Lesson - 19

Browser Events

SkillUp by Dobrea Vladislav

Lesson Plan

- Modifying styles and classes
- Intro to browser events
- Bubbling and capturing
- Event delegation

Styles and Classes

Before we get into JavaScript's ways of dealing with styles and classes – here's an important rule. Hopefully it's obvious enough, but we still have to mention it.

There are generally two ways to style an element:

1. Create a class in CSS and add it: `<div class="...">`
2. Write properties directly into `style`: `<div style="...">`.

JavaScript can modify both classes and `style` properties.

We should always prefer CSS classes to `style`. The latter should only be used if classes "can't handle it".

For example, `style` is acceptable if we calculate coordinates of an element dynamically and want to set them from JavaScript, like this:

```
let top = /* complex calculations */;
let left = /* complex calculations */;

elem.style.left = left; // e.g '123px', calculated at run-time
elem.style.top = top; // e.g '456px'
```

For other cases, like making the text red, adding a background icon – describe that in CSS and then add the class (JavaScript can do that). That's more flexible and easier to support.

ClassName and ClassList

Changing a class is one of the most often used actions in scripts.

In the ancient time, there was a limitation in JavaScript: a reserved word like `"class"` could not be an object property. That limitation does not exist now, but at that time it was impossible to have a `"class"` property, like `elem.class`.

So for classes the similar-looking property `"className"` was introduced: the `elem.className` corresponds to the `"class"` attribute.

For instance:

```
<body class="main page">
  <script>
    alert(document.body.className); // main page
  </script>
</body>
```

If we assign something to `elem.className`, it replaces the whole string of classes. Sometimes that's what we need, but often we want to add/remove a single class.

ClassName and ClassList

There's another property for that: `elem.classList`.

The `elem.classList` is a special object with methods to `add/remove/toggle` a single class.

For instance:

```
<body class="main page">
  <script>
    // add a class
    document.body.classList.add('article');

    alert(document.body.className); // main page article
  </script>
</body>
```

So we can operate both on the full class string using `className` or on individual classes using `classList`. What we choose depends on our needs.

Methods of `classList`:

- `elem.classList.add/remove("class")` – adds/removes the class.
- `elem.classList.toggle("class")` – adds the class if it doesn't exist, otherwise removes it.
- `elem.classList.contains("class")` – checks for the given class, returns `true/false`.

Element style

The property `elem.style` is an object that corresponds to what's written in the `"style"` attribute. Setting `elem.style.width="100px"` works the same as if we had in the attribute `style` a string `width:100px`.

For multi-word property the camelCase is used:

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

For instance:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Prefixed properties

Browser-prefixed properties like `-moz-border-radius`, `-webkit-border-radius` also follow the same rule: a dash means upper case.

For instance:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Reseting style property

Sometimes we want to assign a style property, and later remove it.

For instance, to hide an element, we can set `elem.style.display = "none"`.

Then later we may want to remove the `style.display` as if it were not set. Instead of `delete elem.style.display` we should assign an empty string to it: `elem.style.display = ""`.

```
// if we run this code, the <body> will blink
document.body.style.display = "none"; // hide

setTimeout(() => document.body.style.display = "", 1000); // back to normal
```

If we set `style.display` to an empty string, then the browser applies CSS classes and its built-in styles normally, as if there were no such `style.display` property at all.

Full rewrite with `style.cssText`

Normally, we use `style.*` to assign individual style properties. We can't set the full style like `div.style="color: red; width: 100px"`, because `div.style` is an object, and it's read-only.

To set the full style as a string, there's a special property `style.cssText`:

```
<div id="div">Button</div>

<script>
  // we can set special style flags like "important" here
  div.style.cssText=`color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center;
  `;

  alert(div.style.cssText);
</script>
```

This property is rarely used, because such assignment removes all existing styles: it does not add, but replaces them. May occasionally delete something needed. But we can safely use it for new elements, when we know we won't delete an existing style.

The same can be accomplished by setting an attribute: `div.setAttribute('style', 'color: red...')`.

Mind the units

Don’t forget to add CSS units to values.

