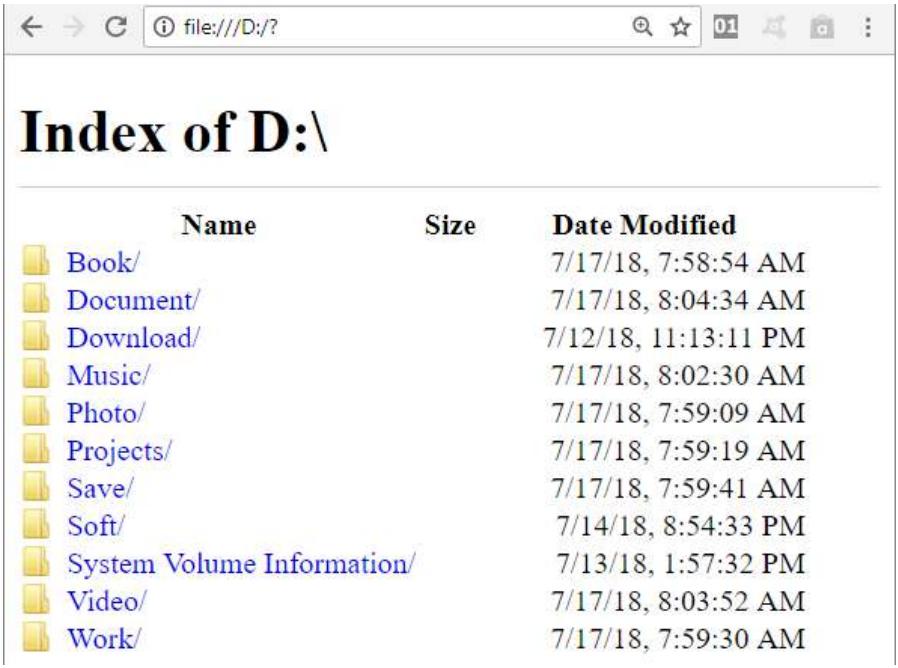
As you can see, the form refers to the "site root", that is, the `action='/'` attribute is set. Let's open the created file in the browser and we'll see the following:



Analyzing the address bar, we can draw conclusions:

- the file protocol (`file://`) was used for working with our page. Web sites use the `http://` and `https://` protocols;
- the site root is the disk `D:/`.

Let's make sure of this by clicking the submit form button. As a result, we get the following:



The appearance of the question mark at the end of the address bar indicates the correct operation of the form. Opening the contents of `D:/` disk confirms that it is the root element of our "site".

Let's make a conclusion: it is not necessary to specify slash as the destination of the form (`action='/'`). This expression can often be found in textbooks or reference books as the simplest example of creating a form. Make sure to remember that this applies to sites and means "site root" where the form with an example is usually located.

When working with files, the `action` attribute should be specified as empty (`action=''`). In this case, data will be transmitted to the same address on which the form was created. Alternatively, you can not specify this attribute at all when declaring a form. But, for educational purposes, it is better to not forget about it and indicate empty quotes.

Replace the `action` attribute of our form with empty quotes, save the file, refresh the page in the browser and click Send. Make sure that there is no redirection and we get to the same page. An indication of the form operation will be the appearance of the question mark at the end of the URI.

To send the form, always use the `GET` method mainly due to the ability to control the correctness of data transmission (more accurately, preparation for transmission) since we cannot verify their receipt from the server side (in view of its actual absence). At this stage, let's just make sure that these forms are correctly assembled, prepared, and included in the URI.

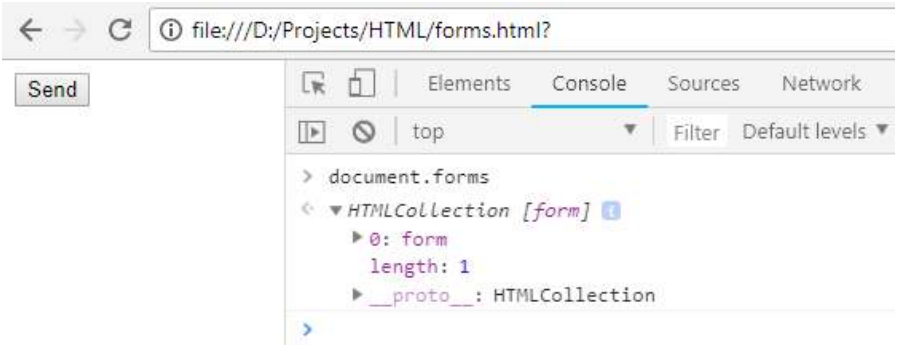
Summarizing, creation of a form when working with a separate file specifying the main attributes will look like:

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

There can be multiple forms on a single page. Each can have its own method and its addressee. To work with forms using JavaScript, the `document` object has a special collection (array), `document.forms`. When creating a new form, a corresponding element is added to this collection through which the form can be controlled programmatically. For example, if two forms are placed on a given HTML page, then in the `document.forms` collection, there will be two elements (`document.forms[0]`

and `document.forms[1]`), each answering for "their" form. The order of forms in a collection corresponds to the order of their declaration in the HTML code.

