📅 26th December 2019

Custom elements

We can create custom HTML elements, described by our class, with its own methods and properties, events and so on.

Once a custom element is defined, we can use it on par with built-in HTML elements.

That's great, as HTML dictionary is rich, but not infinite. There are no `<easy-tabs>`, `<sliding-carousel>`, `<beautiful-upload>` … Just think of any other tag we might need.

We can define them with a special class, and then use as if they were always a part of HTML.

There are two kinds of custom elements:

1. **Autonomous custom elements** – "all-new" elements, extending the abstract `HTMLElement` class.
2. **Customized built-in elements** – extending built-in elements, like a customized button, based on `HTMLButtonElement` etc.

First we'll cover autonomous elements, and then move to customized built-in ones.

To create a custom element, we need to tell the browser several details about it: how to show it, what to do when the element is added or removed to page, etc.

That's done by making a class with special methods. That's easy, as there are only few methods, and all of them are optional.

Here's a sketch with the full list:

```
1   class MyElement extends HTMLElement {
2     constructor() {
3       super();
4       // element created
5     }
6
7     connectedCallback() {
8       // browser calls this method when the element is added to the document
9       // (can be called many times if an element is repeatedly added/removed)
10    }
11
12    disconnectedCallback() {
13      // browser calls this method when the element is removed from the documen
14      // (can be called many times if an element is repeatedly added/removed)
15    }
16
17    static get observedAttributes() {
18      return [/* array of attribute names to monitor for changes */];
19    }
20
```

```
21    attributeChangedCallback(name, oldValue, newValue) {
22      // called when one of attributes listed above is modified
23    }
24
25    adoptedCallback() {
26      // called when the element is moved to a new document
27      // (happens in document.adoptNode, very rarely used)
28    }
29
30    // there can be other element methods and properties
31  }
```

After that, we need to register the element:

```
1  // let the browser know that <my-element> is served by our new class
2  customElements.define("my-element", MyElement);
```

Now for any HTML elements with tag `<my-element>`, an instance of `MyElement` is created, and the aforementioned methods are called. We also can `document.createElement('my-element')` in JavaScript.

> ℹ️ **Custom element name must contain a hyphen -**
>
> Custom element name must have a hyphen `-`, e.g. `my-element` and `super-button` are valid names, but `myelement` is not.
>
> That's to ensure that there are no name conflicts between built-in and custom HTML elements.

Example: "time-formatted"

For example, there already exists `<time>` element in HTML, for date/time. But it doesn't do any formatting by itself.

Let's create `<time-formatted>` element that displays the time in a nice, language-aware format:

```
1  <script>
2  class TimeFormatted extends HTMLElement { // (1)
3
4    connectedCallback() {
5      let date = new Date(this.getAttribute('datetime') || Date.now());
6
7      this.innerHTML = new Intl.DateTimeFormat("default", {
8        year: this.getAttribute('year') || undefined,
9        month: this.getAttribute('month') || undefined,
10       day: this.getAttribute('day') || undefined,
11       hour: this.getAttribute('hour') || undefined,
12       minute: this.getAttribute('minute') || undefined,
13       second: this.getAttribute('second') || undefined,
14       timeZoneName: this.getAttribute('time-zone-name') || undefined,
15     }).format(date);
16   }
17
```

```
18  }
19
20  customElements.define("time-formatted", TimeFormatted); // (2)
21  </script>
22
23  <!-- (3) -->
24  <time-formatted datetime="2019-12-01"
25    year="numeric" month="long" day="numeric"
26    hour="numeric" minute="numeric" second="numeric"
27    time-zone-name="short"
28  ></time-formatted>
```

---

1 December 2019, 5:30:00 GMT+5:30

---

1. The class has only one method `connectedCallback()` – the browser calls it when `<time-formatted>` element is added to page (or when HTML parser detects it), and it uses the built-in Intl.DateTimeFormat data formatter, well-supported across the browsers, to show a nicely formatted time.

2. We need to register our new element by `customElements.define(tag, class)`.

3. And then we can use it everywhere.

ℹ️ **Custom elements upgrade**

If the browser encounters any `<time-formatted>` elements before `customElements.define`, that's not an error. But the element is yet unknown, just like any non-standard tag.

Such "undefined" elements can be styled with CSS selector `:not(:defined)`.

When `customElement.define` is called, they are "upgraded": a new instance of `TimeFormatted` is created for each, and `connectedCallback` is called. They become `:defined`.

