Javascript Events

In JavaScript, **events** are actions or occurrences that happen in the browser, which the system tells your code about so it can react to them. They are fundamental to creating dynamic and interactive web pages.

Think of it like this: your web page is "listening" for things to happen. When something happens (an event), it can trigger a piece of JavaScript code to execute.

Key Concepts of JavaScript Events:

1. **What triggers an event?**
   Events can be triggered by various sources:
   - **User Interactions:**
     - **Mouse Events:** click, dblclick (double click), mouseover (mouse enters an element), mouseout (mouse leaves an element), mousemove (mouse moves over an element), mousedown, mouseup.
     - **Keyboard Events:** keydown (key pressed down), keyup (key released), keypress (key pressed and released).
     - **Form Events:** submit (form submitted), change (value of an input changes), focus (element gains focus), blur (element loses focus).
     - **Touch Events:** touchstart, touchmove, touchend (for touch-enabled devices).
   - **Browser Actions:**
     - load (page or element has finished loading).
     - resize (browser window is resized).
     - scroll (user scrolls the page).
     - unload (document is being unloaded).
   - **API Events:** Events related to specific Web APIs, like play or pause for video/audio elements, error events, etc.
   - **Custom Events:** You can even define and trigger your own custom events.
2. **Event Listeners (Event Handlers):**
   To react to an event, you attach an event listener (also often called an event handler) to a specific HTML element or the document or window object. An event listener is essentially a function that will be executed when the specified event occurs on the element it's attached to.
   The most common and recommended way to attach an event listener is using the addEventListener() method:
   JavaScript
   ```
   element.addEventListener(event, function, useCapture);
   ```

   - element: The HTML element you want to listen for events on.

- event: A string representing the event type (e.g., 'click', 'mouseover', 'keydown').
- function: The function (often called a "callback function") that will be executed when the event occurs.
- useCapture (optional): A boolean value. If true, the event will be handled in the capturing phase; if false (default), it's handled in the bubbling phase. (More on this below in Event Propagation).

Example:

```
<button id="myButton">Click Me</button>
<p id="message"></p>
<script>
    const myButton = document.getElementById('myButton');
    const messageParagraph = document.getElementById('message');
    // Add a click event listener to the button
    myButton.addEventListener('click', function() {
        messageParagraph.textContent = 'Button was clicked!';
        messageParagraph.style.color = 'green';
    });
    // Another way to define the function (named function)
    function handleMouseOver() {
        myButton.style.backgroundColor = 'lightblue';
    }
    myButton.addEventListener('mouseover', handleMouseOver);
    function handleMouseOut() {
        myButton.style.backgroundColor = ''; // Reset background
    }
    myButton.addEventListener('mouseout', handleMouseOut);
</script>
```

Older/Less Recommended Ways:

- **Inline HTML attributes:** onclick="myFunction()" directly in the HTML tag. This mixes HTML and JavaScript and is generally discouraged for larger applications.
- **Direct property assignment:** element.onclick = function() { ... }. This works but only allows one handle r per event type; addEventListener allows multiple.

3. **The Event Object:**
   When an event occurs, the browser creates an event object and passes it as an argument to the event listener function. This object contains useful information about the event that just happened.
   Common properties of the event object:
   - event.type: The type of event (e.g., "click", "keydown").
   - event.target: The HTML element that triggered the event.

- event.currentTarget: The element that the event listener is attached to (can be different from target due to bubbling/capturing).
- event.clientX, event.clientY: Coordinates of the mouse pointer relative to the viewport (for mouse events).
- event.key, event.code: Which key was pressed (for keyboard events).
- event.preventDefault(): A method to stop the browser's default behavior for a given event (e.g., preventing a form from submitting or a link from navigating).
- event.stopPropagation(): A method to stop the event from propagating further up or down the DOM tree.

Example using the event object:

```
<a href="https://www.google.com" id="myLink">Go to Google</a>

<script>
  const myLink = document.getElementById('myLink');
  myLink.addEventListener('click', function(event) {
    event.preventDefault(); // Stop the link from navigating
    console.log('Link clicked!');
    console.log('Event type:', event.type);
    console.log('Target element:', event.target);
    alert('Navigation prevented! You clicked the link.');
  });
</script>
```

4. Event Propagation (Bubbling and Capturing):
   When an event occurs on an element, it doesn't just happen on that element. It propagates through the DOM tree. There are two phases:
   - **Capturing Phase (Trickling Down):** The event starts from the window object, then the document, and then travels down the DOM tree to the target element. Listeners attached with useCapture: true (or simply true as the third argument in addEventListener) will trigger during this phase.
   - **Bubbling Phase (Bubbling Up):** After reaching the target element, the event "bubbles up" from the target element, through its parent elements, all the way back up to the window object. This is the default behavior for addEventListener() (when useCapture is false or omitted).

Example

```
<div id="grandparent">
    <div id="parent">
        <button id="child">Click Me</button>
    </div>
</div>
<script>
    const grandparent = document.getElementById('grandparent');
    const parent = document.getElementById('parent');
    const child = document.getElementById('child');
    grandparent.addEventListener('click', function() {
        console.log('Grandparent clicked (Bubbling)');
    });
    parent.addEventListener('click', function() {
        console.log('Parent clicked (Bubbling)');
    });
    child.addEventListener('click', function() {
        console.log('Child clicked (Bubbling - Target)');
    });
    // Capturing phase listeners
    grandparent.addEventListener('click', function() {
        console.log('Grandparent clicked (Capturing)');
    }, true); // The 'true' here enables capturing
    parent.addEventListener('click', function() {
        console.log('Parent clicked (Capturing)');
    }, true);
    // If you click the "Child" button, the console output will be:
    // Grandparent clicked (Capturing)
    // Parent clicked (Capturing)
    // Child clicked (Bubbling - Target)
    // Parent clicked (Bubbling)
    // Grandparent clicked (Bubbling)
</script>
```

event.stopPropagation(): If you want to prevent an event from bubbling or capturing further, you can call event.stopPropagation() inside your event listener function.

5. Event Delegation:
   This is an advanced but very efficient technique. Instead of attaching a separate

event listener to every individual element in a large list (e.g., 100 list items), you attach a single event listener to their common parent. When an event bubbles up from a child element, the parent's listener catches it, and you can then use event.target to identify which specific child element triggered the event.
Example:

```
<ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
<script>
    const myList = document.getElementById('myList');
    myList.addEventListener('click', function(event) {
        // Check if the clicked element is an <li>
        if (event.target.tagName === 'LI') {
            alert('You clicked: ' + event.target.textContent);
        }
    });
</script>
```

This is more performant because you only have one listener to manage, rather than potentially hundreds.

In summary, JavaScript events are the backbone of interactive web pages. They provide a mechanism for your code to respond to user actions and browser changes, making websites dynamic, engaging, and user-friendly.