Open the developer tools (by pressing the F12 key) in the browser, select Console, enter the line *"document.forms"* in the console and press Enter. In the answer you receive, you can expand the details by clicking on the triangle.



Copy the created form (in the forms.html file) and duplicate it. Notice where the second Send button appears. Repeat the query in the console, make sure to expand the `document.forms` collection.

Elements of the form

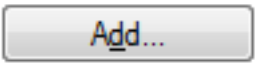
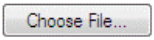
Elements of the form are objects of the common user interface, the values of which will be transferred to the server when the form is submitted. Usually, these are exactly the objects in which you can enter, select, or change their value. The exception, perhaps, is a special hidden element that is added to the form programmatically and the user cannot change it.

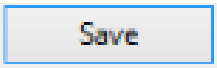
It should be noted that not everything that is contained in a form is considered to be its elements. These do not include "static" HTML objects: labels, drawings, blocks, line breaks, and other markup tools. However, all objects between the opening and closing tags of the form are child forms.

In order to distinguish between form elements (with variable values) and all child objects, two different collections are provided in the form object:

- the `elements` collection contains only the form elements,
- the `children` collection contains all child objects.

Most form elements are created by the `input` tag and differ in values of the `type` attribute. In addition, there are several isolated elements. The main ones are shown in the below table.

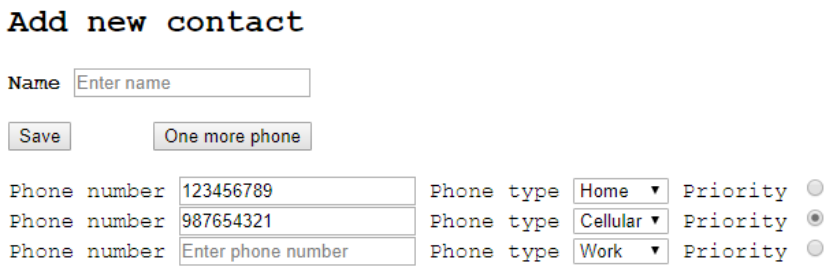
Tag	Description	How it looks like
`<input type= button>`	Button	
`<input type= 'text'>`	Data input by the user	
`<input type= 'checkbox'>`	Checkbox	
`<input type= 'radio'>`	Radio buttons: select one	
`<input type= 'file'>`	Dialogue of selecting a file	

Tag	Description	How it looks like
`<input type= ‘submit’>`	Submit form button	
`<input type= ‘password’>`	Password input field (characters are not displayed)	
`<input type= ‘hidden’>`	Hidden field	It is not displayed (but transferred)
`<input type= ‘date’>`	Select date	
`<input type= ‘number’>`	Number input field	
`<select>`	Drop-down list	
`<textarea>`	Text field	

The above list in the table is not a complete guide, it only displays the most popular elements. A complete list available in the current HTML standard can be found on the World WideWebConsortium (W3C) website at https://www.w3.org or on alternative reference or educational resources.

Elements are associated with the form and fall into its collection if they are created between the opening (`<form>`) and closing (`</form>`) form tags. If an element is created elsewhere, it can be correlated with the form using a special `form` attribute. In this case, an identifier (`id`) must be assigned to the form, and you should specify the specified attribute `<inputform="telForm"...>` for the element created outside the `<formid = "telForm">` tag to bind it to the form.

To illustrate the work with forms, let's create a page that allows adding a new entry (contact) to the phone book. It should contain tools for entering the contact name and phone numbers along with their types. Initially, it is suggested to enter one number, but by pressing the One more phone button, fields for entering a new number and its type must be added. It is also assumed that there is a possibility to choose a priority number. The approximate view of the page we want to get is in the following figure.



Here is the HTML code for the page, and we'll comment on it below. In the previously created forms.html file, replace the contents by copying or typing this code. File is also available in the Sources_5.1 folder, its archive file is attached to the PDF of this lesson.

```
<!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>
```

According to the set conditions, one string of data input is initially created in the form. Therefore, in the above code, only one group of elements is created for the number and its type. Addition of new numbers will be implemented as a reaction to clicking the button with the `add_click()` function call in the next sectionц .

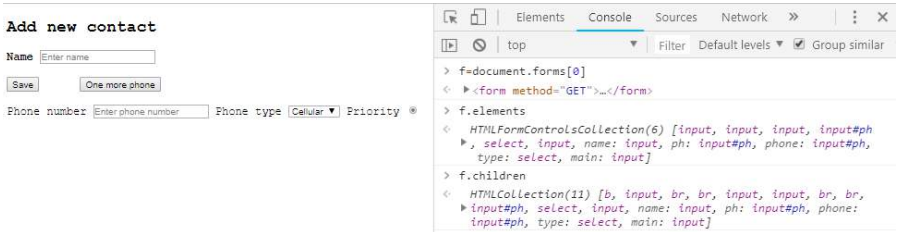
Save changes in the `forms.html` file and refresh the corresponding page in the browser. Make sure that after refreshing the page, its appearance corresponds to the one shown in the figure, with only one line for data input.