For instance, we should not set `elem.style.top` to `10`, but rather to `10px`. Otherwise it wouldn’t work:

```
<body>
  <script>
    // doesn't work!
    document.body.style.margin = 20;
    alert(document.body.style.margin); // '' (empty string, the assignment is

    // now add the CSS unit (px) - and it works
    document.body.style.margin = '20px';
    alert(document.body.style.margin); // 20px

    alert(document.body.style.marginTop); // 20px
    alert(document.body.style.marginLeft); // 20px
  </script>
</body>
```

Please note: the browser “unpacks” the property `style.margin` in the last lines and infers `style.marginLeft` and `style.marginTop` from it.

Computed styles

So, modifying a style is easy. But how to *read* it?

For instance, we want to know the size, margins, the color of an element. How to do it?

The `style` property operates only on the value of the `"style"` attribute, without any CSS cascade.

So we can't read anything that comes from CSS classes using `elem.style`.

For instance, here `style` doesn't see the margin:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  The red text
  <script>
    alert(document.body.style.color); // empty
    alert(document.body.style.marginTop); // empty
  </script>
</body>
```

...But what if we need, say, to increase the margin by `20px`? We would want the current value of it.

getComputedStyle

```
getComputedStyle(element, [pseudo])
```

element

Element to read the value for.

pseudo

A pseudo-element if required, for instance `::before`. An empty string or no argument means the element itself.

The result is an object with styles, like `elem.style`, but now with respect to all CSS classes.

For instance:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  <script>
    let computedStyle = getComputedStyle(document.body);

    // now we can read the margin and the color from it

    alert( computedStyle.marginTop ); // 5px
    alert( computedStyle.color ); // rgb(255, 0, 0)
  </script>

</body>
```

Computed and resolved values

There are two concepts in CSS:

1. A *computed* style value is the value after all CSS rules and CSS inheritance is applied, as the result of the CSS cascade. It can look like `height:1em` or `font-size:125%`.
2. A *resolved* style value is the one finally applied to the element. Values like `1em` or `125%` are relative. The browser takes the computed value and makes all units fixed and absolute, for instance: `height:20px` or `font-size:16px`. For geometry properties resolved values may have a floating point, like `width:50.5px`.

A long time ago `getComputedStyle` was created to get computed values, but it turned out that resolved values are much more convenient, and the standard changed.

So nowadays `getComputedStyle` actually returns the resolved value of the property, usually in `px` for geometry.

⚠️ `getComputedStyle` requires the full property name

We should always ask for the exact property that we want, like `paddingLeft` or `marginTop` or `borderTopWidth`. Otherwise the correct result is not guaranteed.

For instance, if there are properties `paddingLeft/paddingTop`, then what should we get for `getComputedStyle(elem).padding`? Nothing, or maybe a "generated" value from known paddings? There's no standard rule here.

There are other inconsistencies. As an example, some browsers (Chrome) show `10px` in the document below, and some of them (Firefox) – do not:

```
<style>
  body {
    margin: 10px;
  }
</style>
<script>
  let style = getComputedStyle(document.body);
  alert(style.margin); // empty string in Firefox
</script>
```

To manage classes, there are two DOM properties:

- `className` – the string value, good to manage the whole set of classes.
- `classList` – the object with methods `add/remove/toggle/contains`, good for individual classes.

To change the styles:

- The `style` property is an object with camelCased styles. Reading and writing to it has the same meaning as modifying individual properties in the `"style"` attribute. To see how to apply `important` and other rare stuff – there's a list of methods at MDN.
- The `style.cssText` property corresponds to the whole `"style"` attribute, the full string of styles.

To read the resolved styles (with respect to all classes, after all CSS is applied and final values are calculated):

- The `getComputedStyle(elem, [pseudo])` returns the style-like object with them. Read-only.

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:

```
1 <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.

Event Handlers

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.

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

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.

addEventListner

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.

addEventListner syntax

```
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`.

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>
```

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:

```
1 <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.

Bubbling and capturing

Let’s start with an example.

This handler is assigned to `<div>`, but also runs if you click any nested tag like `<em>` or `<code>`:

```
<div onclick="alert('The handler!')">
  <em>If you click on <code>EM</code>, the handler on <code>DIV</code> runs.</
</div>
```

If you click on EM, the handler on DIV runs.

