26th November 2019

Browser default actions

Many events automatically lead to certain actions performed by the browser.

For instance:

- A click on a link – initiates navigation to its URL.
- A click on a form submit button – initiates its submission to the server.
- Pressing a mouse button over a text and moving it – selects the text.

If we handle an event in JavaScript, we may not want the corresponding browser action to happen, and want to implement another behavior instead.

Preventing browser actions

There are two ways to tell the browser we don't want it to act:

- The main way is to use the `event` object. There's a method `event.preventDefault()`.
- If the handler is assigned using `on<event>` (not by `addEventListener`), then returning `false` also works the same.

In this HTML a click on a link doesn't lead to navigation, browser doesn't do anything:

```
1 <a href="/" onclick="return false">Click here</a>
2 or
3 <a href="/" onclick="event.preventDefault()">here</a>
```

Click here or here

In the next example we'll use this technique to create a JavaScript-powered menu.

Returning `false` from a handler is an exception

The value returned by an event handler is usually ignored.

The only exception is `return false` from a handler assigned using `on<event>`.

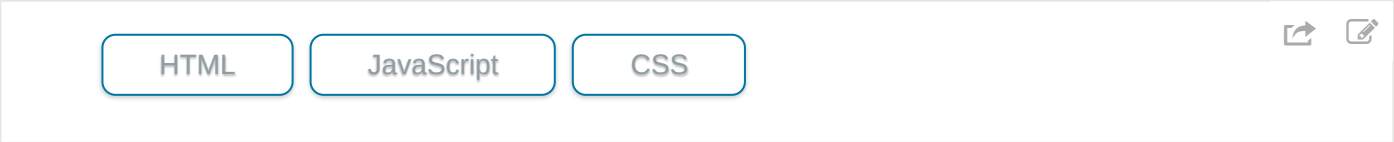
In all other cases, `return` value is ignored. In particular, there's no sense in returning `true`.

Example: the menu

Consider a site menu, like this:

```
<ul id="menu" class="menu">
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/css">CSS</a></li>
</ul>
```

Here's how it looks with some CSS:



Menu items are implemented as HTML-links `<a>`, not buttons `<button>`. There are several reasons to do so, for instance:

- Many people like to use "right click" – "open in a new window". If we use `<button>` or `<span>`, that doesn't work.
- Search engines follow `<a href="...">` links while indexing.

So we use `<a>` in the markup. But normally we intend to handle clicks in JavaScript. So we should prevent the default browser action.

Like here:

```
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  let href = event.target.getAttribute('href');
  alert( href ); // ...can be loading from the server, UI generation etc

  return false; // prevent browser action (don't go to the URL)
};
```

If we omit `return false`, then after our code executes the browser will do its "default action" – navigating to the URL in `href`. And we don't need that here, as we're handling the click by ourselves.

By the way, using event delegation here makes our menu very flexible. We can add nested lists and style them using CSS to "slide down".

Follow-up events

Certain events flow one into another. If we prevent the first event, there will be no second.

For instance, `mousedown` on an `<input>` field leads to focusing in it, and the `focus` event. If we prevent the `mousedown` event, there's no focus.

Try to click on the first `<input>` below – the `focus` event happens. But if you click the second one, there's no focus.

```
<input value="Focus works" onfocus="this.value=''">
<input onmousedown="return false" onfocus="this.value=''" value="Click me'
```

Focus works Click me

That's because the browser action is canceled on `mousedown` . The focusing is still possible if we use another way to enter the input. For instance, the `Tab` key to switch from the 1st input into the 2nd. But not with the mouse click any more.

The "passive" handler option

The optional `passive: true` option of `addEventListener` signals the browser that the handler is not going to call `preventDefault()` .

Why that may be needed?

There are some events like `touchmove` on mobile devices (when the user moves their finger across the screen), that cause scrolling by default, but that scrolling can be prevented using `preventDefault()` in the handler.

So when the browser detects such event, it has first to process all handlers, and then if `preventDefault` is not called anywhere, it can proceed with scrolling. That may cause unnecessary delays and "jitters" in the UI.

The `passive: true` options tells the browser that the handler is not going to cancel scrolling. Then browser scrolls immediately providing a maximally fluent experience, and the event is handled by the way.

For some browsers (Firefox, Chrome), `passive` is `true` by default for `touchstart` and `touchmove` events.

event.defaultPrevented

The property `event.defaultPrevented` is `true` if the default action was prevented, and `false` otherwise.

There's an interesting use case for it.

You remember in the chapter Bubbling and capturing we talked about `event.stopPropagation()` and why stopping bubbling is bad?

Sometimes we can use `event.defaultPrevented` instead, to signal other event handlers that the event was handled.