At this stage, you can see the difference in collections of form elements and its children. Let's call the Developer Console (by pressing F12) if it was closed after the previous exercise. Let's request the first (zero in the array) form from the document collection and save it in the `f` variable by entering the following string into the console:

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

Display the form collections. Type `f.elements` in the console (and press Enter), then `f.children` (and again, Enter). Make sure that there are 6 elements (`5 input` and `1 select`) in the first collection, while there are 11 elements in the second collection including markup elements `b` and `br` (*see the figure on the next page*).

Please note that the form elements are included in both collections, that is, you can access them with the help of any of them. The detailed content of the collections can be viewed by clicking on the expandable "triangles".



Pressing the Add number button does not result in any actions yet, but the Save button sends the form (to be precise, this is not a button, but a `submit` element that automatically submits the form). When you click on it, data appear in the browser's address bar separated from the address by question mark. Fill in the fields, click Save, and make sure that the entered data are transferred to the URI.

If you correctly copied the page code and carefully looked at the data that got to the address bar after pressing Save, then, for sure, you've noticed that the address bar did not contain information about the name entered.

The form transmits data only from those elements to which the program name is assigned (the `name` attribute is defined). In our case, the program name is not assigned to the input element of the name. Therefore, the entered data are ignored. The same situation is with buttons – they are also elements of the form (as it was seen from the collections), but their values are not transmitted.

Try setting the `name` attribute to the name field by overwriting a string like this:

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

and make sure that after that, the name information is added to the URI. Also add the name attribute to the button and to the submit element, make sure that their values, which are captions on the buttons, also appear in the transmission list in this case.

Let's consider one more example illustrating the additional possibilities of managing styles of both the form itself and its elements. Let's take the Logbook authorization form (*see the figure above*) and make a much simplified version of it. Create a new file `form2.html` in your working folder.

The form itself is styled as a block element (`display:block`) with the blue background color (`background-color:#5599EE`). Set its width (`width:400px`), height (`height:160px`) and padding (`padding: 20px`). Save the style descriptions under the `#logForm` ID.

We'll get the following HTML document.

```
<!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>
```

Open the received document in the browser and make sure that a blue rectangle is displayed on a blank page.

Add the caption LogForm to the left side of the form. In the body of the document between the opening and closing tags of the form, insert the `<div id="formText">LogForm </div>` block. In style definitions, set white to it (`color:white`), font size (`font-size:40px`), bold (`font-weight:bold`).

In order for the form elements to be positioned to the right of the caption (wrap), let's set it to the left float (`float:left`), set the width close to it or half the width of the form (`width:200px`) and the height should be equal to the form (`height:200px`). Don't forget that the height calculation also includes padding added both from above and from below.

A complete style description of the block will be as follows:

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

Next, place the form elements: the city selector, the login and password fields, and the submit form button. All of them have a similar style of the rounded block, but the send button differs in color and text alignment. Let's create a style class that is responsible for the round shape of the elements. Most style definitions have already been repeated and are not commented on. The basis of rounding is properties of the border radius (`border-radius:15px`) and disabling the border itself (`border-width:0`).

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

To highlight the Send button with green and give it center alignment, let's create the following style definition:

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

Now let's create form elements. In the body of the form, create a city selector. As an example, we restrict ourselves to three. It should be noted that the selector does not support

a placeholder, therefore, the technology of creating an additional selection field (option) with the attributes of hidden and disabled is applied. At the same time, this option is set as the selected one, and its value is set to empty (`value=""`), which will allow it to be controlled by software in the future. Placeholder option is placed in the last place in the selector. The resulting code for the selector is as follows:

```
<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>
```

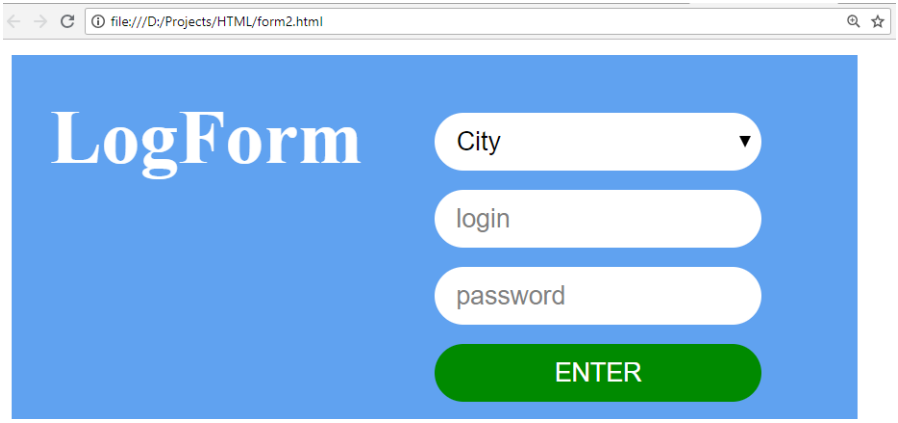
Creating the remaining elements does not contain any tricks. Specify the class `rounded` for all of them, which is responsible for the rounded frame. For the submit form button, set an additional identifier (`enter`) that specifies its features. The HTML code of elements will look as follows:

```
<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"/>
```

Full code of the `form2.html` file is available in the Sources_5.1 folder, its archive file is attached to the PDF of this lesson.

