Form properties and methods

Forms and control elements, such as `<input>` have a lot of special properties and events.
Working with forms will be much more convenient when we learn them.

Navigation form and elements

Document forms are members of the special collection `document.forms`.
That's a so-called *"named collection"*: it's both named and ordered. We can use both the name or the number in the document to get the form.

```
document.forms.my; // the form with name="my"
document.forms[0]; // the first form in the document
```

When we have a form, then any element is available in the named collection `form.elements`.
For instance:

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  // get the form
  let form = document.forms.my; // <form name="my"> element

  // get the element
  let elem = form.elements.one; // <input name="one"> element

  alert(elem.value); // 1
</script>
```

There may be multiple elements with the same name. This is typical with radio buttons and checkboxes.
In that case, `form.elements[name]` is a *collection*. For instance:

```
<form>
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>

<script>
let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0]); // [object HTMLInputElement]
</script>
```

These navigation properties do not depend on the tag structure. All control elements, no matter how deep they are in the form, are available in `form.elements`.

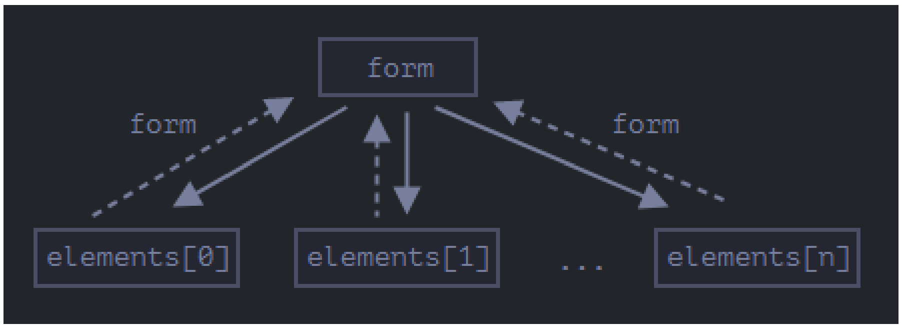
Fieldsets as "subforms"

A form may have one or many `<fieldset>` elements inside it. They also have `elements` property that lists form controls inside them.
For instance:

```
<body>
  <form id="form">
    <fieldset name="userFields">
      <legend>info</legend>
      <input name="login" type="text">
    </fieldset>
  </form>

  <script>
    alert(form.elements.login); // <input name="login">

    let fieldset = form.elements.userFields;
    alert(fieldset); // HTMLFieldSetElement

    // we can get the input by name both from the form and from the
fieldset
    alert(fieldset.elements.login == form.elements.login); // true
  </script>
</body>
```

Backreference element form

For any element, the form is available as element.form. So a form references all elements, and elements reference the form.
Here's the picture:



For instance:

```
<form id="form">
  <input type="text" name="login">
</form>

<script>
  // form -> element
  let login = form.login;

  // element -> form
  alert(login.form); // HTMLFormElement
</script>
```

Form elements

Let’s talk about form controls.

Input and textarea

We can access their value as input.value (string) or input.checked (boolean) for checkboxes and radio buttons.
Like this:
input.value = "New value";
textarea.value = "New text";

input.checked = true; // for a checkbox or radio button

Forms: event and method submit

The `submit` event triggers when the form is submitted, it is usually used to validate the form before sending it to the server or to abort the submission and process it in JavaScript.
The method `form.submit()` allows to initiate form sending from JavaScript. We can use it to dynamically create and send our own forms to server.
Let’s see more details of them.

Event: submit

There are two main ways to submit a form:

- The first – to click `<input type="submit">` or `<input type="image">`.
- The second – press `Enter` on an input field.

Both actions lead to `submit` event on the form. The handler can check the data, and if there are errors, show them and call `event.preventDefault()`, then the form won’t be sent to the server.

In the form below:
Go into the text field and press `Enter`.
Click `<input type="submit">`.

Both actions show `alert` and the form is not sent anywhere due to `return false`:

```
<form onsubmit="alert('submit!');return false">
  First: Enter in the input field <input type="text" value="text"><br>
  Second: Click "submit": <input type="submit" value="Submit">
</form>
```

Relation between `submit` and `click`

When a form is sent using `Enter` on an input field, a `click` event triggers on the `<input type="submit">`.
That’s rather funny, because there was no click at all.
Here’s the demo:

```
<form onsubmit="return false">
 <input type="text" size="30" value="Focus here and press enter">
 <input type="submit" value="Submit" onclick="alert('click')">
```

