**Forms and Form Fields**

I shall this very day, at Doctor’s feast,  
My bounden service duly pay thee.  
But one thing!—For insurance’ sake, I pray thee,  
Grant me a line or two, at least.

Mephistopheles, *in Goethe's Faust*

Forms were introduced briefly in the [previous chapter](http://eloquentjavascript.net/17_http.html#http_forms) as a way to *submit* information provided by the user over HTTP. They were  designed for a pre-JavaScript Web, assuming that interaction with the server always happens by navigating to a new page.

But their elements are part of the DOM like the rest of the page, and the DOM elements that represent form fields support a number of properties and events that are not present on other elements. These make it possible to inspect and control such input fields with JavaScript programs and do things such as adding functionality to a traditional form or using forms and fields as building blocks in a JavaScript application.

**Fields**

A web form consists of any number of input fields grouped in a <form> tag. HTML allows a number of different styles of fields, ranging from simple on/off checkboxes to drop-down menus and fields for text input. This book won’t try to comprehensively discuss all field types, but we will start with a rough overview.

A lot of field types use the <input> tag. This tag’s type attribute is used to select the field’s style. These are some commonly used <input> types:

|  |  |
| --- | --- |
| text | A single-line text field |
| password | Same as text but hides the text that is typed |
| checkbox | An on/off switch |
| radio | (Part of) a multiple-choice field |
| file | Allows the user to choose a file from their computer |

Form fields do not necessarily have to appear in a <form> tag. You can put them anywhere in a page. Such fields cannot be submitted (only a form as a whole can), but when responding to input with JavaScript, we often do not want to submit our fields normally anyway.



1

<p><input type="text" value="abc"> (text)</p>

2

<p><input type="password" value="abc"> (password)</p>

3

<p><input type="checkbox" checked> (checkbox)</p>

4

<p><input type="radio" value="A" name="choice">

5

<input type="radio" value="B" name="choice" checked>

6

<input type="radio" value="C" name="choice"> (radio)</p>

7

<p><input type="file"> (file)</p>

The JavaScript interface for such elements differs with the type of the element. We’ll go over each of them later in the chapter.

Multiline text fields have their own tag, <textarea>, mostly because using an attribute to specify a multiline starting value would be awkward. The <textarea> requires a matching </textarea> closing tag and uses the text between those two, instead of using its value attribute, as starting text.

<textarea>

one

two

three

</textarea>

Finally, the <select> tag is used to create a field that allows the user to select from a number of predefined options.

<select>

<option>Pancakes</option>

<option>Pudding</option>

<option>Ice cream</option>

</select>

Whenever the value of a form field changes, it fires a "change" event.

**Focus**

Unlike most elements in an HTML document, form fields can get *keyboard focus*. When clicked—or activated in some other way—they become the currently active element, the main recipient of keyboard input.

If a document has a text field, text typed will end up in there only when the field is focused. Other fields respond differently to keyboard events. For example, a <select> menu tries to move to the option that contains the text the user typed and responds to the arrow keys by moving its selection up and down.

We can control focus from JavaScript with the focus and blur methods. The first moves focus to the DOM element it is called on, and the second removes focus. The value in document.activeElement corresponds to the currently focused element.

<input type="text">

<script>

document.querySelector("input").focus();

console.log(document.activeElement.tagName);

// → INPUT

document.querySelector("input").blur();

console.log(document.activeElement.tagName);

// → BODY

</script>

For some pages, the user is expected to want to interact with a form field immediately. JavaScript can be used to focus this field when the document is loaded, but HTML also provides the autofocus attribute, which produces the same effect but lets the browser know what we are trying to achieve. This makes it possible for the browser to disable the behavior when it is not appropriate, such as when the user has focused something else.

<input type="text" autofocus>

Browsers traditionally also allow the user to move the focus through the document by pressing the Tab key. We can influence the order in which elements receive focus with the tabindex attribute. The following example document will let focus jump from the text input to the OK button, rather than going through the help link first:

<input type="text" tabindex=1> <a href=".">(help)</a>

<button onclick="console.log('ok')" tabindex=2>OK</button>

By default, most types of HTML elements cannot be focused. But you can add a tabindex attribute to any element, which will make it focusable.

**Disabled fields**

All form fields can be *disabled* through their disabled attribute, which also exists as a property on the element’s DOM object.