After completing the code execution, save the result and open the `forms2.html` file in the browser. If it is already open, then refresh the page by pressing F5 or by clicking the corresponding button on the browser interface.

As a result, our form is as follows in the browser:



Fill in the form fields, click ENTER. Make sure that the entered data fall into the address bar of the browser.

Enable the developer console and inspect the collections of `document.forms`, `document.forms[0].elements`, `document.forms[0].children` as described above.

Homework

1. Complete the created form of a phone book with the possibility to:
   - input dates of birth (`input type="date"`);
   - select a file with a photo (`input type="file"`);
   - input e-mail (`input type="email"`);
   - input a website (`input type="url"`);
   - input the "secret word" (`input type="password"`);
   - choose favorite color (`input type="color"`);
   - check the "remind birthday" checkbox (`input type="checkbox"`);
   - reset input data (`input type="reset"`).
2. Note in what format data are transferred from each of the form elements (in the browser's address bar).
3. Apply style definitions for form elements by preference.

Homework Questions

1. What are HTML forms? What are their purposes?
2. How to get programmatic access to a form? To all forms?
3. What are form elements? How can you access them?
4. In what ways can you specify that the element belongs to the form?
5. Under what conditions are data from an element transmitted with form being sent?

Programming Form Elements

Element Collections Revisited

Let's return to the form of a phone book created earlier and consider in more detail its structure from different points of view. Let's remind the code of this form and its appearance:

```
<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>
```

As we already know, there are two collections for storing information about the form structure: `elements` and `children`. The first one stores only elements of the form, the second one all the children. However, there is another collection, `childNodes`, which stores all the child nodes. To understand the difference between these collections, we will analyze them in detail.

To analyze collections, we will create tools for displaying them on the browser page. Of course, you can carry out analysis using the developer console tools, as we've done it before, but it is more convenient to create special buttons for reuse. After refreshing a page in the console, you will have to repeat the steps on accessing the form and its collections, while clicking on the button will be much easier.

After the form declaration, create an additional button that will provide analysis of the `elements` collection, and also add a paragraph for displaying data on the page. Make sure to add elements after the form (after the closing tag `</form>`), otherwise the form collections we are interested in will change after adding new buttons to the form.

Code snippet for creating a button and paragraph is shown below, the full code iis available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file form5_2-1.html).

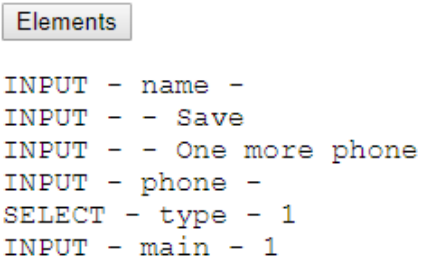
```
<button onclick="showElements()">Elements</button>
<p id="out"></p>
```

In the section of script description, add the showElements() function that will be responsible for button click:

```
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>"
     }
}
```

Function requests a collection of elements of the first form of the document (document.forms[0].elements), accesses the paragraph intended for displaying information (p=document.getElementById("out")), then it cycles through the collection and displays data on tag name (tagName), program name (name) and value (value) of each element. To separate the output, use the - sign with spaces on both sides.

After clicking on the button, the following content appears on the page:



We've already observed these 6 elements in the console in the previous lesson. You can see that, for example, the first input element has the name name, but does not have value (empty

space after the second -). This is an empty entry string for the contact's name. You can enter arbitrary data into it, click the Elements button again, and make sure our conclusions are correct.

On the contrary, the second element does not have a name (empty space between the hyphens -), but it has the value Save. This is a button (more precisely, the submit element) of sending the form. We did not give it a name intentionally to exclude this item from the list of data transferred to the server.

At the same time, the last elements have both a name and a value. This is achieved by applying the `selected` attributes for the list item (`select`) and `checked` for the radio button (the last `input`).

Similarly, let's analyze the `children` collection. Let's add the appropriate button immediately after the Elements button, use the same paragraph for output, we won't create a new one.

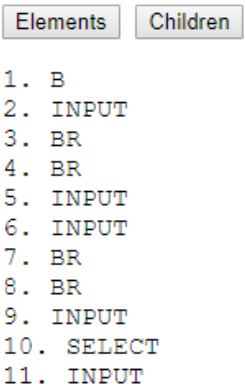
```
<button onclick="showChildren()">Children</button>
```

In the `<script>` section, let's describe the function responsible for button click:

```
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>"
     }
}
```

The function content almost completely repeats the previous one, but first, `children` are used as the analyzed collection. Second, we display only the element's serial number and name of its tag since values are relevant only for form elements and names are also not given to all elements.

Save the modified file and refresh the page in the browser. After clicking the Children button, the following text will appear.



