8th February 2020

Introduction to browser events

An event is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).

Here's a list of the most useful DOM events, just to take a look at:

Mouse events:

- `click` – when the mouse clicks on an element (touchscreen devices generate it on a tap).
- `contextmenu` – when the mouse right-clicks on an element.
- `mouseover` / `mouseout` – when the mouse cursor comes over / leaves an element.
- `mousedown` / `mouseup` – when the mouse button is pressed / released over an element.
- `mousemove` – when the mouse is moved.

Form element events:

- `submit` – when the visitor submits a `<form>`.
- `focus` – when the visitor focuses on an element, e.g. on an `<input>`.

Keyboard events:

- `keydown` and `keyup` – when the visitor presses and then releases the button.

Document events:

- `DOMContentLoaded` – when the HTML is loaded and processed, DOM is fully built.

CSS events:

- `transitionend` – when a CSS-animation finishes.

There are many other events. We'll get into more details of particular events in next chapters.

Event handlers

To react on events we can assign a *handler* – a function that runs in case of an event.

Handlers are a way to run JavaScript code in case of user actions.

There are several ways to assign a handler. Let's see them, starting from the simplest one.

HTML-attribute

A handler can be set in HTML with an attribute named `on<event>`.

For instance, to assign a `click` handler for an `input`, we can use `onclick`, like here:

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

On mouse click, the code inside `onclick` runs.

Please note that inside `onclick` we use single quotes, because the attribute itself is in double quotes. If we forget that the code is inside the attribute and use double quotes inside, like this: `onclick="alert("Click!")"`, then it won't work right.

An HTML-attribute is not a convenient place to write a lot of code, so we'd better create a JavaScript function and call it there.

Here a click runs the function `countRabbits()`:

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Rabbit number " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">
```

Count rabbits!

As we know, HTML attribute names are not case-sensitive, so `ONCLICK` works as well as `onClick` and `onCLICK` … But usually attributes are lowercased: `onclick`.

DOM property

We can assign a handler using a DOM property `on<event>`.

For instance, `elem.onclick`:

```
<input id="elem" type="button" value="Click me">
<script>
  elem.onclick = function() {
    alert('Thank you');
  };
</script>
```

Click me

If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM property.

So this way is actually the same as the previous one.

The handler is always in the DOM property: the HTML-attribute is just one of the ways to initialize it.

These two code pieces work the same:

1. Only HTML:

```
<input type="button" onclick="alert('Click!')" value="Button">
```

Button

2. HTML + JS:

```
<input type="button" id="button" value="Button">
<script>
  button.onclick = function() {
    alert('Click!');
  };
</script>
```

Button

As there's only one `onclick` property, we can't assign more than one event handler.

In the example below adding a handler with JavaScript overwrites the existing handler:

```
<input type="button" id="elem" onclick="alert('Before')" value="Click me">
<script>
  elem.onclick = function() { // overwrites the existing handler
    alert('After'); // only this will be shown
  };
</script>
```

Click me

By the way, we can assign an existing function as a handler directly:

```
function sayThanks() {
  alert('Thanks!');
}

elem.onclick = sayThanks;
```

To remove a handler – assign `elem.onclick = null`.

Accessing the element: this

The value of `this` inside a handler is the element. The one which has the handler on it.

In the code below `button` shows its contents using `this.innerHTML` :

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

Click me

Possible mistakes

If you're starting to work with events – please note some subtleties.

The function should be assigned as `sayThanks` , not `sayThanks()` .

```
// right
button.onclick = sayThanks;

// wrong
button.onclick = sayThanks();
```

If we add parentheses, `sayThanks()` – is a function call. So the last line actually takes the *result* of the function execution, that is `undefined` (as the function returns nothing), and assigns it to `onclick` . That doesn't work.

…On the other hand, in the markup we do need the parentheses:

```
<input type="button" id="button" onclick="sayThanks()">
```

The difference is easy to explain. When the browser reads the attribute, it creates a handler function with *body from its content*: `sayThanks()` .

So the markup generates this property:

```
button.onclick = function() {
  sayThanks(); // the attribute content
};
```

Use functions, not strings.

The assignment `elem.onclick = "alert(1)"` would work too. It works for compatibility reasons, but is strongly not recommended.

Don't use `setAttribute` for handlers.

Such a call won't work:

```
// a click on <body> will generate errors,
// because attributes are always strings, function becomes a string
document.body.setAttribute('onclick', function() { alert(1) });
```

DOM-property case matters.

Assign a handler to `elem.onclick`, not `elem.ONCLICK`, because DOM properties are case-sensitive.

addEventListener

The fundamental problem of the aforementioned ways to assign handlers – we can't assign multiple handlers to one event.

For instance, one part of our code wants to highlight a button on click, and another one wants to show a message.

We'd like to assign two event handlers for that. But a new DOM property will overwrite the existing one:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // replaces the previous handler
```

Web-standard developers understood that long ago and suggested an alternative way of managing handlers using special methods `addEventListener` and `removeEventListener`. They are free of such a problem.