```
</form>
```

Method submit

To submit a form to the server manually, we can call `form.submit()`.
Then the `submit` event is not generated. It is assumed that if the programmer calls `form.submit()`, then the script already did all related processing.

Sometimes that's used to manually create and send a form, like this:

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// the form must be in the document to submit it
document.body.append(form);

form.submit();
```

Client-side form validation

Client-side validation is an initial check and an important feature of good user experience; by catching invalid data on the client-side, the user can fix it straight away. If it gets to the server and is then rejected, a noticeable delay is caused by a round trip to the server and then back to the client-side to tell the user to fix their data.
However, client-side validation should not be considered an exhaustive security measure! Your apps should always perform security checks on any form-submitted data on the as the client-side, because client-side validation is too easy to bypass, so malicious users can still easily send bad data through to your server. Read Website security for an idea of what happen; implementing server-side validation is somewhat beyond the scope of this module, but you should bear it in mind.

what_is_form_validation

Go to any popular site with a registration form, and you will notice that they provide feedback when you don't enter your data in the format they are expecting. You'll get messages such as:

- "This field is required" (You can't leave this field blank).
- "Please enter your phone number in the format xxx-xxxx" (A specific data format is required for it to be considered valid).
- "Please enter a valid email address" (the data you entered is not in the right format).
- "Your password needs to be between 8 and 30 characters long and contain one uppercase letter, one symbol, and a number." (A very specific data format is required for your data).

This is called **form validation**. When you enter data, the browser and/or the web server will check to see that the data is in the correct format and within the constraints set by the application. Validation done in the browser is called **client-side** validation, while validation done on the server is called **server-side** validation. In this chapter we are focusing on client-side validation.

If the information is correctly formatted, the application allows the data to be submitted to the server and (usually) saved in a database; if the information isn't correctly formatted, it gives the user an error message explaining what needs to be corrected, and lets them try again.

We want to make filling out web forms as easy as possible. So why do we insist on validating our forms? There are three main reasons:

- **We want to get the right data, in the right format.** Our applications won't work properly if our users' data is stored in the wrong format, is incorrect, or is omitted altogether.
- **We want to protect our users' data**. Forcing our users to enter secure passwords makes it easier to protect their account information.
- **We want to protect ourselves**. There are many ways that malicious users can misuse unprotected forms to damage the application.

  Warning: Never trust data passed to your server from the client. Even if your form is validating correctly and preventing malformed input on the client-side, a malicious user can still alter the network request.

Different types of client side validation

There are two different types of client-side validation that you'll encounter on the web:

- **Built-in form validation** uses HTML form validation features, which we've discussed in many places throughout this module. This validation generally doesn't require much JavaScript. Built-in form validation has better performance than JavaScript, but it is not as customizable as JavaScript validation.
- **JavaScript** validation is coded using JavaScript. This validation is completely customizable, but you need to create it all (or use a library).

Built in form validation examples

```
<form>
  <label for="choose">Would you prefer a banana or cherry?</label>
  <input id="choose" name="i-like" />
  <button>Submit</button>
</form>
```
Copy to ClipboardCopy to Clipboard

```
input:invalid {
  border: 2px dashed red;
}

input:valid {
  border: 2px solid black;
}
```
Copy to ClipboardCopy to Clipboard

To begin, make a copy of fruit-start.html in a new directory on your hard drive.

Add a required attribute to your input, as shown below.

```
<form>
  <label for="choose">Would you prefer a banana or cherry? (required)</label>
  <input id="choose" name="i-like" required />
  <button>Submit</button>
</form>
```

Note the CSS that is included in the example file:

```
input:invalid {
  border: 2px dashed red;
}

input:invalid:required {
  background-image: linear-gradient(to right, pink, lightgreen);
}

input:valid {
  border: 2px solid black;
}
```

This CSS causes the input to have a red dashed border when it is invalid and a more subtle solid black border when valid. We also added a background gradient when the input is required *and* invalid. Try out the new behavior in the example below:

Regexps are quite complex, and we don't intend to teach you them exhaustively in this article. Below are some examples to give you a basic idea of how they work.

- a — Matches one character that is a (not b, not aa, and so on).
- abc — Matches a, followed by b, followed by c.
- ab?c — Matches a, optionally followed by a single b, followed by c. (ac or abc)
- ab*c — Matches a, optionally followed by any number of bs, followed by c. (ac, abc, abbbbbc, and so on).
- a|b — Matches one character that is a or b.
- abc|xyz — Matches exactly abc or exactly xyz (but not abcxyz or a or y, and so on).