These are eleven elements also familiar to us from the previous lesson that correspond to the form structure. The first `B` object is a bold Name caption in front of the name input field. The `BR` objects are line breaks, which we applied for design.

Similarly, after the Children button, let's add a third button that analyzes the most complete `childNodes` collection.

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

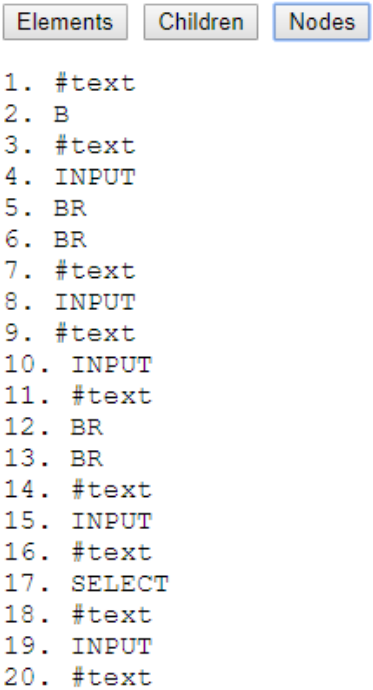
Its click function will look as follows:

```
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>"
    }
}
```

In this collection, not all elements have a tag name, so the `nodeName` field is used, the most common representation of the DOM object. The full code is available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file form5_2-1.html).

Save the changes, refresh the page, and click on the Nodes button. This will result in the following:



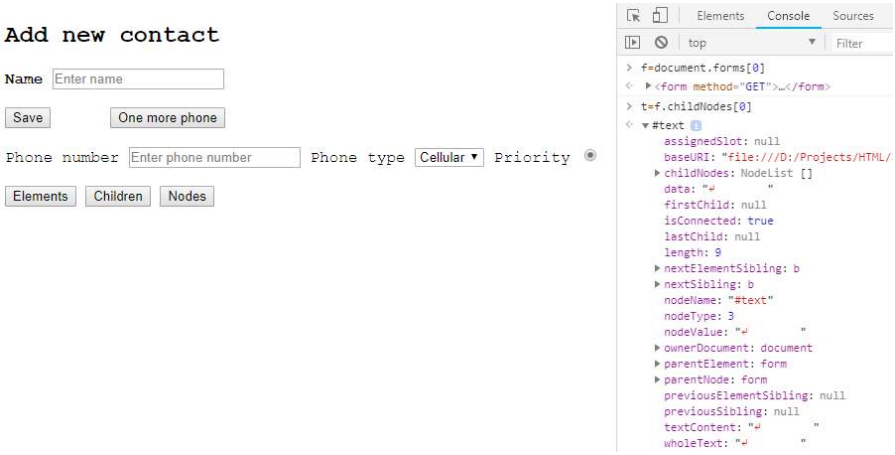
Nine elements with the node name `#text` were added to the existing 11 elements of the `children` collection. These are text elements related to the actual design of the page, including not fully intentional.

Thus, the first `#text` node corresponds to the separating element between the opening tag of the form and the beginning of the Name caption. Note that in the source code, this caption (`<b>Name</b>`) is formed from a new line after the `<form>` tag and contains an indent at the beginning of the line for the code's beauty:

```
<form method='GET'> ← line break
 <b>Name</b>
↑ indent
```

Call the browser console (F12) and ask for the details of this element. Enter `f=document.forms[0]` to get the form (press Enter). Next, introduce `t=f.childNodes[0]` to detail the first (zero) element of the `childNodes` collection of the form. After pressing Enter, we will see `#text`. Let's click on the triangle to the left of the caption to expand detailed data and get the field list of this object (*see fig. on page 35*).

Its contents are duplicated in several fields (`data`, `nodeValue`, `textContent`, `wholeText`) and represents a line break (symbol ↲) followed by several spaces corresponding to the indent in the HTML code of the form. As you probably remember, any sequence of space and break characters is treated by the browser as a single space, which is displayed in the collection mode of line elements and is ignored when building block or floating elements.



We can verify this by examining the second element of `#text` in the `childNodes` collection. Apparently, it is located in the third place in the collection between the elements `B` and `INPUT`. Looking closely at the HTML code of the form, pay attention to the space after the closing `</b>` tag and opening `<input...` in the `<b>Name</b> <input type="text"` line (the gap is increased for clarity).

Remove this space in the HTML code so that the tags follow each other inseparably (`</b><input...`). Save the file and refresh the page in the browser. Make sure that the space between Name and input field disappears:

And when the Nodes button is clicked, 19 elements are displayed without the `#text` element at the 3rd position that we examined.

You can repeat the steps described above to exclude the first `#text` element by removing spaces and line break between the form tags and the Name caption. Make sure that the element disappears from the collection, but the appearance of the page does not change in any way since the browser has not yet entered the mode of assembling line elements, and spaces are ignored.

The `#text` elements, in addition to HTML code, also correspond to the captions that are not made in their own tags. For example, the 14th node `#text` (in the original collection of 20 elements) corresponds with the caption Phone number. This can be verified by repeating the above actions in the browser console (`f=document.forms[0], t=f.childNodes[13]`).