The syntax to add a handler:

```
element.addEventListener(event, handler[, options]);
```

`event`

Event name, e.g. `"click"`.

`handler`

The handler function.

`options`

An additional optional object with properties:

- `once`: if `true`, then the listener is automatically removed after it triggers.
- `capture`: the phase where to handle the event, to be covered later in the chapter Bubbling and capturing. For historical reasons, `options` can also be `false/true`, that's the same as `{capture: false/true}`.
- `passive`: if `true`, then the handler will not `preventDefault()`, we'll cover that later in Browser default actions.

To remove the handler, use `removeEventListener`:

```
element.removeEventListener(event, handler[, options]);
```

Removal requires the same function

To remove a handler we should pass exactly the same function as was assigned.

That doesn't work:

```
elem.addEventListener( "click" , () => alert('Thanks!'));
// ....
elem.removeEventListener( "click", () => alert('Thanks!'));
```

The handler won't be removed, because `removeEventListener` gets another function – with the same code, but that doesn't matter.

Here's the right way:

```
function handler() {
  alert( 'Thanks!' );
}

input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

Please note – if we don't store the function in a variable, then we can't remove it. There's no way to "read back" handlers assigned by `addEventListener`.

Multiple calls to `addEventListener` allow to add multiple handlers, like this:

```
<input id="elem" type="button" value="Click me"/>

<script>
  function handler1() {
    alert('Thanks!');
  };

  function handler2() {
    alert('Thanks again!');
  }

  elem.onclick = () => alert("Hello");
  elem.addEventListener("click", handler1); // Thanks!
  elem.addEventListener("click", handler2); // Thanks again!
</script>
```

As we can see in the example above, we can set handlers *both* using a DOM-property and `addEventListener`. But generally we use only one of these ways.

For some events, handlers only work with `addEventListener`

There exist events that can't be assigned via a DOM-property. Must use `addEventListener`.

For instance, the event `DOMContentLoaded`, that triggers when the document is loaded and DOM is built.

```
document.onDOMContentLoaded = function() {
  alert("DOM built"); // will never run
};
```

```
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM built"); // this way it works
});
```

So `addEventListener` is more universal. Although, such events are an exception rather than the rule.

Event object

To properly handle an event we'd want to know more about what's happened. Not just a "click" or a "keypress", but what were the pointer coordinates? Which key was pressed? And so on.

When an event happens, the browser creates an *event object*, puts details into it and passes it as an argument to the handler.

Here's an example of getting mouse coordinates from the event object:

```
<input type="button" value="Click me" id="elem">

<script>
  elem.onclick = function(event) {
    // show event type, element and coordinates of the click
    alert(event.type + " at " + event.currentTarget);
    alert("Coordinates: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Some properties of `event` object:

`event.type`

Event type, here it's `"click"`.

`event.currentTarget`

Element that handled the event. That's exactly the same as `this`, unless the handler is an arrow function, or its `this` is bound to something else, then we can get the element from `event.currentTarget`.

`event.clientX / event.clientY`

Window-relative coordinates of the cursor, for mouse events.

There are more properties. They depend on the event type, so we'll study them later when we come to different events in details.

The event object is also accessible from HTML

If we assign a handler in HTML, we can also use the `event` object, like this:

```
<input type="button" onclick="alert(event.type)" value="Event type">
```

Event type

That's possible because when the browser reads the attribute, it creates a handler like this: `function(event) { alert(event.type) }`. That is: its first argument is called `"event"`, and the body is taken from the attribute.

Object handlers: handleEvent

We can assign not just a function, but an object as an event handler using `addEventListener`. When an event occurs, its `handleEvent` method is called.

For instance:

```
<button id="elem">Click me</button>

<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " at " + event.currentTarget);
    }
  });
</script>
```

As we can see, when `addEventListener` receives an object as the handler, it calls `object.handleEvent(event)` in case of an event.