<button>I'm all right</button>

<button disabled>I'm out</button>

Disabled fields cannot be focused or changed, and unlike active fields, they usually look gray and faded.

When a program is in the process of handling an action caused by some button or other control, which might require communication with the server and thus take a while, it can be a good idea to disable the control until the action finishes. That way, when the user gets impatient and clicks it again, they don’t accidentally repeat their action.

**The form as a whole**

When a field is contained in a <form> element, its DOM element will have a property form linking back to the form’s DOM element. The <form> element, in turn, has a property called elements that contains an array-like collection of the fields inside it.

The name attribute of a form field determines the way its value will be identified when the form is submitted. It can also be used as a property name when accessing the form’s elements property, which acts both as an array-like object (accessible by number) and a map (accessible by name).

<form action="example/submit.html">

Name: <input type="text" name="name"><br>

Password: <input type="password" name="password"><br>

<button type="submit">Log in</button>

</form>

<script>

var form = document.querySelector("form");

console.log(form.elements[1].type);

// → password

console.log(form.elements.password.type);

// → password

console.log(form.elements.name.form == form);

// → true

</script>

A button with a type attribute of submit will, when pressed, cause the form to be submitted. Pressing Enter when a form field is focused has the same effect.

Submitting a form normally means that the browser navigates to the page indicated by the form’s action attribute, using either a GET or a POST request. But before that happens, a "submit" event is fired. This event can be handled by JavaScript, and the handler can prevent the default behavior by calling preventDefault on the event object.

<form action="example/submit.html">

Value: <input type="text" name="value">

<button type="submit">Save</button>

</form>

<script>

var form = document.querySelector("form");

form.addEventListener("submit", function(event) {

console.log("Saving value", form.elements.value.value);

event.preventDefault();

});

</script>

Intercepting "submit" events in JavaScript has various uses. We can write code to verify that the values the user entered make sense and immediately show an error message instead of submitting the form when they don’t. Or we can disable the regular way of submitting the form entirely, as in the previous example, and have our program handle the input, possibly using XMLHttpRequest to send it over to a server without reloading the page.

**Text fields**

Fields created by <input> tags with a type of text or password, as well as textarea tags, share a common interface. Their DOM elements have a valueproperty that holds their current content as a string value. Setting this property to another string changes the field’s content.

The selectionStart and selectionEnd properties of text fields give us information about the cursor and selection in the text. When nothing is selected, these two properties hold the same number, indicating the position of the cursor. For example, 0 indicates the start of the text, and 10 indicates the cursor is after the 10th character. When part of the field is selected, the two properties will differ, giving us the start and end of the selected text. Like value, these properties may also be written to.

As an example, imagine you are writing an article about Khasekhemwy but have some trouble spelling his name. The following code wires up a <textarea> tag with an event handler that, when you press F2, inserts the string “Khasekhemwy” for you.

<textarea></textarea>

<script>

var textarea = document.querySelector("textarea");

textarea.addEventListener("keydown", function(event) {

// The key code for F2 happens to be 113

if (event.keyCode == 113) {

replaceSelection(textarea, "Khasekhemwy");

event.preventDefault();

}

});

function replaceSelection(field, word) {

var from = field.selectionStart, to = field.selectionEnd;

field.value = field.value.slice(0, from) + word +

field.value.slice(to);

// Put the cursor after the word

field.selectionStart = field.selectionEnd =

from + word.length;

}

</script>

The replaceSelection function replaces the currently selected part of a text field’s content with the given word and then moves the cursor after that word so that the user can continue typing.

The "change" event for a text field does not fire every time something is typed. Rather, it fires when the field loses focus after its content was changed. To respond immediately to changes in a text field, you should register a handler for the "input" event instead, which fires for every time the user types a character, deletes text, or otherwise manipulates the field’s content.

The following example shows a text field and a counter showing the current length of the text entered:

<input type="text"> length: <span id="length">0</span>

<script>

var text = document.querySelector("input");

var output = document.querySelector("#length");

text.addEventListener("input", function() {

output.textContent = text.value.length;

});

</script>

**Checkboxes and radio buttons**

A checkbox field is a simple binary toggle. Its value can be extracted or changed through its checked property, which holds a Boolean value.

