Event Handlers in JavaScript (contd.)

Using Property Event Handlers

- By convention, JavaScript objects that fire events have a corresponding **onevent** properties (named by prefixing **on** to the name of the event).
- These properties are called to run associated handler code when the event is fired, and may also be called directly by your own code.
- To set event handler code you can just assign it to the appropriate **onevent** property.
- Only one event handler can be assigned for every event in an element. If needed the handler can be replaced by assigning another function to the same property.

Example:

```
const btn =
document.querySelector("button");

function greet(event) {
  console.log("greet:", event);
}

btn.onclick = greet;
```

In this example, we show how to set a simple **greet()** function for the click event using the **onclick** property.

Event Handlers in JavaScript (contd.)

Using addEventListeners

- The most flexible way to set an event handler on an element is to use the **`EventTarget.addEventListener`** method.
- This approach allows multiple listeners to be assigned to an element, and for listeners to be removed if needed (using **`EventTarget.removeEventListener`**).
- **Note:** The ability to add and remove event handlers allows you to, for example, have the same button performing different actions in different circumstances.
- In addition, in more complex programs cleaning up old/unused event handlers can improve efficiency.

Example:

```
const btn =
document.querySelector("button");

function greet(event) {
  console.log("greet:", event);
}

btn.addEventListener("click",
greet);
```

In given example we show how a simple greet() function can be set as a listener/event handler for the click event.

Event Object

- The **event** object is a JavaScript object that contains all the details about an event, such as the type of event, the element that triggered the event, and the state of the keyboard keys.
- Conventional names of event object used in JavaScript are **e**, **event**, or even **evt**.
- **Properties:**
  - **type**: Identifies the type of the event (e.g., "click", "keydown").
  - **target**: The element that triggered the event.
  - **currentTarget**: The element to which the event handler is attached.
  - **preventDefault()**: Method to prevent the default action associated with the event.
  - **stopPropagation()**: Method to stop the event from bubbling up the DOM tree.
  - **keyCode**: The code of the key pressed (for keyboard events).
  - **clientX** and **clientY** (for mouse events): The X and Y coordinates of the mouse pointer when the event was triggered.

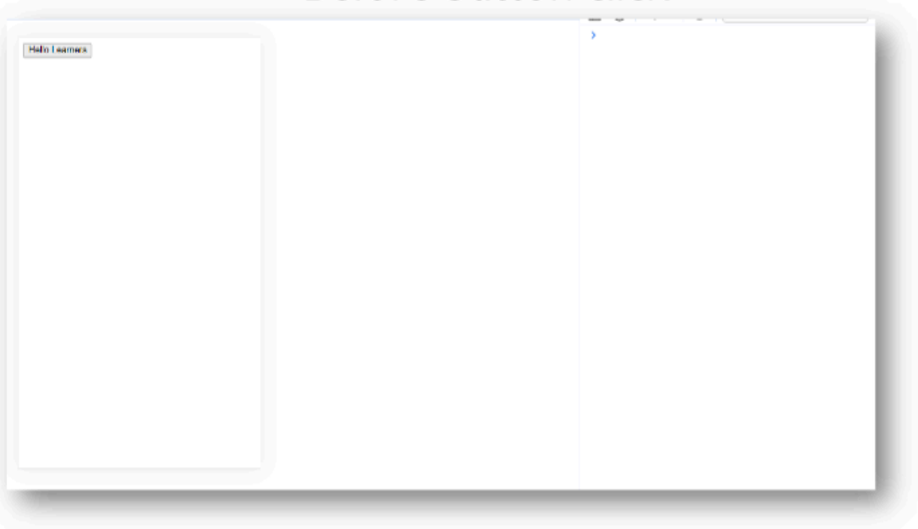
Event Object (contd.)

Example: Using the Event Object

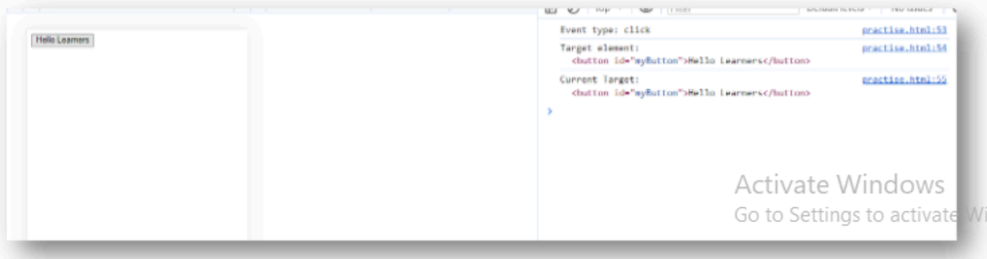
```
document.getElementById("myButton").
addEventListener("click",
function(event)
{
      console.log("Event type:", event.type);
      console.log("Target element:",
      event.target);
      console.log("Current Target:",
      event.currentTarget);
}
);
```

Output

Before button click



After button click



edureka!

Event Propagation

What is Propagation?