We could also use a class for that:

```
<button id="elem">Click me</button>

<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "Mouse button pressed";
          break;
        case 'mouseup':
          elem.innerHTML += "...and released.";
          break;
      }
```

```
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Here the same object handles both events. Please note that we need to explicitly setup the events to listen using `addEventListener`. The `menu` object only gets `mousedown` and `mouseup` here, not any other types of events.

The method `handleEvent` does not have to do all the job by itself. It can call other event-specific methods instead, like this:

```
<button id="elem">Click me</button>

<script>
  class Menu {
    handleEvent(event) {
      // mousedown -> onMousedown
      let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
      this[method](event);
    }

    onMousedown() {
      elem.innerHTML = "Mouse button pressed";
    }

    onMouseup() {
      elem.innerHTML += "...and released.";
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Now event handlers are clearly separated, that may be easier to support.

Summary

There are 3 ways to assign event handlers:

1. HTML attribute: `onclick="..."`.
2. DOM property: `elem.onclick = function`.
3. Methods: `elem.addEventListener(event, handler[, phase])` to add, `removeEventListener` to remove.

HTML attributes are used sparingly, because JavaScript in the middle of an HTML tag looks a little bit odd and alien. Also can't write lots of code in there.

DOM properties are ok to use, but we can't assign more than one handler of the particular event. In many cases that limitation is not pressing.

The last way is the most flexible, but it is also the longest to write. There are few events that only work with it, for instance `transtionend` and `DOMContentLoaded` (to be covered). Also `addEventListener` supports objects as event handlers. In that case the method `handleEvent` is called in case of the event.

No matter how you assign the handler – it gets an event object as the first argument. That object contains the details about what's happened.

We'll learn more about events in general and about different types of events in the next chapters.

Tasks

Hide on click

importance: 5

Add JavaScript to the `button` to make `<div id="text">` disappear when we click it.

The demo:

Click to hide the text

Text

Open a sandbox for the task.

solution

Hide self

importance: 5

Create a button that hides itself on click.

Like this: Click to hide

solution

Which handlers run?

importance: 5

There's a button in the variable. There are no handlers on it.

Which handlers run on click after the following code? Which alerts show up?

```
button.addEventListener("click", () => alert("1"));

button.removeEventListener("click", () => alert("1"));

button.onclick = () => alert(2);
```

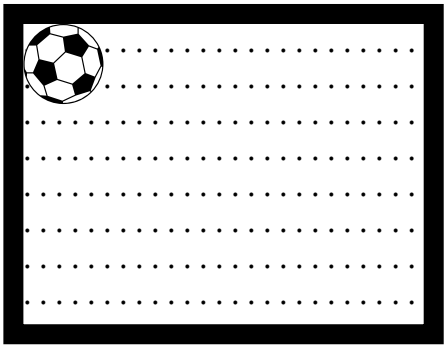
solution

Move the ball across the field

importance: 5

Move the ball across the field to a click. Like this:

Click on a field to move the ball there.



Requirements:

- The ball center should come exactly under the pointer on click (if possible without crossing the field edge).
- CSS-animation is welcome.
- The ball must not cross field boundaries.
- When the page is scrolled, nothing should break.

Notes:

- The code should also work with different ball and field sizes, not be bound to any fixed values.
- Use properties `event.clientX/event.clientY` for click coordinates.

Open a sandbox for the task.

solution

Create a sliding menu

importance: 5

Create a menu that opens/collapses on click:

▶ Sweeties (click me)!

P.S. HTML/CSS of the source document is to be modified.

Open a sandbox for the task.

solution

Add a closing button

importance: 5

There's a list of messages.

Use JavaScript to add a closing button to the right-upper corner of each message.

The result should look like this:

Horse [x]

The horse is one of two extant subspecies of Equus ferus. It is an odd-toed ungulate mammal belonging to the taxonomic family Equidae. The horse has evolved over the past 45 to 55 million years from a small multi-toed creature, Eohippus, into the large, single-toed animal of today.

Donkey [x]

The donkey or ass (Equus africanus asinus) is a domesticated member of the horse family, Equidae. The wild ancestor of the donkey is the African wild ass, E. africanus. The donkey has been used as a working animal for at least 5000 years.

Cat [x]

The domestic cat (Latin: Felis catus) is a small, typically furry, carnivorous mammal. They are often called house cats when kept as indoor pets or simply cats when there is no need to distinguish them from other felids and felines. Cats are often valued by humans for companionship and for their ability to hunt vermin.

Open a sandbox for the task.

solution

Carousel

importance: 4

Create a “carousel” – a ribbon of images that can be scrolled by clicking on arrows.

Later we can add more features to it: infinite scrolling, dynamic loading etc.

P.S. For this task HTML/CSS structure is actually 90% of the solution.

Open a sandbox for the task.

solution

Previous lesson Next lesson

Share

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…)

privacy policy