Let's see a practical example.

By default the browser on `contextmenu` event (right mouse click) shows a context menu with standard options. We can prevent it and show our own, like this:

```
<button>Right-click shows browser context menu</button>

<button oncontextmenu="alert('Draw our menu'); return false">
  Right-click shows our context menu
</button>
```

Right-click shows browser context menu | Right-click shows our context menu

Now, in addition to that context menu we'd like to implement document-wide context menu.

Upon right click, the closest context menu should show up.

```
<p>Right-click here for the document context menu</p>
<button id="elem">Right-click here for the button context menu</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Button context menu");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Document context menu");
  };
</script>
```

Right-click here for the document context menu

Right-click here for the button context menu

The problem is that when we click on `elem`, we get two menus: the button-level and (the event bubbles up) the document-level menu.

How to fix it? One of solutions is to think like: "When we handle right-click in the button handler, let's stop its bubbling" and use `event.stopPropagation()`:

```
<p>Right-click for the document menu</p>
<button id="elem">Right-click for the button menu (fixed with event.stopPropa

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    event.stopPropagation();
    alert("Button context menu");
  };

  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Document context menu");
```

```
  };
</script>
```

Right-click for the document menu

Right-click for the button menu (fixed with event.stopPropagation)

Now the button-level menu works as intended. But the price is high. We forever deny access to information about right-clicks for any outer code, including counters that gather statistics and so on. That's quite unwise.

An alternative solution would be to check in the `document` handler if the default action was prevented? If it is so, then the event was handled, and we don't need to react on it.

```
<p>Right-click for the document menu (added a check for event.defaultPrevented)</p>
<button id="elem">Right-click for the button menu</button>

<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Button context menu");
  };

  document.oncontextmenu = function(event) {
    if (event.defaultPrevented) return;

    event.preventDefault();
    alert("Document context menu");
  };
</script>
```

Right-click for the document menu (added a check for event.defaultPrevented)

Right-click for the button menu

Now everything also works correctly. If we have nested elements, and each of them has a context menu of its own, that would also work. Just make sure to check for `event.defaultPrevented` in each `contextmenu` handler.

event.stopPropagation() and event.preventDefault()

As we can clearly see, `event.stopPropagation()` and `event.preventDefault()` (also known as `return false`) are two different things. They are not related to each other.

Nested context menus architecture

There are also alternative ways to implement nested context menus. One of them is to have a single global object with a handler for `document.oncontextmenu`, and also methods that allow us to store other handlers in it.

The object will catch any right-click, look through stored handlers and run the appropriate one.

But then each piece of code that wants a context menu should know about that object and use its help instead of the own `contextmenu` handler.

Summary

There are many default browser actions:

- `mousedown` – starts the selection (move the mouse to select).
- `click` on `<input type="checkbox">` – checks/unchecks the `input`.
- `submit` – clicking an `<input type="submit">` or hitting `Enter` inside a form field causes this event to happen, and the browser submits the form after it.
- `keydown` – pressing a key may lead to adding a character into a field, or other actions.
- `contextmenu` – the event happens on a right-click, the action is to show the browser context menu.
- …there are more…

All the default actions can be prevented if we want to handle the event exclusively by JavaScript.

To prevent a default action – use either `event.preventDefault()` or `return false`. The second method works only for handlers assigned with `on<event>`.

The `passive: true` option of `addEventListener` tells the browser that the action is not going to be prevented. That's useful for some mobile events, like `touchstart` and `touchmove`, to tell the browser that it should not wait for all handlers to finish before scrolling.

If the default action was prevented, the value of `event.defaultPrevented` becomes `true`, otherwise it's `false`.

Stay semantic, don't abuse

Technically, by preventing default actions and adding JavaScript we can customize the behavior of any elements. For instance, we can make a link `<a>` work like a button, and a button `<button>` behave as a link (redirect to another URL or so).

But we should generally keep the semantic meaning of HTML elements. For instance, `<a>` should perform navigation, not a button.

Besides being "just a good thing", that makes your HTML better in terms of accessibility.

Also if we consider the example with `<a>`, then please note: a browser allows us to open such links in a new window (by right-clicking them and other means). And people like that. But if we make a button behave as a link using JavaScript and even look like a link using CSS, then `<a>`-specific browser features still won't work for it.

Tasks

Why "return false" doesn't work?

importance: 3

Why in the code below `return false` doesn't work at all?

```
<script>
  function handler() {
    alert( "..." );
    return false;
  }
</script>

<a href="https://w3.org" onclick="handler()">the browser will go to w3.org</a
```

the browser will go to w3.org

The browser follows the URL on click, but we don't want it.