Now delete the contents of the <body> element, and replace it with the following:

```
<form>
  <div>
    <label for="choose">Would you prefer a banana or a cherry?</label>
    <input
      type="text"
      id="choose"
      name="i-like"
      required
      minlength="6"
      maxlength="6" />
  </div>
  <div>
    <label for="number">How many would you like?</label>
    <input type="number" id="number" name="amount" value="1" min="1" max="10" />
  </div>
  <div>
    <button>Submit</button>
  </div>
</form>
```
Copy to ClipboardCopy to Clipboard

- Here you'll see that we've given the text field a minlength and maxlength of six, which is the same length as banana and cherry.
- We've also given the number field a min of one and a max of ten. Entered numbers outside this range will show as invalid; users won't be able to use the increment/decrement arrows to move the value outside of this range. If the user manually enters a number outside of this range, the data is invalid. The number is not required, so removing the value will still result in a valid value..

Here is a full example to show usage of HTML's built-in validation features. First, some HTML:

```
<form>
  <p>
    <fieldset>
      <legend>Do you have a driver's license?<span aria-label="required">*</span></legend>
      <!-- While only one radio button in a same-named group can be selected at a time,
           and therefore only one radio button in a same-named group having the "required"
           attribute suffices in making a selection a requirement -->
      <input type="radio" required name="driver" id="r1" value="yes"><label for="r1">Yes</label>
      <input type="radio" required name="driver" id="r2" value="no"><label for="r2">No</label>
    </fieldset>
  </p>
  <p>
    <label for="n1">How old are you?</label>
    <!-- The pattern attribute can act as a fallback for browsers which
         don't implement the number input type but support the pattern attribute.
         Please note that browsers that support the pattern attribute will make it
         fail silently when used with a number field.
         Its usage here acts only as a fallback -->
    <input type="number" min="12" max="120" step="1" id="n1" name="age"
           pattern="\d+">
  </p>
  <p>
    <label for="t1">What's your favorite fruit?<span aria-label="required">*</span></label>
    <input type="text" id="t1" name="fruit" list="l1" required
           pattern="[Bb]anana|[Cc]herry|[Aa]pple|[Ss]trawberry|[Ll]emon|[Oo]range">
    <datalist id="l1">
      <option>Banana</option>
      <option>Cherry</option>
      <option>Apple</option>
      <option>Strawberry</option>
      <option>Lemon</option>
      <option>Orange</option>
    </datalist>
  </p>
  <p>
    <label for="t2">What's your email address?</label>
    <input type="email" id="t2" name="email">
  </p>
  <p>
```

```
    <label for="t3">Leave a short message</label>
    <textarea id="t3" name="msg" maxlength="140" rows="5"></textarea>
  </p>
  <p>
    <button>Submit</button>
  </p>
</form>
```
Copy to ClipboardCopy to Clipboard

And now some CSS to style the HTML:

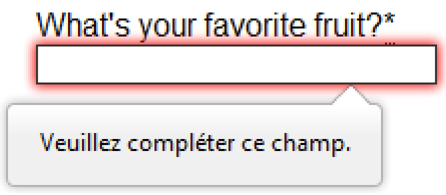
```
form {
  font: 1em sans-serif;
  max-width: 320px;
}

p > label {
  display: block;
}

input[type="text"],
input[type="email"],
input[type="number"],
textarea,
fieldset {
  width: 100%;
  border: 1px solid #333;
  box-sizing: border-box;
}

input:invalid {
  box-shadow: 0 0 5px 1px red;
}

input:focus:invalid {
  box-shadow: none;
}
```

Implementing a customized error message

As you saw in the HTML validation constraint examples earlier, each time a user tries to submit an invalid form, the browser displays an error message. The way this message is displayed depends on the browser.

These automated messages have two drawbacks:

- There is no standard way to change their look and feel with CSS.
- They depend on the browser locale, which means that you can have a page in one language but an error message displayed in another language, as seen in the following Firefox screenshot.



Customizing these error messages is one of the most common use cases of the Constraint Validation API. Let's work through a simple example of how to do this.

We'll start with some simple HTML (feel free to put this in a blank HTML file; use a fresh copy of as a basis, if you like):

```
<form>
  <label for="mail">
    I would like you to provide me with an email address:
  </label>
  <input type="email" id="mail" name="mail" />
  <button>Submit</button>
</form>
```

Copy to ClipboardCopy to Clipboard

And add the following JavaScript to the page:

```
const email = document.getElementById("mail");

email.addEventListener("input", (event) => {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting an email address!");
  } else {
    email.setCustomValidity("");
  }
});
```

Here we store a reference to the email input, then add an event listener to it that runs the contained code each time the value inside the input is changed.

