Lecture 15: Event Listener and Handler

The "First Thought" Principle: The Doorbell

Imagine your HTML page is a house. It has rooms (`div` s), furniture (`p` aragraphs), and interactive objects like a doorbell (`button`).

By itself, the doorbell is just a button on the wall. Pushing it does nothing. It's a static object.

An **event** is the **"ding-dong"** signal that the doorbell sends when it's pushed.
An **event listener** is the person **inside the house whose job is to listen for the doorbell.**
An **event handler** is the **action that person takes** when they hear the "ding-dong" (e.g., "get up and open the door").

In summary: An event is a signal that something happened. Your JavaScript code's job is to *listen* for these signals and *react* to them.

Without events, a webpage is a silent, static house. With events, it becomes a responsive home that reacts to the user.

In-Depth Guide to JavaScript Events

1. The Three Core Components

To make a webpage interactive, you always need three things:

1. **The Target Element:** Which HTML element are we waiting for an action on? (e.g., the `button`).
2. **The Event Type:** What specific action are we listening for? (e.g., a `'click'`).
3. **The Listener/Handler:** What code should be executed when that action happens? (e.g., a function that changes some text).

2. The Modern Way: `addEventListener()`

This is the standard, most powerful, and recommended way to handle events. You call it on the target element.

Syntax: `targetElement.addEventListener('eventType', functionToRun);`

Let's build a complete, simple example:

HTML:

```
<button id="action-button">Click Me</button> <p id="status-text">Waiting f
or action...</p>
```

JavaScript:

```
// Step 1: Select the Target Element const myButton = document.getElementB
yId('action-button'); const statusText = document.getElementById('status-t
ext'); // Step 2: Define the Handler Function (the "action plan") // This
function will be executed when the event occurs. function onButtonClick()
{ console.log("Button was clicked!"); statusText.textContent = "Action Per
formed!"; statusText.style.color = 'green'; } // Step 3: Attach the Listen
er // We tell the button: "Hey, start listening for a 'click' event. // Wh
en you hear one, execute the `onButtonClick` function." myButton.addEventL
istener('click', onButtonClick);
```

Critical Point: We pass the function name `onButtonClick` directly. We do **not** call it with `()`. We are giving `addEventListener` a reference to our function—the "recipe"—to be used later.

```
// Mouse Events element.addEventListener('click', (e) =>
console.log('Clicked')); element.addEventListener('dblclick', (e) =>
console.log('Double clicked')); element.addEventListener('mousedown', (e)
=> console.log('Mouse down')); element.addEventListener('mouseup', (e) =>
console.log('Mouse up')); element.addEventListener('mousemove', (e) =>
console.log('Mouse moving')); element.addEventListener('mouseenter', (e)
=> console.log('Mouse entered')); element.addEventListener('mouseleave',
(e) => console.log('Mouse left'));
```

```
// Keyboard Events document.addEventListener('keydown', (e) => {
console.log('Key pressed:', e.key); console.log('Key code:', e.code);
console.log('Ctrl pressed:', e.ctrlKey); console.log('Shift pressed:',
e.shiftKey); console.log('Alt pressed:', e.altKey); });
```

```
document.addEventListener('keyup', (e) => console.log('Key released'));
document.addEventListener('keypress', (e) => console.log('Key press')); //
Deprecated
```

```
// Form Events const input = document.querySelector('input'); const form =
document.querySelector('form');
```

```
input.addEventListener('focus', () => console.log('Input focused'));
input.addEventListener('blur', () => console.log('Input blurred'));
input.addEventListener('input', (e) => console.log('Input value:',
e.target.value)); input.addEventListener('change', (e) =>
console.log('Changed:', e.target.value));
```

```
form.addEventListener('submit', (e) => { e.preventDefault(); // Prevent
form submission console.log('Form submitted'); });
```

```
// Window Events window.addEventListener('load', () => console.log('Page
fully loaded')); window.addEventListener('DOMContentLoaded', () =>
console.log('DOM ready')); window.addEventListener('resize', () =>
console.log('Window resized')); window.addEventListener('scroll', () =>
console.log('Page scrolled'));
```

---

3. The `event` Object: The Information Packet

When an event occurs and your handler function is called, the browser automatically passes a special object as the **first argument** to your function. This is the `event` **object**.

First Thought: "The `event` object is an information packet that tells me everything about what just happened."

Let's modify our handler to accept this object:

```
function onButtonClick(event) { // Let's inspect the information packet co
nsole.log(event); } myButton.addEventListener('click', onButtonClick);
```

If you run this and click the button, your console will show a `PointerEvent` or `MouseEvent` object filled with useful information: the exact x/y coordinates of the click, whether the Shift key was held down, and much more.

Two of the most important properties of the `event` object are:

A. `event.target`
This property tells you the **specific element that triggered the event**. This is incredibly useful when you have one listener on a parent element watching for clicks on many children.