- Propagation describes the process by which events move through the Document Object Model (DOM) tree.
- This tree organizes elements in a hierarchy of parent, child, and sibling relationships. Imagine propagation like electricity flowing along a wire until it arrives at its intended target.
- An event must traverse each node in the DOM until it either reaches its final destination or is deliberately halted.
- **Event propagation** is a mechanism that defines how events propagate or travel through the DOM tree in web browsers.
- There are two types:
  - **Event Bubbling**
  - **Event Capturing**

Event Propagation – Bubbling

- In JavaScript, **event bubbling** is a method of event propagation within the HTML DOM API.
- This process operates on the principle that whenever an event occurs on an element, event handlers are first invoked on that element, then on its parent, and subsequently on all ancestor elements in the hierarchy.
- For instance, consider that element 'X' contains element 'Y', and element 'X' is clicked.
- In such a case, the click event activates the event on element 'X', which then bubbles up to trigger the event on element 'Y' as well.

```
addEventListener(EventType, Action, userCapture)
```

- **EventType:** This refers to the specific type of event that initiates the action.
- **Action:** This is the designated task to be executed when the event is triggered.
- **userCapture:** This Boolean value denotes the phase of an event. By default, this value is set to false, indicating that the event is in the bubbling phase.

Event Propagation – Bubbling Example

Example:

```
<body>
  <div class="grandparent">
    <div class="parent">
      <div class="child"></div>
    </div>
  </div>
  <script>
    const grandparent = document.querySelector(".grandparent");
    const parent = document.querySelector(".parent");
    const child = document.querySelector(".child");
    grandparent.addEventListener("click", () => { alert("grand parent element clicked");});
    parent.addEventListener("click", () => { alert("parent element clicked");});
    child.addEventListener("click", () => { alert("chld element clicked");});
  </script>
</body>
```

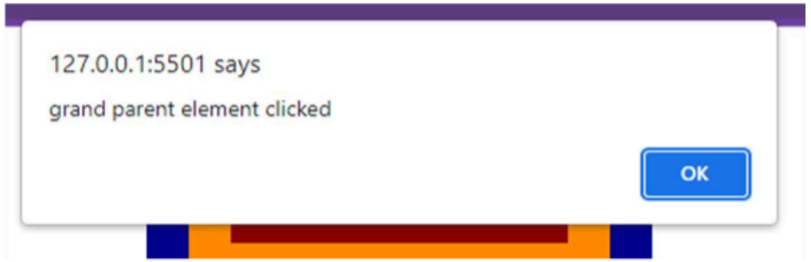
Event Propagation – Bubbling Example

Before clicking child box

Output

After clicking child

After clicking ok



After clicking ok

Event Propagation – Capturing

- **Event capturing**, sometimes mistakenly referred to as **trickle-down**, is the opposite of bubbling.
- It is the process where an event starts at the top of the DOM tree (document) and then descends down the tree until it reaches the target element.
- It's the first phase in the event propagation model.

```
addEventListener(EventType, Action, userCapture)
```

- **EventType:** This refers to a specific type of event that triggers an action.
- **Action:** This is the task to be performed when the event is triggered.
- **userCapture:** This is a Boolean value indicating the phase of an event. Setting this value to true indicates the capture phase, which is the initial phase where events can be intercepted as they trickle down the DOM hierarchy.