Inside the contained code, we check whether the email input's `validity.typeMismatch` property returns `true`, meaning that the contained value doesn't match the pattern for a well-formed email address. If so, we call the method with a custom message. This renders the input invalid, so that when you try to submit the form, submission fails and the custom error message is displayed.

If the `validity.typeMismatch` property returns `false`, we call the `setCustomValidity()` method with an empty string. This renders the input valid, so the form will submit.

A more detailed example

Now that we've seen a really simple example, let's see how we can use this API to build some slightly more complex custom validation.

First, the HTML. Again, feel free to build this along with us:

```
<form novalidate>
  <p>
    <label for="mail">
      <span>Please enter an email address:</span>
      <input type="email" id="mail" name="mail" required minlength="8" />
      <span class="error" aria-live="polite"></span>
    </label>
  </p>
  <button>Submit</button>
</form>
```
Copy to ClipboardCopy to Clipboard

This simple form uses the novalidate attribute to turn off the browser's automatic validation; this lets our script take control over validation. However, this doesn't disable support for the constraint validation API nor the application of CSS pseudo-classes, etc. That means that even though the browser doesn't automatically check the validity of the form before sending its data, you can still do it yourself and style the form accordingly.

Our input to validate is an, which is `required`, and has a `minlength` of 8 characters. Let's check these using our own code, and show a custom error message for each one.

We are aiming to show the error messages inside a `<span>` element. The attribute is set on that `<span>` to make sure that our custom error message will be presented to everyone, including it being read out to screen reader users.

Note: A key point here is that setting the `novalidate` attribute on the form is what stops the form from showing its own error message bubbles, and allows us to instead display the custom error messages in the DOM in some manner of our own choosing.

Now onto some basic CSS to improve the look of the form slightly, and provide some visual feedback when the input data is invalid:

```
body {
  font: 1em sans-serif;
  width: 200px;
  padding: 0;
  margin: 0 auto;
}

p * {
  display: block;
}

input[type="email"] {
  appearance: none;

  width: 100%;
  border: 1px solid #333;
  margin: 0;

  font-family: inherit;
  font-size: 90%;

  box-sizing: border-box;
}
```

```
/* This is our style for the invalid fields */
input:invalid {
  border-color: #900;
  background-color: #fdd;
}

input:focus:invalid {
  outline: none;
}

/* This is the style of our error messages */
.error {
  width: 100%;
  padding: 0;

  font-size: 80%;
  color: white;
  background-color: #900;
  border-radius: 0 0 5px 5px;

  box-sizing: border-box;
}

.error.active {
  padding: 0.3em;
}
```

Now let's look at the JavaScript that implements the custom error validation.

```
// There are many ways to pick a DOM node; here we get the form itself and the email
// input box, as well as the span element into which we will place the error message.
const form = document.querySelector("form");
const email = document.getElementById("mail");
const emailError = document.querySelector("#mail + span.error");

email.addEventListener("input", (event) => {
  // Each time the user types something, we check if the
  // form fields are valid.

  if (email.validity.valid) {
    // In case there is an error message visible, if the field
    // is valid, we remove the error message.
    emailError.textContent = ""; // Reset the content of the message
    emailError.className = "error"; // Reset the visual state of the message
  } else {
    // If there is still an error, show the correct error
    showError();
  }
});

form.addEventListener("submit", (event) => {
  // if the email field is valid, we let the form submit
  if (!email.validity.valid) {
    // If it isn't, we display an appropriate error message
    showError();
    // Then we prevent the form from being sent by canceling the event
    event.preventDefault();
  }
});

function showError() {
  if (email.validity.valueMissing) {
    // If the field is empty,
    // display the following error message.
    emailError.textContent = "You need to enter an email address.";
  } else if (email.validity.typeMismatch) {
    // If the field doesn't contain an email address,
    // display the following error message.
    emailError.textContent = "Entered value needs to be an email address.";
  } else if (email.validity.tooShort) {
    // If the data is too short,
    // display the following error message.
    emailError.textContent = `Email should be at least ${email.minLength} characters; you entered
${email.value.length}.`;
  }

  // Set the styling appropriately
  emailError.className = "error active";
}
```
Copy to ClipboardCopy to Clipboard

The comments explain things pretty well, but briefly:

- Every time we change the value of the input, we check to see if it contains valid data. If it has then we remove any error message being shown. If the data is not valid, we run `showError()` to show the appropriate error.
- Every time we try to submit the form, we again check to see if the data is valid. If so, we let the form submit. If not, we run `showError()` to show the appropriate error, and stop the form submitting with [preventDefault()](#).
- The `showError()` function uses various properties of the input's `validity` object to determine what the error is, and then displays an error message as appropriate.