How to fix?

solution

Catch links in the element

importance: 5

Make all links inside the element with `id="contents"` ask the user if they really want to leave. And if they don't then don't follow.

Like this:

#contents

How about to read Wikipedia or visit *W3.org* and learn about modern standards?

Details:

- HTML inside the element may be loaded or regenerated dynamically at any time, so we can't find all links and put handlers on them. Use event delegation.
- The content may have nested tags. Inside links too, like `<a href=".."><i>...</i></a>` .

Open a sandbox for the task.

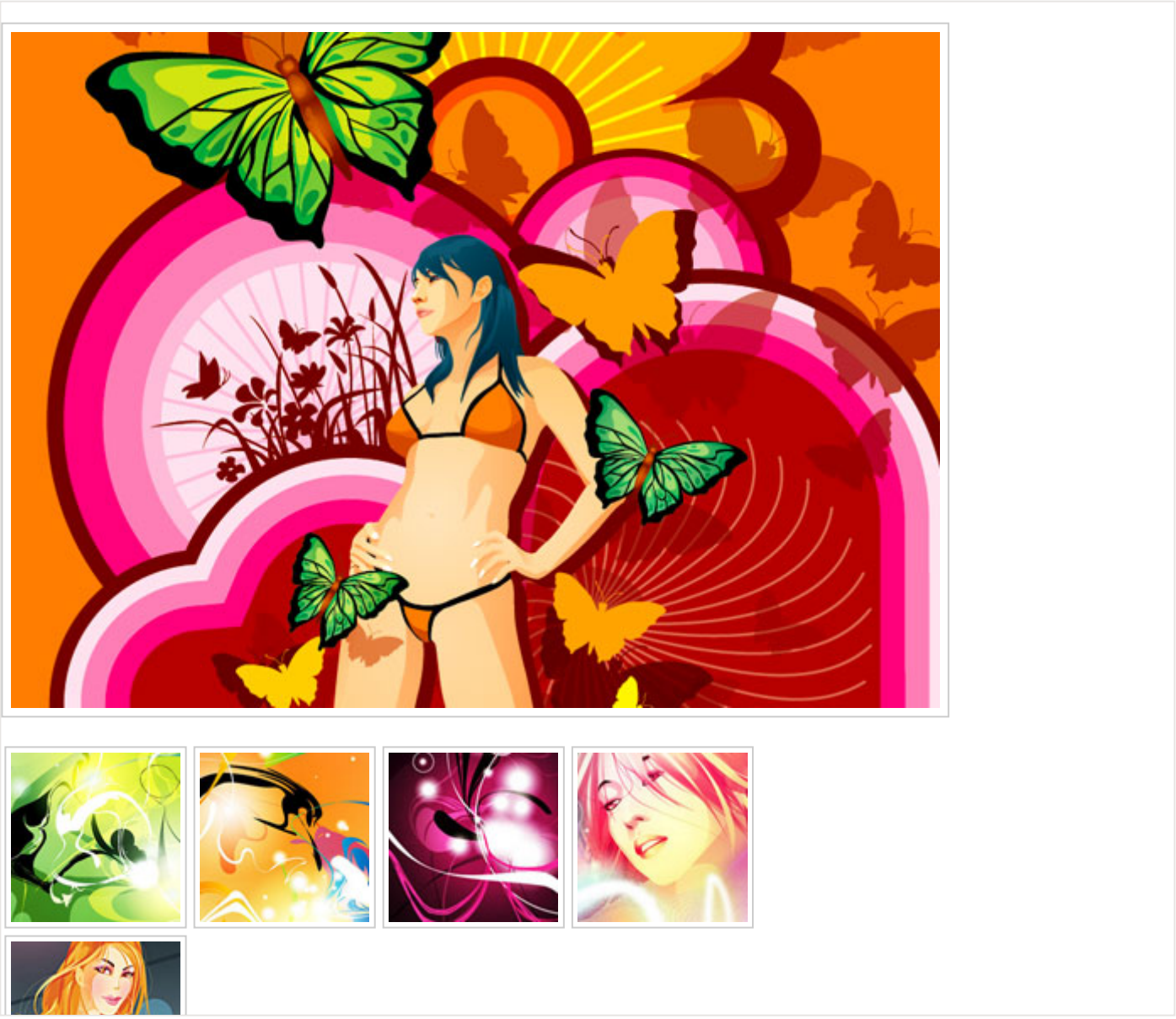
solution

Image gallery

importance: 5

Create an image gallery where the main image changes by the click on a thumbnail.

Like this:



P.S. Use event delegation.

Open a sandbox for the task.

solution

Previous lesson Next lesson

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…)