Example:

```
<ul id="item-list"> <li>Item 1</li> <li>Item 2</li> <li>Item 3</li> </ul>
```

```
const list = document.getElementById('item-list'); list.addEventListener
('click', function(event) { // 'this' would be the <ul>, but 'event.targe
t' is the specific <li> that was clicked! console.log("You clicked on:", e
vent.target.textContent); });
```

This pattern, called **event delegation**, is very efficient.

B. `event.preventDefault()`
This method stops the browser's **default behavior** for an element.

Classic Use Case: Form Submission
By default, when you submit an HTML form, the browser tries to send the data to a server and **reloads the page**. We usually want to stop this in a modern web app.

HTML:

```
<form id="my-form"> <input type="text" id="username"> <button type="submi
t">Submit</button> </form>
```

JavaScript:

```
const myForm = document.getElementById('my-form'); myForm.addEventListener
('submit', function(event) { // STOP the browser from reloading the page.
event.preventDefault(); const usernameInput = document.getElementById('use
rname'); console.log(`Form submitted with username: ${usernameInput.value}
`); // Now we can send this data using fetch() without a page refresh. });
```

---

4. Common Event Types to Know

- **Mouse Events:**
  - `click` : A single click.
  - `dblclick` : A double click.
  - `mousedown` : When the mouse button is pressed down.
  - `mouseup` : When the mouse button is released.
  - `mouseover` : When the mouse pointer enters an element.
  - `mouseout` : When the mouse pointer leaves an element.
  - `mousemove` : Fires continuously as the mouse moves over an element.
- **Keyboard Events:**
  - `keydown` : When a key is pressed down.
  - `keyup` : When a key is released.
  - `keypress` : (Older, avoid) Fires when a key that produces a character is pressed.

- **Form Events:**
  - `submit` : When a form is submitted.
  - `input` : Fires immediately when the value of an `<input>` , `<select>` , or `<textarea>` changes.
  - `change` : Fires when the value changes and the element loses focus.
- **Window Events:**
  - `load` : Fires when the entire page (including images, scripts, etc.) has finished loading.
  - `DOMContentLoaded` : Fires when the HTML document has been fully parsed and the DOM tree is ready (this is often a better choice than `load` as it fires earlier).
  - `scroll` : Fires when the user scrolls the document.

---

5. Removing Event Listeners

If you add an event listener, especially one that fires often (like `mousemove` or `scroll`), it's good practice to clean it up when it's no longer needed to prevent memory leaks. You do this with `.removeEventListener()` .

CRITICAL RULE: To remove a listener, you must have a **reference to the exact same function** that was used to add it. This means you cannot use an anonymous function.

```
// This works: function handleMouseMove() { /* ... */ } window.addEventLis
tener('mousemove', handleMouseMove); window.removeEventListener('mousemov
e', handleMouseMove); // Correct! // This does NOT work: window.addEventLi
stener('mousemove', () => { /* ... */ }); window.removeEventListener('mous
emove', () => { /* ... */ }); // WRONG! This is a different anonymous func
tion.
```

Bubbling and Capturing

The Setup

HTML Structure:

```
<div id="grandparent">GRANDPARENT <div id="parent">PARENT <button id="chil
d">CHILD (Click Me)</button> </div> </div>
```

JavaScript Code:
We will attach **only one** event listener, and it will be on the `grandparent`.

```
const grandparent = document.getElementById('grandparent'); grandparent.ad
dEventListener('click', (event) => { console.log("--- Event Listener on GR
ANDPARENT was triggered ---"); // The element the listener is attached to
console.log("event.currentTarget:", event.currentTarget.id); // The elemen
t where the click originated console.log("event.target:", event.target.i
d); });
```

(Note: `event.currentTarget` is the technically precise property for "the element the listener is on." In simple cases, it's the same as `this`, but it's clearer and works with arrow functions.)

The Action: You click on the `child` button.

Here is the complete, detailed flow of what the browser does, in order.

Phase 1: The Capturing Phase (The "Trickle Down")

The event is created and starts its journey from the top of the DOM tree down towards the target (`child` button). At each level, the browser checks for any *capturing* event listeners (`useCapture: true`).

1. **Event starts at `window`**. Browser asks: "Is there a *capturing* `click` listener on `window`?"
   - **Answer:** No. Event continues down.
2. **Event reaches `document`**. Browser asks: "Is there a *capturing* `click` listener on `document`?"
   - **Answer:** No. Event continues down.
3. **Event reaches `<html>`**. Browser asks: "Is there a *capturing* `click` listener on `<html>`?"
   - **Answer:** No. Event continues down.

4. **Event reaches** `<body>`. Browser asks: "Is there a *capturing* `click` listener on `<body>`?"
    - **Answer:** No. Event continues down.
5. **Event reaches** `<div id="grandparent">`. Browser asks: "Is there a *capturing* `click` listener on `grandparent`?"
    - **Answer:** No. Our listener was registered for the bubbling phase (the default). Event continues down.
6. **Event reaches** `<div id="parent">`. Browser asks: "Is there a *capturing* `click` listener on `parent`?"
    - **Answer:** No. Event continues down.

The capturing phase is now complete. **Nothing has been logged to the console yet.**

Phase 2: The Target Phase

The event has now arrived at the element that was directly clicked.