<input type="checkbox" id="purple">

<label for="purple">Make this page purple</label>

<script>

var checkbox = document.querySelector("#purple");

checkbox.addEventListener("change", function() {

document.body.style.background =

checkbox.checked ? "mediumpurple" : "";

});

</script>

The <label> tag is used to associate a piece of text with an input field. Its forattribute should refer to the id of the field. Clicking the label will activate the field, which focuses it and toggles its value when it is a checkbox or radio button.

A radio button is similar to a checkbox, but it’s implicitly linked to other radio buttons with the same name attribute so that only one of them can be active at any time.

Color:

<input type="radio" name="color" value="mediumpurple"> Purple

<input type="radio" name="color" value="lightgreen"> Green

<input type="radio" name="color" value="lightblue"> Blue

<script>

var buttons = document.getElementsByName("color");

function setColor(event) {

document.body.style.background = event.target.value;

}

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

buttons[i].addEventListener("change", setColor);

</script>

The document.getElementsByName method gives us all elements with a givenname attribute. The example loops over those (with a regular for loop, not forEach, because the returned collection is not a real array) and registers an event handler for each element. Remember that event objects have a targetproperty referring to the element that triggered the event. This is often useful in event handlers like this one, which will be called on different elements and need some way to access the current target.

**Select fields**

Select fields are conceptually similar to radio buttons—they also allow the user to choose from a set of options. But where a radio button puts the layout of the options under our control, the appearance of a <select> tag is determined by the browser.

Select fields also have a variant that is more akin to a list of checkboxes, rather than radio boxes. When given the multiple attribute, a <select> tag will allow the user to select any number of options, rather than just a single option.

<select multiple>

<option>Pancakes</option>

<option>Pudding</option>

<option>Ice cream</option>

</select>

This will, in most browsers, show up differently than a non-multiple select field, which is commonly drawn as a *drop-down* control that shows the options only when you open it.

The size attribute to the <select> tag is used to set the number of options that are visible at the same time, which gives you explicit control over the drop-down’s appearance. For example, setting the size attribute to "3" will make the field show three lines, whether it has the multiple option enabled or not.

Each <option> tag has a value. This value can be defined with a valueattribute, but when that is not given, the text inside the option will count as the option’s value. The value property of a <select> element reflects the currently selected option. For a multiple field, though, this property doesn’t mean much since it will give the value of only *one* of the currently selected options.

The <option> tags for a <select> field can be accessed as an array-like object through the field’s options property. Each option has a property called selected, which indicates whether that option is currently selected. The property can also be written to select or deselect an option.

The following example extracts the selected values from a multiple select field and uses them to compose a binary number from individual bits. Hold Ctrl (or Command on a Mac) to select multiple options.

<select multiple>

<option value="1">0001</option>

<option value="2">0010</option>

<option value="4">0100</option>

<option value="8">1000</option>

</select> = <span id="output">0</span>

<script>

var select = document.querySelector("select");

var output = document.querySelector("#output");

select.addEventListener("change", function() {

var number = 0;

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

var option = select.options[i];

if (option.selected)

number += Number(option.value);

}

output.textContent = number;

});

</script>

**File fields**

File fields were originally designed as a way to upload files from the browser’s machine through a form. In modern browsers, they also provide a way to read such files from JavaScript programs. The field acts as a manner of gatekeeper. The script cannot simply start reading private files from the user’s computer, but if the user selects a file in such a field, the browser interprets that action to mean that the script may read the file.

A file field usually looks like a button labeled with something like “choose file” or “browse”, with information about the chosen file next to it.

<input type="file">

<script>

var input = document.querySelector("input");

input.addEventListener("change", function() {

if (input.files.length > 0) {

var file = input.files[0];

console.log("You chose", file.name);

if (file.type)

console.log("It has type", file.type);

}

});

</script>

The files property of a file field element is an array-like object (again, not a real array) containing the files chosen in the field. It is initially empty. The reason there isn’t simply a file property is that file fields also support a multiple attribute, which makes it possible to select multiple files at the same time.

Objects in the files property have properties such as name (the filename), size (the file’s size in bytes), and type (the media type of the file, such as text/plain or image/jpeg).

What it does not have is a property that contains the content of the file. Getting at that is a little more involved. Since reading a file from disk can take time, the interface will have to be asynchronous to avoid freezing the document. You can think of the FileReader constructor as being similar to XMLHttpRequest but for files.