Isn’t it a bit strange? Why does the handler on `<div>` run if the actual click was on `<em>`?

Bubbling

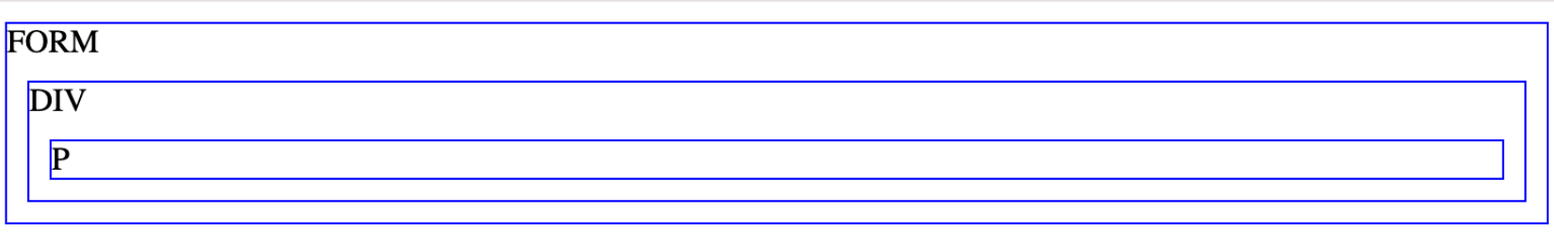
The bubbling principle is simple.

When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

Let's say we have 3 nested elements `FORM > DIV > P` with a handler on each of them:

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```



A click on the inner `<p>` first runs `onclick`:

1. On that `<p>`.
2. Then on the outer `<div>`.
3. Then on the outer `<form>`.
4. And so on upwards till the `document` object.

⚠️ *Almost* all events bubble.

The key word in this phrase is “almost”.

For instance, a `focus` event does not bubble. There are other examples too, we’ll meet them. But still it’s an exception, rather than a rule, most events do bubble.

Event target

A handler on a parent element can always get the details about where it actually happened.

The most deeply nested element that caused the event is called a *target* element, accessible as `event.target`.

Note the differences from `this` (= `event.currentTarget`):

- `event.target` – is the "target" element that initiated the event, it doesn't change through the bubbling process.
- `this` – is the "current" element, the one that has a currently running handler on it.

For instance, if we have a single handler `form.onclick`, then it can "catch" all clicks inside the form. No matter where the click happened, it bubbles up to `<form>` and runs the handler.

In `form.onclick` handler:

- `this` (= `event.currentTarget`) is the `<form>` element, because the handler runs on it.
- `event.target` is the actual element inside the form that was clicked.

Stoping bubbling

A bubbling event goes from the target element straight up. Normally it goes upwards till `<html>`, and then to `document` object, and some events even reach `window`, calling all handlers on the path.

But any handler may decide that the event has been fully processed and stop the bubbling.

The method for it is `event.stopPropagation()`.

For instance, here `body.onclick` doesn't work if you click on `<button>`:

```
<body onclick="alert(`the bubbling doesn't reach here`)">
  <button onclick="event.stopPropagation()">Click me</button>
</body>
```

Click me

event.stopImmediatePropagation()

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.

In other words, `event.stopPropagation()` stops the move upwards, but on the current element all other handlers will run.

