ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

SESSION 4 - BUBBLING AND DELEGATION

Event handlers
Bubbling
Delegation

EVENT HANDLERS

You can register an event handler the following way:

```
node.addEventListener(
    event_type,
    function_to_run
);
```

```
<a id="link" href="">Link to click</a>
```

```
var link = document.getElementById("link");

link.addEventListener(
    "click",
    function (event) {
      console.log(event)
    }
);
```

Every event handler declared gets an event object assigned to its argument handler function. In the previous example, I called it `event`.

BUBBLING

Remember that in the DOM, when an event is triggered, it fires first on the most downwards element that triggered it (the last element that doesn't have any children) and then goes up the tree, triggering the same event for the parents and grand-parents and etc. of that element.

This means that all event handlers that have been declared on elements up the tree will fire if any children or grand-children are firing the event.

Bubbling Example

See it live

You can prevent bubbling if needed by calling the `stopPropagation` method on the event object:

```
event.stopPropagation();
```

In this case, an event will not bubble up to its parents or grand-parents.

EXERCISE 1

Can you create a small test case for an event that does not bubble up the tree? For example, mouse leave does not bubble up.

Use the html in exercise1.html and create a simple test case for this.

```
var parent = document.getElementById("parent");
var child = document.getElementById("child");


parent.addEventListener(
  "mouseleave",
  function () {
    alert("parent mouse leave event!");
  }
);

child.addEventListener(
  "mouseleave",
  function () {
    alert("child mouse leave event!");
  }
);
```

DELEGATION

The event object is interesting because it has a `target` property, that returns the first most nested element that triggered the event.

It also has a `currentTarget` property, that returns the item the current event handler was attached to.

Remember that the event object is automatically assigned as the argument of the function used when declaring the event handler.

We can use this to our advantage! With event delegation.

Using this pattern, you can declare only one event handler on a parent that has multiple children and using the `event` object find out which child triggered the event.

In exercise2.html, declare a new event handler for a click on the `div` that has an id of "button-wrapper".

Console log `currentTarget` of the event object.

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    console.log(event.currentTarget);
  }
);
```

Using the exact same code as previously, now console log the `target` of the event object.

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    console.log(event.target);
  }
);
```

The second example tells us which button was clicked, without having to declare one event handler per button.

The other way to achieve this, without event delegation, is to declare one event handler per button:

```
function handlerFunction (event) {
  console.log(event.target);
}

document.querySelectorAll(".button")[0].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[1].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[2].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[3].addEventListener(
  "click",
  handlerFunction
);
document.querySelectorAll(".button")[4].addEventListener(
  "click",
  handlerFunction
);
```

There is one more thing we need to improve, if you go back to the following code and test it:

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    console.log(event.target);
  }
);
```

You will see that it fires when we click on the buttons, but also when we click on any area of the parent, but outside a button.

We probably want to change our event handler slightly so that the handler function checks which element is actually clicked before doing anything.

Amend the previous code to check if the `target` element is a button or not.

There are mutliple ways of doing this, you can check the type of html element of the button, or check the class on the element.

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    if (event.target.classList.contains("button")) {
      console.log(event.target);
    }
  }
);
```

```
var wrapper = document.querySelectorAll(".button-wrapper")[0];

wrapper.addEventListener(
  "click",
  function (event) {
    if (event.target.nodeName === "P") {
      console.log(event.target);
    }
  }
);
```

why is event delegation usefull?

- lets you write less code and declare less event handlers
- if you need to remove or add elements dynamically in the DOM, attaching event handlers can quickly become a problem.

Imagine our button situation, where you need to declare an event handler for each button in the list. If you need to add or remove buttons dynamically, you will also need to update your event handlers as you create the buttons.

Using event delegation, you can add or remove buttons safely, declare one event handler on the wrapper and check which button is clicked.

EXERCISE 3

In exercise3.html, there is a list of elements with a description.

Write a small app that adds an element when clicking the add button. The description should be the text entered in the input field.

When cliking anywhere on an exsiting element, it removes it from the DOM.

Extra: use the revealing module pattern!

```
var
  newEl,
  newElementDescription = document.querySelectorAll(".element-description
  addButton = document.querySelectorAll(".add-button")[0],
  elWrapper = document.querySelectorAll(".element-wrapper")[0],
  inputEl = document.querySelectorAll(".element-description")[0];


elWrapper.addEventListener(
  "click",
  function (event) {
    var el = event.target;
    if (el.classList.contains("element")) {
      el.parentNode.removeChild(el);
    }
  }
);

addButton.addEventListener(
  "click",
  function(event){
    newEl = "<li class=\"element\">" + inputEl.value + "<\/li>";
    elWrapper.insertAdjacentHTML("beforeend", newEl);
  }
);
```