In addition to the caption, this element also contains a line break and an indent before and after the text. That is, the element collects everything that is between the closing tag (in this case, `<br/>`) and the next opening (`input` for entering phone number). Again, note that for the Name caption enclosed in the `<b>` tag, the `B` object is created instead of the `#text` node.

Let's summarize some results on the study of the form structure and its collections.

All the means of designing HTML code in the form of line breaks, indents (spaces or tabs) between the tags fall into the DOM structure of the page eventually. They increase the collections of child elements and, naturally, memory used and performance of refreshing a page or a part of it. In our simple example, the form has almost half of such elements (9 of 20), which does not allow them to be neglected justifying

that there are not so many of them. Of course, you should not abandon the beautiful design of HTML code, at least at the development stage. But at the delivery of the finished product, you can think about optimization.

Programming style may contain optimized constructions. For example, if you move space in the Name caption inside the `<b>` tag, that is, write `<b>Name</b><input...` instead of `<b>Name</b>  <input...` (spaces are enlarged for clarity) then the extra `#text` element will not be created, and the indent between the caption and the input field will be preserved.

Spaces inside the tags (`<b>Name  </ b>`), as well as spaces that separate their attributes (`<form  method='GET'>`) are not transferred to the DOM structure as separate objects. This can be used for code design tasks and for preventing creation of new elements.

Captions designed without their own tags fall into special `#text` objects. On the one hand, it can serve as a recommendation to still enclose captions in tags for a more predictable document structure. On the other hand, the extension of the form by creating additional captions in the process of executing program scripts (as they say, "on the fly") will require us to use special `#text` nodes that do not correspond to any standard tag.

Adding Elements

After understanding features of the form structure and its elements, let's set the task to implement the functionality of adding a new field for entering an additional phone number

by clicking the Add number button. Initially, we created the event handler for the One more phone button as a function `add_click()` (`onclick="add_click()"`) for this purpose.

In order to add a new phone number, you need to create multiple additional elements and include them in the form. Let's analyze in detail the composition of a single line for entering a phone number:

1. First there is a break (`<br>`) that provides transition to a new line.
2. Then the caption: Phone number.
3. Next, the `INPUT` element for entering the phone number.
4. The caption: Phone type.
5. Selector of selecting number type.
6. The caption: Priority.
7. Radio button for marking the main number.

Also, remember that the form data is sent only from those elements which have names set (the `name` attribute). And names must be different for different fields, so that there is an opportunity to split data on the server side. The exception is only the radio buttons for selecting the main number, for which the name should be the same to ensure their dependence among themselves.

To form different names for new input elements, let's introduce the global counter variable `phoneCounter`, whose value we will assign to the field names. At the time of the first start of the function, this variable must be initialized. Let's use the test `if(typeof phoneCounter == 'undefined')` to

determine the first run of the function at which this variable should be set to 1. After that, the counter can be incremented in the usual way (phoneCounter++). Accordingly, the beginning of the function will look like this (full code with all functions is available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file form5_2-2.html).

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

The first element of the collection of document forms is saved in the f variable, which is our main form to which we will add elements.

To create new elements in Javascript, several functions are provided. The document.createElement function creates an element by the name of the tag, for example, a line break can be created by calling document.createElement('br'), and the input element is document.createElement('input'). Another function, document.createTextNode, is used to create nodes of the #text type for which the tag name is not specified. As an argument, the function takes text to be displayed. For example, document.createTextNode('Phone number').

It should be noted that the described functions only create objects of a given type but do not include them in the structure of the document. For this purpose, you need to use the appendChild method of the object to which elements are added. In our case, it's forms stored in the f variable.

Thus, adding a line break to the form will look as follows:

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

And adding the Phone number caption as follows:

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

Adding input field for a new number will require additional actions. The newly created element of the `input` type must explicitly specify the `'text'` type (in the previous section, it was shown that the element can behave quite differently depending on the type). Also, it is necessary to form a name considering counter variable and, by analogy with the already existing field, add a placeholder: Enter phone number.

As a result, the creation of the field and its addition to the form will have the following code:

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

Adding the captions Phone type and Priority, as well as the radio buttons (the `input` variant) are completely similar to the examples described above.

Let us dwell on adding a list of selecting type of phone number.

Let's recall its structure:

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

In order to repeat the same element, create an element of the `select` type, three elements of the `option` type, fill them with data and add them to the parent `select`. However, in this case, you can use one more function that allows you to create copies (clones) of elements – `cloneNode`. Since the new element will differ only in name, this way will be simpler – create a clone and change the name instead of re-creating a new element step-by-step with the identical internal structure.

In order to create a clone of an element, you need to get access to it. You can do this in several ways; let's use the shortest collection of the `elements` form. Since the list has a name (`name='type'`), you can get access to it by accessing the collection by this name `f.elements['type']`. You can save this value to a program variable (`selector`) and then call the clone method to create a new variable `var newSelector = selector.cloneNode(true)`. The `true` argument in the function call indicates that you need to copy the entire contents of the element. Now we should only change name of the new object and add it to the form.