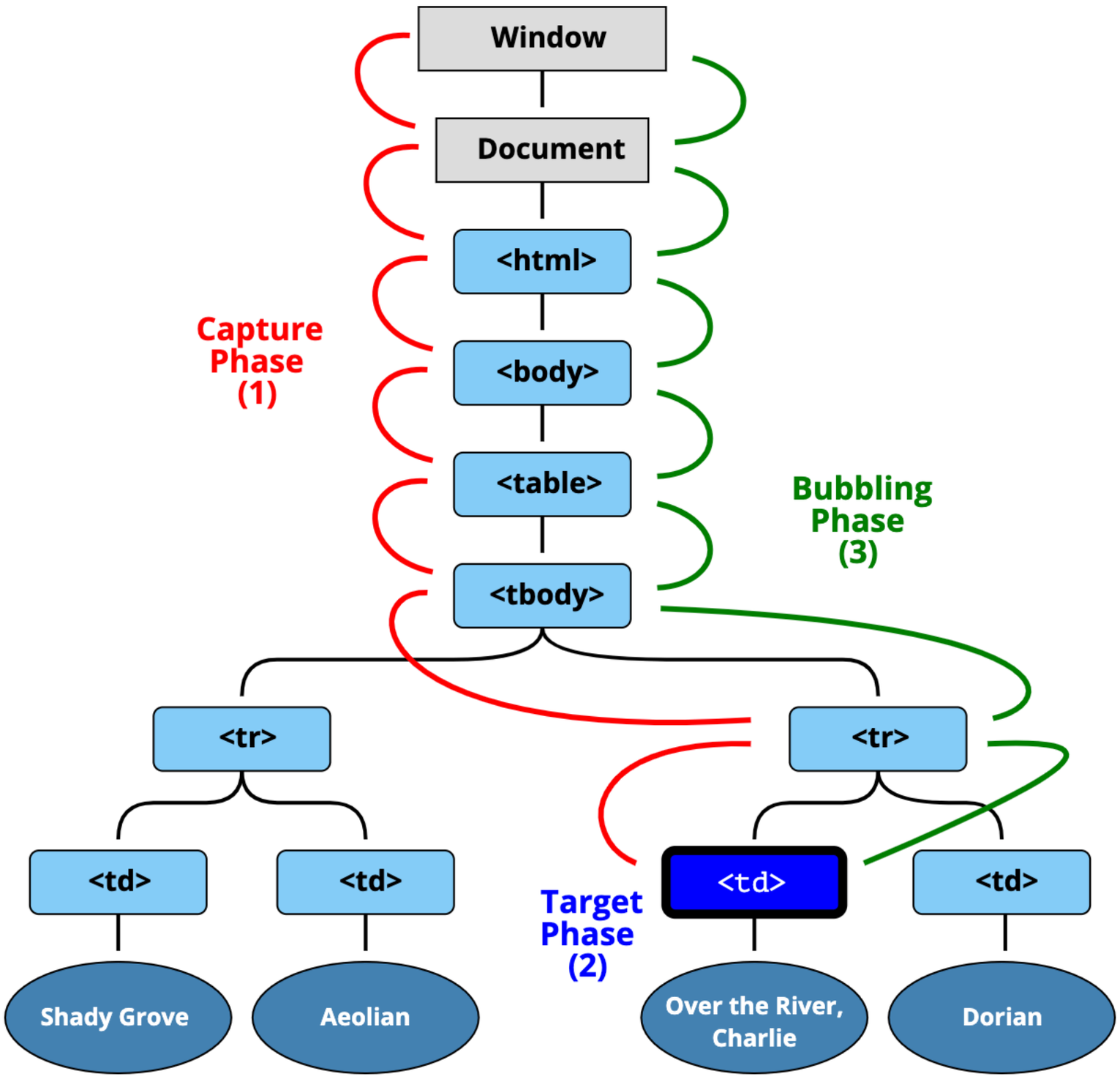
To stop the bubbling and prevent handlers on the current element from running, there's a method `event.stopImmediatePropagation()`. After it no other handlers execute.

Capturing

There's another phase of event processing called "capturing". It is rarely used in real code, but sometimes can be useful.

The standard DOM Events describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.


Window
Document
<html>
<body>
<table>
<tbody>
<tr>
<tr>
<td>
<td>
<td>
<td>
Shady Grove
Aeolian
Over the River, Charlie
Dorian
Capture Phase (1)
Target Phase (2)
Bubbling Phase (3)

That is: for a click on `<td>` the event first goes through the ancestors chain down to the element (capturing phase), then it reaches the target and triggers there (target phase), and then it goes up (bubbling phase), calling handlers on its way.

Before we only talked about bubbling, because the capturing phase is rarely used. Normally it is invisible to us.

Handlers added using `on<event>`-property or using HTML attributes or using two-argument `addEventListener(event, handler)` don't know anything about capturing, they only run on the 2nd and 3rd phases.