1. **Event is on** `<button id="child">`.
2. The browser checks for any `click` listeners (both capturing and bubbling) attached directly to the `child` button.
3. **Answer:** No. We didn't attach any listeners to the child.

The target phase is complete. **Still, nothing has been logged to the console.**

Phase 3: The Bubbling Phase (The "Bubble Up")

The event now turns around and starts its journey back up the DOM tree from the target. At each level, the browser checks for any *bubbling* event listeners (the default).

1. **Event leaves** `child` **and bubbles to its parent,** `<div id="parent">`.
    - Browser asks: "Is there a *bubbling* `click` listener on `parent`?"
    - **Answer:** No. Event continues up.

2. **Event bubbles to the next parent,** `<div id="grandparent">`.
   - Browser asks: "Is there a *bubbling* `click` listener on `grandparent`?"
   - **Answer: YES!** We attached one.
   - **Action:** The browser executes our callback function. It passes the `event` object that was created way back at the beginning of the process.

   Inside the Callback:
   - `event.currentTarget` is the element this listener is running on: the `grandparent` div.
   - `event.target` is the element where the event originated: the `child` button.

   The following is now printed to the console:

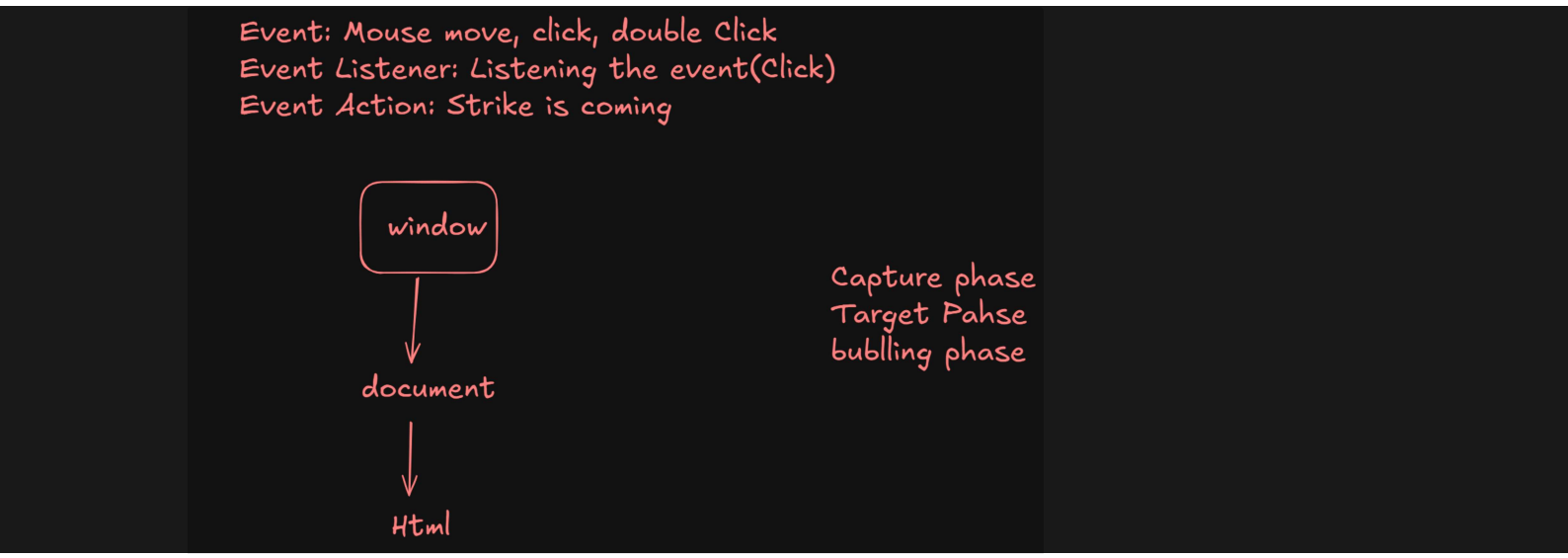
```
--- Event Listener on GRANDPARENT was triggered --- event.currentTarge
t: grandparent event.target: child
```

3. **Bubbling Continues:**
   - After our listener finishes, the event continues bubbling up to `<body>`, `<html>`, `document`, and `window`.
   - The browser checks for listeners at each of these levels. Since there are none, nothing else happens.
4. **Event Ends:** The event has completed its journey.

Summary and Conclusion

This flow perfectly demonstrates **Event Delegation**.

- We put a "net" (the event listener) on a high-level ancestor (`grandparent`).
- The event (`click`) happened on a deeply nested descendant (`child`).
- The **bubbling phase** is the mechanism that guarantees the event signal will travel up from the `child` and eventually be "caught" by our net on the `grandparent`.
- Inside the listener, the `event.target` property gives us the crucial information we need to identify the **original source** of the event, allowing us to distinguish between clicks on different grandchildren.


Event: Mouse move, click, double Click
Event Listener: Listening the event(Click)
Event Action: Strike is coming
window
document
Html
Capture phase
Target Pahse
bublling phase