<input type="file" multiple>

<script>

var input = document.querySelector("input");

input.addEventListener("change", function() {

Array.prototype.forEach.call(input.files, function(file) {

var reader = new FileReader();

reader.addEventListener("load", function() {

console.log("File", file.name, "starts with",

reader.result.slice(0, 20));

});

reader.readAsText(file);

});

});

</script>

Reading a file is done by creating a FileReader object, registering a "load"event handler for it, and calling its readAsText method, giving it the file we want to read. Once loading finishes, the reader’s result property contains the file’s content.

The example uses Array.prototype.forEach to iterate over the array since in a normal loop it would be awkward to get the correct file and reader objects from the event handler. The variables would be shared by all iterations of the loop.

FileReaders also fire an "error" event when reading the file fails for any reason. The error object itself will end up in the reader’s error property. If you don’t want to remember the details of yet another inconsistent asynchronous interface, you could wrap it in a Promise (see [Chapter 17](http://eloquentjavascript.net/17_http.html#promises)) like this:

function readFile(file) {

return new Promise(function(succeed, fail) {

var reader = new FileReader();

reader.addEventListener("load", function() {

succeed(reader.result);

});

reader.addEventListener("error", function() {

fail(reader.error);

});

reader.readAsText(file);

});

}

It is possible to read only part of a file by calling slice on it and passing the result (a so-called *blob* object) to the file reader.

**Storing data client-side**

Simple HTML pages with a bit of JavaScript can be a great medium for “mini applications”—small helper programs that automate everyday things. By connecting a few form fields with event handlers, you can do anything from converting between degrees Celsius and Fahrenheit to computing passwords from a master password and a website name.

When such an application needs to remember something between sessions, you cannot use JavaScript variables since those are thrown away every time a page is closed. You could set up a server, connect it to the Internet, and have your application store something there. We will see how to do that in [Chapter 20](http://eloquentjavascript.net/20_node.html#node). But this adds a lot of extra work and complexity. Sometimes it is enough to just keep the data in the browser. But how?

You can store string data in a way that survives page reloads by putting it in the localStorage object. This object allows you to file string values under names (also strings), as in this example:

localStorage.setItem("username", "marijn");

console.log(localStorage.getItem("username"));

// → marijn

localStorage.removeItem("username");

A value in localStorage sticks around until it is overwritten, it is removed with removeItem, or the user clears their local data.

Sites from different domains get different storage compartments. That means data stored in localStorage by a given website can, in principle, only be read (and overwritten) by scripts on that same site.

Browsers also enforce a limit on the size of the data a site can store in localStorage, typically on the order of a few megabytes. That restriction, along with the fact that filling up people’s hard drives with junk is not really profitable, prevents this feature from eating up too much space.

The following code implements a simple note-taking application. It keeps the user’s notes as an object, associating note titles with content strings. This object is encoded as JSON and stored in localStorage. The user can select a note from a <select> field and change that note’s text in a <textarea>. A note can be added by clicking a button.

Notes: <select id="list"></select>

<button onclick="addNote()">new</button><br>

<textarea id="currentnote" style="width: 100%; height: 10em">

</textarea>

<script>

var list = document.querySelector("#list");

function addToList(name) {

var option = document.createElement("option");

option.textContent = name;

list.appendChild(option);

}

// Initialize the list from localStorage

var notes = JSON.parse(localStorage.getItem("notes")) ||

{"shopping list": ""};

for (var name in notes)

if (notes.hasOwnProperty(name))

addToList(name);

function saveToStorage() {

localStorage.setItem("notes", JSON.stringify(notes));

}

var current = document.querySelector("#currentnote");

current.value = notes[list.value];

list.addEventListener("change", function() {

current.value = notes[list.value];

});

current.addEventListener("change", function() {

notes[list.value] = current.value;

saveToStorage();

});

function addNote() {

var name = prompt("Note name", "");

if (!name) return;

if (!notes.hasOwnProperty(name)) {

notes[name] = "";

addToList(name);

saveToStorage();

}

list.value = name;

current.value = notes[name];

}

</script>

The script initializes the notes variable to the value stored in localStorageor, if that is missing, to a simple object with only an empty "shopping list"note in it. Reading a field that does not exist from localStorage will yield null. Passing null to JSON.parse will make it parse the string "null" and return null. Thus, the || operator can be used to provide a default value in a situation like this.

Whenever the note data changes (when a new note is added or an existing note changed), the saveToStorage function is called to update the storage field. If this application was intended to handle thousands of notes, rather than a handful, this would be too expensive, and we’d have to come up with a more complicated way to store them, such as giving each note its own storage field.