To catch an event on the capturing phase, we need to set the handler `capture` option to `true`:

```
elem.addEventListener(..., {capture: true})
// or, just "true" is an alias to {capture: true}
elem.addEventListener(..., true)
```

There are two possible values of the `capture` option:

- If it's `false` (default), then the handler is set on the bubbling phase.
- If it's `true`, then the handler is set on the capturing phase.

Note that while formally there are 3 phases, the 2nd phase ("target phase": the event reached the element) is not handled separately: handlers on both capturing and bubbling phases trigger at that phase.

Event delegation

Capturing and bubbling allow us to implement one of most powerful event handling patterns called *event delegation*.

The idea is that if we have a lot of elements handled in a similar way, then instead of assigning a handler to each of them – we put a single handler on their common ancestor.

In the handler we get `event.target`, see where the event actually happened and handle it.

Event delegation

Let's see an example – the Ba-Gua diagram reflecting the ancient Chinese philosophy.

Here it is:

***Bagua* Chart: Direction, Element, Color, Meaning**

Northwest Metal Silver Elders	**North** Water Blue Change	**Northeast** Earth Yellow Direction
West Metal Gold Youth	**Center** All Purple Harmony	**East** Wood Blue Future
Southwest Earth Brown Tranquility	**South** Fire Orange Fame	**Southeast** Wood Green Romance

Event delegation

Our task is to highlight a cell `<td>` on click.

Instead of assign an `onclick` handler to each `<td>` (can be many) – we'll setup the "catch-all" handler on `<table>` element.

It will use `event.target` to get the clicked element and highlight it.

The code:

```
let selectedTd;

table.onclick = function(event) {
  let target = event.target; // where was the click?

  if (target.tagName != 'TD') return; // not on TD? Then we're not interested

  highlight(target); // highlight it
};

function highlight(td) {
  if (selectedTd) { // remove the existing highlight if any
    selectedTd.classList.remove('highlight');
  }
  selectedTd = td;
  selectedTd.classList.add('highlight'); // highlight the new td
}
```

Such a code doesn't care how many cells there are in the table. We can add/remove `<td>` dynamically at any time and the highlighting will still work.

Event delegation

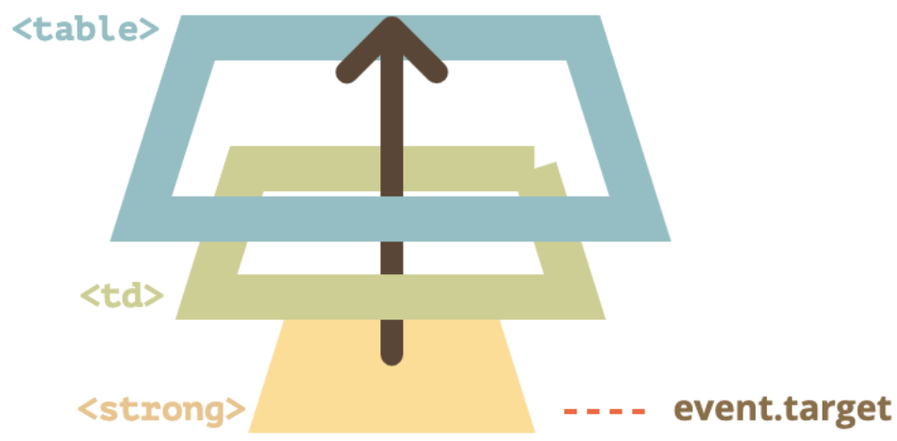
Still, there's a drawback.

The click may occur not on the `<td>`, but inside it.

In our case if we take a look inside the HTML, we can see nested tags inside `<td>`, like `<strong>`:

```
<td>
  <strong>Northwest</strong>
  ...
</td>
```

Naturally, if a click happens on that `<strong>` then it becomes the value of `event.target`.



In the handler `table.onclick` we should take such `event.target` and find out whether the click was inside `<td>` or not.