To get the information about custom elements, there are methods:

- `customElements.get(name)` – returns the class for a custom element with the given `name`,

- `customElements.whenDefined(name)` – returns a promise that resolves (without value) when a custom element with the given `name` becomes defined.

ℹ️ **Rendering in `connectedCallback`, not in `constructor`**

In the example above, element content is rendered (created) in `connectedCallback`.

Why not in the `constructor`?

The reason is simple: when `constructor` is called, it's yet too early. The element is created, but the browser did not yet process/assign attributes at this stage: calls to `getAttribute` would return `null`. So we can't really render there.

Besides, if you think about it, that's better performance-wise – to delay the work until it's really needed.

The `connectedCallback` triggers when the element is added to the document. Not just appended to another element as a child, but actually becomes a part of the page. So we can build detached DOM, create elements and prepare them for later use. They will only be actually rendered when they make it into the page.

Observing attributes

In the current implementation of `<time-formatted>`, after the element is rendered, further attribute changes don't have any effect. That's strange for an HTML element. Usually, when we change an attribute, like `a.href`, we expect the change to be immediately visible. So let's fix this.

We can observe attributes by providing their list in `observedAttributes()` static getter. For such attributes, `attributeChangedCallback` is called when they are modified. It doesn't trigger for an attribute for performance reasons.

Here's a new `<time-formatted>`, that auto-updates when attributes change:

```
1  <script>
2  class TimeFormatted extends HTMLElement {
3
4    render() { // (1)
5      let date = new Date(this.getAttribute('datetime') || Date.now());
6
7      this.innerHTML = new Intl.DateTimeFormat("default", {
8        year: this.getAttribute('year') || undefined,
9        month: this.getAttribute('month') || undefined,
10       day: this.getAttribute('day') || undefined,
11       hour: this.getAttribute('hour') || undefined,
12       minute: this.getAttribute('minute') || undefined,
13       second: this.getAttribute('second') || undefined,
14       timeZoneName: this.getAttribute('time-zone-name') || undefined,
15     }).format(date);
16   }
17
18   connectedCallback() { // (2)
19     if (!this.rendered) {
20       this.render();
21       this.rendered = true;
22     }
23   }
24
25   static get observedAttributes() { // (3)
26     return ['datetime', 'year', 'month', 'day', 'hour', 'minute', 'second', '
27   }
28
29   attributeChangedCallback(name, oldValue, newValue) { // (4)
30     this.render();
31   }
32
33 }
34
35 customElements.define("time-formatted", TimeFormatted);
36 </script>
37
38 <time-formatted id="elem" hour="numeric" minute="numeric" second="numeric"></
39
40 <script>
41 setInterval(() => elem.setAttribute('datetime', new Date()), 1000); // (5)
42 </script>
```

```
13:27:01
```

1. The rendering logic is moved to `render()` helper method.
2. We call it once when the element is inserted into page.
3. For a change of an attribute, listed in `observedAttributes()`, `attributeChangedCallback` triggers.
4. …and re-renders the element.
5. At the end, we can easily make a live timer.

Rendering order

When HTML parser builds the DOM, elements are processed one after another, parents before children. E.g. if we have `<outer><inner></inner></outer>`, then `<outer>` element is created and connected to DOM first, and then `<inner>`.

That leads to important consequences for custom elements.

For example, if a custom element tries to access `innerHTML` in `connectedCallback`, it gets nothing:

```
1  <script>
2  customElements.define('user-info', class extends HTMLElement {
3
4    connectedCallback() {
5      alert(this.innerHTML); // empty (*)
6    }
7
8  });
9  </script>
10
11  <user-info>John</user-info>
```

If you run it, the `alert` is empty.

That's exactly because there are no children on that stage, the DOM is unfinished. HTML parser connected the custom element `<user-info>`, and is going to proceed to its children, but just didn't yet.

If we'd like to pass information to custom element, we can use attributes. They are available immediately.

Or, if we really need the children, we can defer access to them with zero-delay `setTimeout`.

This works:

```
1  <script>
2  customElements.define('user-info', class extends HTMLElement {
3
4    connectedCallback() {
5      setTimeout(() => alert(this.innerHTML)); // John (*)
6    }
7
8  });
9  </script>
10
11  <user-info>John</user-info>
```

Now the `alert` in line `(*)` shows "John", as we run it asynchronously, after the HTML parsing is complete. We can process children if needed and finish the initialization.

On the other hand, this solution is also not perfect. If nested custom elements also use `setTimeout` to initialize themselves, then they queue up: the outer `setTimeout` triggers first, and then the inner one.