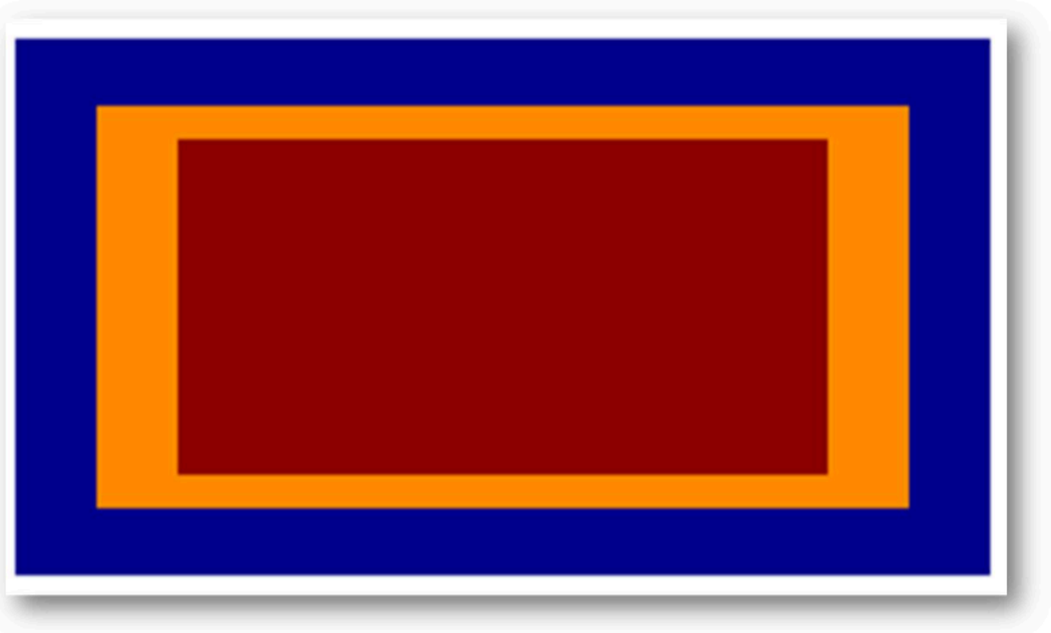
Event Propagation – Capturing Example

Example

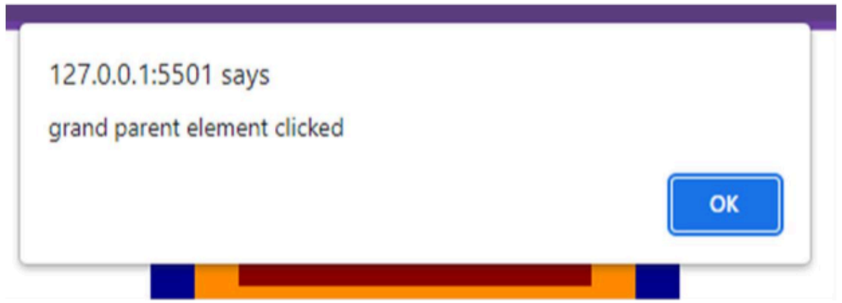
```
<body>
  <div class="grandparent">
    <div class="parent">
      <div class="child"></div>
    </div>
  </div>
  <script>
    const grandparent = document.querySelector(".grandparent");
    const parent = document.querySelector(".parent");
    const child = document.querySelector(".child");
    grandparent.addEventListener("click", () => { alert("grand parent element clicked");},true);
    parent.addEventListener("click", () => { alert("parent element clicked");},true);
    child.addEventListener("click", () => { alert("chld element clicked");},true);
  </script>
</body>
```

Event Propagation – Capturing Example

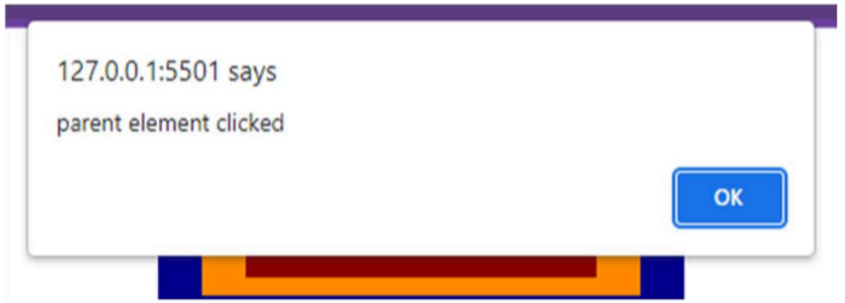
Before clicking child box



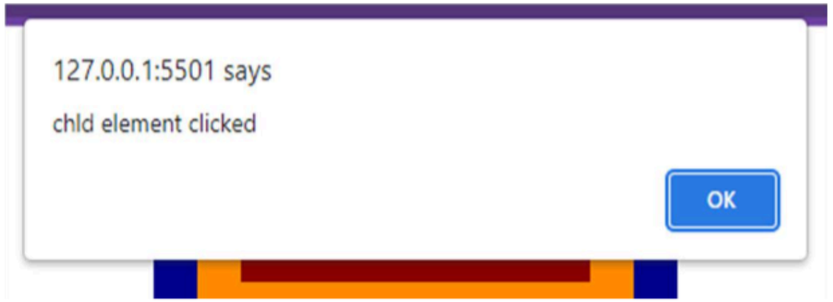
Output



After clicking grand parent



After clicking ok



After clicking ok

Stopping Propagation

- Sometimes, we may want to stop the propagation of an event to prevent it from reaching elements higher or lower in the DOM hierarchy.
- This can be achieved using the **stopPropagation()** method.
- In the example, when the button is clicked, the event listener attached to it fires.
- Inside this listener function, `event.stopPropagation()` is called, preventing the event from propagating further.
- As a result, the event does not reach the inner or outer divs.

Consider the following HTML structure:

```
<div id="outer">
  <div id="inner">
    <button id="btn">Click me!</button>
  </div>
</div>
```

And the script associated with it:

```
document.getElementById('btn').addEventListener('click',
function (event) {
            alert('Button clicked!');
            event.stopPropagation();
        });
        document.getElementById('inner').addEventListener('click
', function () {
            alert('Inner div clicked!');
        });
        document.getElementById('outer').addEventListener('click
', function () {
            alert('Outer div clicked!');
        });
```

Event Browser Monitoring and Compatibility

- The `monitorEvents()` method in browsers enables users to track various events happening on HTML elements.
- This function requires the element as its argument and records every event along with its details on the browser's console.
- For instance, in the following scenario, the user is monitoring all click events occurring on a button element using the monitorEvents() method:

```
var btn = document.getElementbyClass(".btnClick");
monitorEvents(btn, 'click');
```

- To stop monitoring events, you can use unmonitorEvents() function

```
var btn = document.getElementbyClass(".btnClick");
unmonitorEvents(btn, 'click');
```