Event delegation

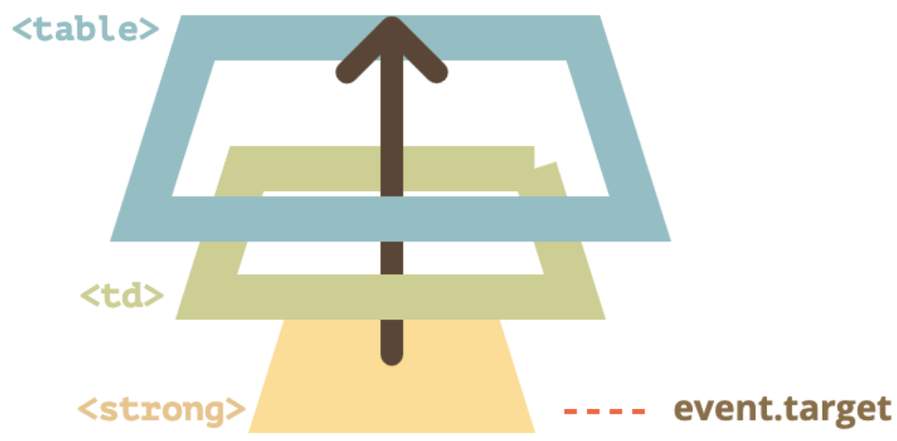
Still, there's a drawback.

The click may occur not on the `<td>`, but inside it.

In our case if we take a look inside the HTML, we can see nested tags inside `<td>`, like `<strong>`:

```
<td>
  <strong>Northwest</strong>
  ...
</td>
```

Naturally, if a click happens on that `<strong>` then it becomes the value of `event.target`.



In the handler `table.onclick` we should take such `event.target` and find out whether the click was inside `<td>` or not.

Here's the improved code:

```
table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)

  highlight(td); // (4)
};
```

Explanations:

1. The method `elem.closest(selector)` returns the nearest ancestor that matches the selector. In our case we look for `<td>` on the way up from the source element.
2. If `event.target` is not inside any `<td>`, then the call returns immediately, as there's nothing to do.
3. In case of nested tables, `event.target` may be a `<td>`, but lying outside of the current table. So we check if that's actually *our table's* `<td>`.
4. And, if it's so, then highlight it.

As the result, we have a fast, efficient highlighting code, that doesn't care about the total number of `<td>` in the table.

Summary

Event delegation is really cool! It's one of the most helpful patterns for DOM events.

It's often used to add the same handling for many similar elements, but not only for that.

The algorithm:

1. Put a single handler on the container.
2. In the handler – check the source element `event.target`.
3. If the event happened inside an element that interests us, then handle the event.

Benefits:

- Simplifies initialization and saves memory: no need to add many handlers.
- Less code: when adding or removing elements, no need to add/remove handlers.
- DOM modifications: we can mass add/remove elements with `innerHTML` and the like.

The delegation has its limitations of course:

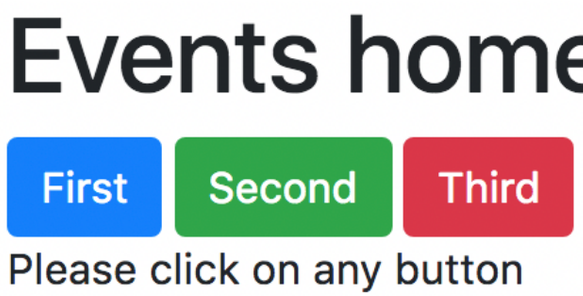
- First, the event must be bubbling. Some events do not bubble. Also, low-level handlers should not use `event.stopPropagation()`.
- Second, the delegation may add CPU load, because the container-level handler reacts on events in any place of the container, no matter whether they interest us or not. But usually the load is negligible, so we don't take it into account.

Learning Resources

1. **Styles and Classes: https://javascript.info/styles-and-classes**
2. **Intro to browser event https://javascript.info/introduction-browser-events**
3. **Bubbling and capture: https://javascript.info/bubbling-and-capturing**
4. **Event Delegation: https://javascript.info/event-delegation**

Home Work

1. Create a page, on which clicking on the button ‘First’, ‘Second’, ‘Third’, will change text on the page from ‘Please click on any button’ to ‘You clicked First button’, or ‘You clicked Second button’ и ‘You clicked Third button’.



2. By clicking on the button, console log, the following information: tagName, event phase (example bubbling)

3. Advanced: Use event delegation principle