Here is the live result:

The constraint validation API gives you a powerful tool to handle form validation, letting you have enormous control over the user interface above and beyond what you can do with HTML and CSS alone.

In some cases, such as how to build custom form controls, you won't be able to or won't want to use the Constraint Validation API. You're still able to use JavaScript to validate your form, but you'll just have to write your own.

To validate a form, ask yourself a few questions:

What kind of validation should I perform?

> You need to determine how to validate your data: string operations, type conversion, regular expressions, and so on. It's up to you.

What should I do if the form doesn't validate?

> This is clearly a UI matter. You have to decide how the form will behave. Does the form send the data anyway? Should you highlight the fields that are in error? Should you display error messages?

How can I help the user to correct invalid data?

> In order to reduce the user's frustration, it's very important to provide as much helpful information as possible in order to guide them in correcting their inputs. You should offer up-front suggestions so they know what's expected, as well as clear error messages.

An example that doesn't use the constraint validation API

In order to illustrate this, the following is a simplified version of the previous example without the Constraint Validation API.

The HTML is almost the same; we just removed the HTML validation features.

```
<form>
  <p>
    <label for="mail">
      <span>Please enter an email address:</span>
      <input type="text" id="mail" name="mail" />
      <span class="error" aria-live="polite"></span>
    </label>
  </p>
  <button>Submit</button>
</form>
```
Copy to ClipboardCopy to Clipboard

Similarly, the CSS doesn't need to change very much; we've just turned the :invalid CSS pseudo-class into a real class and avoided using the attribute selector.

```
body {
  font: 1em sans-serif;
  width: 200px;
  padding: 0;
  margin: 0 auto;
}

form {
  max-width: 200px;
}

p * {
  display: block;
}

input.mail {
  appearance: none;
  width: 100%;
  border: 1px solid #333;
  margin: 0;

  font-family: inherit;
  font-size: 90%;

  box-sizing: border-box;
}

/* This is our style for the invalid fields */
input.invalid {
  border-color: #900;
  background-color: #fdd;
}

input:focus.invalid {
  outline: none;
}

/* This is the style of our error messages */
.error {
  width: 100%;
  padding: 0;

  font-size: 80%;
  color: white;
  background-color: #900;
  border-radius: 0 0 5px 5px;
  box-sizing: border-box;
}

.error.active {
  padding: 0.3em;
}
```
Copy to ClipboardCopy to Clipboard

The big changes are in the JavaScript code, which needs to do much more heavy lifting.

```
const form = document.querySelector("form");
const email = document.getElementById("mail");
const error = email.nextElementSibling;

// As per the HTML Specification
const emailRegExp =
  /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/;

// Now we can rebuild our validation constraint
// Because we do not rely on CSS pseudo-class, we have to
// explicitly set the valid/invalid class on our email field
window.addEventListener("load", () => {
  // Here, we test if the field is empty (remember, the field is not required)
  // If it is not, we check if its content is a well-formed email address.
  const isValid = email.value.length === 0 || emailRegExp.test(email.value);
  email.className = isValid ? "valid" : "invalid";
});

// This defines what happens when the user types in the field
email.addEventListener("input", () => {
  const isValid = email.value.length === 0 || emailRegExp.test(email.value);
  if (isValid) {
    email.className = "valid";
    error.textContent = "";
    error.className = "error";
  } else {
    email.className = "invalid";
  }
});

// This defines what happens when the user tries to submit the data
form.addEventListener("submit", (event) => {
  event.preventDefault();

  const isValid = email.value.length === 0 || emailRegExp.test(email.value);
  if (!isValid) {
    email.className = "invalid";
    error.textContent = "I expect an email, darling!";
    error.className = "error active";
  } else {
    email.className = "valid";
    error.textContent = "";
    error.className = "error";
  }
});
```

LEARN MORE

Reading

1. Read 4.1 Form properties and methods
2. Read 4.4 Forms:Event and method submit
3. Watch Using FormData Objects Effectively (13min) OR read 3.2 FormData
4. Read MDN: Client-Side Form Validation