When the user adds a new note, the code must update the text field explicitly, even though the <select> field has a "change" handler that does the same thing. This is necessary because "change" events fire only when the *user*changes the field’s value,  not when a script does it.

There is another object similar to localStorage called sessionStorage. The difference between the two is that the content of sessionStorage is forgotten at the end of each session, which for most browsers means whenever the browser is closed.

**Summary**

HTML can express various types of form fields, such as text fields, checkboxes, multiple-choice fields, and file pickers.

Such fields can be inspected and manipulated with JavaScript. They fire the "change" event when changed, the "input" event when text is typed, and various keyboard events. These events allow us to notice when the user is interacting with the fields. Properties like value (for text and select fields) or checked (for checkboxes and radio buttons) are used to read or set the field’s content.

When a form is submitted, its "submit" event fires. A JavaScript handler can call preventDefault on that event to prevent the submission from happening. Form field elements do not have to be wrapped in <form> tags.

When the user has selected a file from their local file system in a file picker field, the FileReader interface can be used to access the content of this file from a JavaScript program.

The localStorage and sessionStorage objects can be used to save information in a way that survives page reloads. The first saves the data forever (or until the user decides to clear it), and the second saves it until the browser is closed.

**Exercises**

**A JavaScript workbench**

Build an interface that allows people to type and run pieces of JavaScript code.

Put a button next to a <textarea> field, which, when pressed, uses the Function constructor we saw in [Chapter 10](http://eloquentjavascript.net/10_modules.html#eval) to wrap the text in a function and call it. Convert the return value of the function, or any error it raised, to a string and display it after the text field.

<textarea id="code">return "hi";</textarea>

<button id="button">Run</button>

<pre id="output"></pre>

<script>

// Your code here.

</script>

**Autocompletion**

Extend a text field so that when the user types, a list of suggested values is shown below the field. You have an array of possible values available and should show those that start with the text that was typed. When a suggestion is clicked, replace the text field’s current value with it.

<input type="text" id="field">

<div id="suggestions" style="cursor: pointer"></div>

<script>

// Builds up an array with global variable names, like

// 'alert', 'document', and 'scrollTo'

var terms = [];

for (var name in window)

terms.push(name);

// Your code here.

</script>

**Conway’s Game of Life**

Conway’s Game of Life is a simple simulation that creates artificial “life” on a grid, each cell of which is either live or not. Each generation (turn), the following rules are applied:

* Any live cell with fewer than two or more than three live neighbors dies.
* Any live cell with two or three live neighbors lives on to the next generation.
* Any dead cell with exactly three live neighbors becomes a live cell.

A neighbor is defined as any adjacent cell, including diagonally adjacent ones.

Note that these rules are applied to the whole grid at once, not one square at a time. That means the counting of neighbors is based on the situation at the start of the generation, and changes happening to neighbor cells during this generation should not influence the new state of a given cell.

Implement this game using whichever data structure you find appropriate. Use Math.random to populate the grid with a random pattern initially. Display it as a grid of checkbox fields, with a button next to it to advance to the next generation. When the user checks or unchecks the checkboxes, their changes should be included when computing the next generation.

<div id="grid"></div>

<button id="next">Next generation</button>

<script>

// Your code here.

</script>

HINTS:

To solve the problem of having the changes conceptually happen at the same time, try to see the computation of a generation as a pure function, which takes one grid and produces a new grid that represents the next turn.

Representing the grid can be done in any of the ways shown in Chapters [7](http://eloquentjavascript.net/07_elife.html#grid) and [15](http://eloquentjavascript.net/15_game.html#level). Counting live neighbors can be done with two nested loops, looping over adjacent coordinates. Take care not to count cells outside of the field and to ignore the cell in the center, whose neighbors we are counting.

Making changes to checkboxes take effect on the next generation can be done in two ways. An event handler could notice these changes and update the current grid to reflect them, or you could generate a fresh grid from the values in the checkboxes before computing the next turn.

If you choose to go with event handlers, you might want to attach attributes that identify the position that each checkbox corresponds to so that it is easy to find out which cell to change.

To draw the grid of checkboxes, you either can use a <table> element (see[Chapter 13](http://eloquentjavascript.net/13_dom.html#exercise_table)) or simply put them all in the same element and put <br> (line break) elements between the rows.