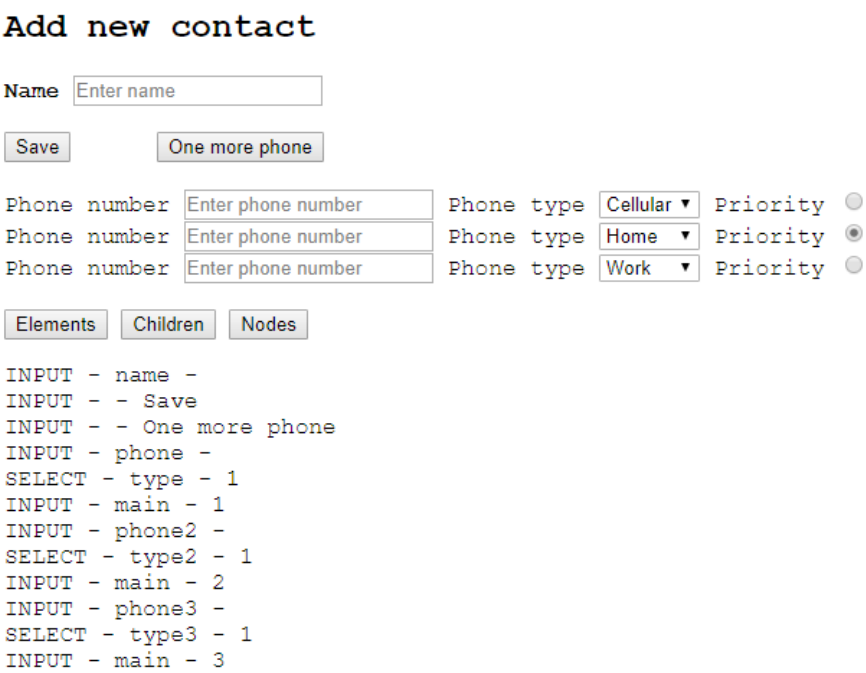
The full function code is shown below.

Full code with all functions is available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file 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);
}
```

Save the code and refresh the browser page. Now when you click on the One more phone button, additional input "lines" appear. These lines appear before collection research buttons since they are taken outside the form, and lines are added to the form itself. Make sure that when you click on these buttons, new items (`phone2, type2, phone3, type3, ...`) are displayed in the collection. Check that the radio buttons work correctly and selection lists do not depend on each other.



Enter the test data and click Save. Verify that the entered data are correct and complete in the address bar of the browser.

About Cloning and Inserting in Detail

As we have seen, the creation of complex elements with a complex internal structure can be simplified by applying the clone method. It should be remembered that a clone creates a complete copy of an object including names and values of all compound blocks. If you need to change all these names and values in the new element, then the clone method will not be any simpler or shorter than recreating the object by one element. If there are not many changes in the new complex then cloning really simplifies the code.

In our case, the new "line" for phone number input completely repeats the previous one except for the new field names for phone number and its type. You can expect that cloning simplifies addition of elements.

To use cloning, put the elements that are added in a common container ( ). Let's change the definition of the form:

```
<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>
```

The One more phone and Save buttons are moved down, the "line" with the number, type and radio button is placed in the `<div>` block. Since the block element is created in a new line, breaks (`<br>`) in the new form are not used, and the margin is specified by the style of the wrapper (`style="margin:10px 0;"`).

In the new form, the difference between the `elements` and `children` collections is even more apparent. Request the composition of these collections in the console. In the `elements` collection, the same 6 elements are visible without grouping into separate blocks. Whereas the `children` collection have only 5 elements, the third of them is a `div` block, where several elements of the form are grouped.

This should be remembered when working with forms: the `elements` collection does not store information about grouping elements in blocks, but `children` stores.

Let's also change the `add_click()` function. Full code with all functions is available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file 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]);
}
```

The first lines coincide with the previous version of the function and were commented on earlier.

We get access to the `<div>` block, which is responsible for the "line" of phone number input. As it was already checked in the browser console, this block is on the third position of the `children` form collection. Accordingly, you can access it using the line `var line = f.children[2]`.

Then a clone of this block is created (`var newLine = line.cloneNode(true)`).

After cloning, replace the names of input fields considering the counter, and reset the values of the input field for name

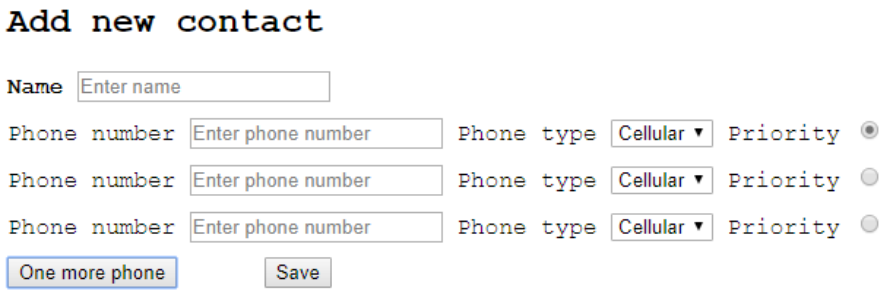
(`newLine.children[0].value = ''`) and the radio button (`newLine.children[2].checked = false`) so that cloning creates empty field and radio button was not clicked.