So the outer element finishes the initialization before the inner one.

Let's demonstrate that on example:

```
1  <script>
2  customElements.define('user-info', class extends HTMLElement {
3    connectedCallback() {
4      alert(`${this.id} connected.`);
5      setTimeout(() => alert(`${this.id} initialized.`));
6    }
7  });
8  </script>
9
10 <user-info id="outer">
11   <user-info id="inner"></user-info>
12 </user-info>
```

Output order:

1. outer connected.
2. inner connected.
3. outer initialized.
4. inner initialized.

We can clearly see that the outer element finishes initialization `(3)` before the inner one `(4)`.

There's no built-in callback that triggers after nested elements are ready. If needed, we can implement such thing on our own. For instance, inner elements can dispatch events like `initialized`, and outer ones can listen and react on them.

Customized built-in elements

New elements that we create, such as `<time-formatted>`, don't have any associated semantics. They are unknown to search engines, and accessibility devices can't handle them.

But such things can be important. E.g, a search engine would be interested to know that we actually show a time. And if we're making a special kind of button, why not reuse the existing `<button>` functionality?

We can extend and customize built-in HTML elements by inheriting from their classes.

For example, buttons are instances of `HTMLButtonElement`, let's build upon it.

1. Extend `HTMLButtonElement` with our class:

```
1  class HelloButton extends HTMLButtonElement { /* custom element methods */
```

2. Provide an third argument to `customElements.define` , that specifies the tag:

```
1  customElements.define('hello-button', HelloButton, {extends: 'button'});
```

There may be different tags that share the same DOM-class, that's why specifying `extends` is needed.

3. At the end, to use our custom element, insert a regular `<button>` tag, but add `is="hello-button"` to it:

```
1  <button is="hello-button">...</button>
```

Here's a full example:

```
1  <script>
2  // The button that says "hello" on click
3  class HelloButton extends HTMLButtonElement {
4    constructor() {
5      super();
6      this.addEventListener('click', () => alert("Hello!"));
7    }
8  }
9
10 customElements.define('hello-button', HelloButton, {extends: 'button'});
11 </script>
12
13 <button is="hello-button">Click me</button>
14
15 <button is="hello-button" disabled>Disabled</button>
```

Click me   Disabled

Our new button extends the built-in one. So it keeps the same styles and standard features like `disabled` attribute.

References

- HTML Living Standard: https://html.spec.whatwg.org/#custom-elements.
- Compatiblity: https://caniuse.com/#feat=custom-elements.

Summary

Custom elements can be of two types:

1. "Autonomous" – new tags, extending `HTMLElement` .

   Definition scheme:

```
1  class MyElement extends HTMLElement {
2    constructor() { super(); /* ... */ }
```

```
 3    connectedCallback() { /* ... */ }
 4    disconnectedCallback() { /* ... */  }
 5    static get observedAttributes() { return [/* ... */]; }
 6    attributeChangedCallback(name, oldValue, newValue) { /* ... */ }
 7    adoptedCallback() { /* ... */ }
 8    }
 9  customElements.define('my-element', MyElement);
10  /* <my-element> */
```

2. "Customized built-in elements" – extensions of existing elements.

   Requires one more `.define` argument, and `is="..."` in HTML:

```
1  class MyButton extends HTMLButtonElement { /*...*/ }
2  customElements.define('my-button', MyElement, {extends: 'button'});
3  /* <button is="my-button"> */
```

Custom elements are well-supported among browsers. Edge is a bit behind, but there's a polyfill https://github.com/webcomponents/polyfills/tree/master/packages/webcomponentsjs.

✅ Tasks

Live timer element 🔗

We already have `<time-formatted>` element to show a nicely formatted time.

Create `<live-timer>` element to show the current time:

1. It should use `<time-formatted>` internally, not duplicate its functionality.
2. Ticks (updates) every second.
3. For every tick, a custom event named `tick` should be generated, with the current date in `event.detail` (see chapter Dispatching custom events).

Usage:

```
1  <live-timer id="elem"></live-timer>
2
3  <script>
4    elem.addEventListener('tick', event => console.log(event.detail));
5  </script>
```

Demo:

> 13:27:01

Open a sandbox for the task.

solution

Previous lesson	Next lesson

Share 🐦 f

🖧 Tutorial map

💬 Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)

© 2007—2020 Ilya Kantorabout the projectcontact usterms of usage
privacy policy