[12] Events and Propagation

An event is an action or occurrence that takes place in the browser, such as a user clicking a button, moving the mouse, pressing a key, or a page finishing loading. JavaScript can detect these events and respond to them.

Event Types

There are numerous types of events in JavaScript, including:

- Mouse Events: These events are related to mouse actions, like clicking, moving, hovering, etc. Examples include click, mousedown, mouseup, and mousemove.
- Keyboard Events: These events are triggered by keyboard actions. Examples include keydown, keyup, and keypress.
- Form Events: These events are related to form elements, such as submit, focus, blur, and change.
- Window Events: Events related to the browser window itself, such as load, resize, and scroll.
- Custom Events: You can create your own custom events to trigger specific actions within your JavaScript code.

When an event occurs in HTML, the event belongs to a certain event object, like a mouse click event belongs to the MouseEvent object:
https://www.w3schools.com/jsref/obj_events.asp

All possible events: https://www.w3schools.com/jsref/dom_obj_event.asp

Event Listeners

Event listeners are functions that you attach to HTML elements. They "listen" for a specific event to occur and execute a callback function when that event occurs. Event listeners are added using the addEventListener method.

The `addEventListener()` method makes it easier to control how the event reacts to bubbling.

- The `addEventListener()` method attaches an event handler to the specified element.
- You can add event listeners to any DOM object not only HTML elements. i.e the window object.
- You can easily remove an event listener by using the `removeEventListener()` method.
- The `addEventListener()` method allows you to add many events to the same element, without overwriting existing events.

```
---Syntax:
element.addEventListener(event, function, useCapture);

---Example[1]
const button = document.getElementById('myButton');
button.addEventListener('click', function() {
// Code to execute when the button is clicked
});

---Example[2]
document.getElementById("myBtn").addEventListener("click", displayDate);
```

The event object

What is "e" or "event" (the parameter we pass in our functions)?

- When an event occurs, an event object is created and passed to the event handler. This object contains information about the event, such as the type of event, the target element, and any event-specific data. You can access this object within your event handler function.

```
---Example[1]
myBtn.addEventListener("click", function (e){
if (e.ctrlKey) {
console.log('CTRL key was pressed while clicking.');
}
});

---Example[2]
element.addEventListener('click', function(event) {
console.log('Event type:', event.type);
console.log('Target element:', event.target);
});
```

Event target

In JavaScript, the "event target" refers to the DOM (Document Object Model) element to which an event is originally dispatched or sent. When an event occurs, it typically happens on a specific element within the web page, and this element is considered the event target.

- For example, when a user clicks a button on a web page, the button element becomes the event target for the click event. Similarly, when a user types something into an input field, that input field becomes the event target for keyboard-related events.

```
---Example[1]
document.getElementById('myButton').addEventListener('click', function(event) {
const target = event.target; // This is the event target
// You can now work with the event target, such as changing its properties or behavior.
});
```

Preventing Default Behavior

Some events have default behaviors associated with them, such as form submissions or links navigating to a new page. You can prevent these default behaviors using the preventDefault() method, which is often used to enhance user interactions in web applications. Use Cases:

- Form Validation: Preventing form submission until data is validated.
- Ajax Requests: Intercepting form submissions to send data to the server via AJAX instead of a full page reload.
- Single-Page Applications (SPAs): Managing navigation and page updates without actually loading new HTML pages.

```
---Example[1]
const link = document.getElementById('myLink');
link.addEventListener('click', function(event) {
event.preventDefault(); // Prevent the link from navigating
});
```

Event Propagation

JavaScript event propagation is an important concept when dealing with multiple nested elements and event handling. It involves three phases: capturing, target, and bubbling.

Event delegation is a technique that takes advantage of these phases to handle events efficiently.

1. Event Bubbling:
    - Bubbling is the default behavior in which an event starts from the target element that triggered it and then bubbles up through the DOM hierarchy, reaching higher-level ancestor elements.
    - This means that if you have nested elements, such as a div within a section within a body, and you click on the innermost div, the event will propagate from the div to the section to the body, triggering any event listeners along the way.
    - Event listeners attached to parent elements can also respond to the same event.
    - In the following example, if you click the innermost div, the order of log messages will be "Div clicked," "Section clicked," and "Body clicked."

```
---Example[1]
document.body.addEventListener('click', function() {
console.log('Body clicked');
});
document.querySelector('section').addEventListener('click', function() {
console.log('Section clicked');
});
document.querySelector('div').addEventListener('click', function() {
console.log('Div clicked');
});
```

2. Event Capturing:
    - Capturing is the opposite of bubbling. It allows you to capture an event at the top of the DOM hierarchy and then propagate it down to the target element.
    - Event listeners attached during the capturing phase are executed before those attached during the bubbling phase.
    - You can enable capturing by setting the useCapture parameter to true when using addEventListener.
    - In the following example if you click the innermost div, the order of log messages will be "Body clicked during capturing phase," "Section clicked during capturing phase," and "Div clicked during capturing phase."

```
---Example[1]
```

```
document.body.addEventListener('click', function() {
console.log('Body clicked during capturing phase');
}, true);
document.querySelector('section').addEventListener('click', function() {
console.log('Section clicked during capturing phase');
}, true);
document.querySelector('div').addEventListener('click', function() {
console.log('Div clicked during capturing phase');
}, true);
```

3. Event Delegation:
    - Event delegation is a technique where you attach a single event listener to a common ancestor of multiple elements you want to interact with.
    - This is particularly useful when dealing with dynamically generated elements or a large number of elements, as it avoids the need to attach individual listeners to each element.
    - You can identify the actual target of the event by examining event.target within the handler.
    - In this example, you have a <ul> element with several <li> elements. Instead of attaching click listeners to each <li>, you attach one listener to the <ul>. When you click an <li>, the event bubbles up to the <ul>, and you can use event.target to identify the specific <li> that was clicked.

```
---Example[1]
document.querySelector('ul').addEventListener('click', function(event) {
if (event.target.tagName === 'LI') {
console.log('List item clicked:', event.target.textContent);
}
});
```