The last line of code adds a new element to the form. But instead of the previously considered `appendChild` method, the `insertBefore` method is used. The `appendChild` method adds an element to the end of the collection, while `insertBefore` can insert into any place. Because we moved the add and send buttons, `appendChild` will add lines after the buttons which is not entirely correct in terms of design.

In order to use the `insertBefore` method, you need to specify two arguments. The first argument of this method is a new element added to the form. The second is the element before which you need to insert a new one.

The `phoneCounter` entered for field names helps to determine the "place" where you want to insert a new line. Its value allows us to determine the number of "lines" and insert a new line at the end of the others but before adding and sending form buttons.

Save the changes, refresh the browser page, make sure the buttons work.



Refusal of line breaks (
) and use of margins allowed smooth adjustment of the distance between elements. Visually, the elements do not seem as dense as in the previous examples.

So, the clone mechanism really allowed to significantly simplify creation of a group of new elements. The insertBefore method, in turn, provides a more flexible way of adding form elements.

Deleting Elements

Finally, consider the process of removing elements programmatically.

To remove the elem element from the child nodes of the obj object, use the obj.removeChild(elem) method. That is, to delete an element, you must first obtain programmatic access to the parent object, then to the child element that you want to delete and then call the removeChild method of the parent object transferring child as a parameter.

Elements can be deleted by one or by blocks of elements. In the latter case, the elements should be grouped in some way (for example, in the block) and reference to the group of elements must be passed to the removeChild method.

Expand the functionality of our form by adding the ability to delete records that were added during the work (that is, all except the first one). Grouping of form elements into blocks made for clone tasks will simplify the process of deletion.

First, you need to modify the add_click() function in which you create new "strings" to input phone numbers. Let's add button for deleting this line to the composition of new

lines. Before the last line of the `add_click()` function, let's add the following code (full function code is available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file form5_2-4.html).

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

Most operations have already been described above: an element of the `input`" type is created, its type is `button` and the value (caption on the button) is `Remove`. Next, set the click event handler, linking it to the `rem_click` function using the `addEventListener` method. Finally, add the button to the cloned `newLine` block.

Second, you need to create a `rem_click` function, which will be responsible for clicking the button and, in fact, deleting the line. Since the same function is used for all buttons, we will distinguish them using a message source, that is, prototype of the handler function should provide the argument: `rem_click (event)`.

When a button event occurs, its handler is called and the `event` object is passed. In this event, message source is stored in the `event.target` field. It is this construction that will enable us to determine the button that was clicked.

However, we are not interested in the button itself but in the `<div>` block wherein it is located. You can access this

block in several ways: using the `parentNode` method or the `closest` method. The parentNode method returns parent element, which is a `<div>` block for the button. The closest method allows you to find the closest parent of the given type. The second method is preferable if the button is nested in any additional elements. Accordingly, to determine the "line" block, in which the button was pressed, you can use the following code:

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

The parent object for the block is form. We have repeatedly received access to the form through collection of document forms. The final code for the rem_click function is as follows:

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

At the end of the function, we must not forget to reduce the counter of our blocks because this value is used to find the last line.

After compiling all the functions, check efficiency of all form functions by means of experiment.

* Due to the implementation of deleting blocks in the program, a potential error has appeared. Suppose a user double-clicked the Add Phone Number button. In this case, the fields with

the names phone2 and phone3 are formed in new rows. After that, the user deletes the second (middle) line (with phone2), and the last (with phone3) remains. The counter variable phoneCounter in the delete function (rem_click) will be reduced (from 3 to 2). If you click the add button again, the counter will increase (from 2 to 3) and form the phone3 name again. As a result, the form will have two lines with the same field names (phone3). You cannot refuse to reduce the counter because its value is used to determine the last line in the form.

This problem can be solved by dividing counters to form names and for remaining number of rows.

Try to do it yourself. The finished solution is available in the Sources_5.2 folder, its archive file is attached to the PDF of this lesson (file form5_2-5.html).

Homework

1. Implement the ability to add several dates with the choice of type to the contact: birthday, profession day, day of the first meeting (you can add or change the list). Use the `<select>` element to select date type.
2. Implement the ability to download several files with the signatures of each of them. The content of the signature is not limited to (`<textarea>`).
3. Implement the ability to delete the added elements (paragraphs 1-2) separately.
4. Check functionality of the form, make sure that all data, including from the added elements, are transmitted when the form is submitted (they fall into the address bar of the browser).

Homework questions

1. Reveal the sequence to add a new element to the form.
2. What is the difference between the `appendChild` and `insertBefore` methods?
3. What is the purpose of mechanism for clone elements? What is the order of creating a clone?
4. How do I set or change name of a newly created element? How do you set an event handler to it?
5. How do you delete an element from a form?

Lesson №5
Forms