JavaScript Events

- Performing an action in response to some user interaction (a key press, a mouse click, a scroll, a window resize, etc.) is an extremely important task in a modern web app.

- JS allows us to respond to user interactions via ***events***.

- Specifically, JS allows us to register functions as handlers for specific events.
    - The handler function is called each time the associated event occurs.
        - Details about the event are provided via an argument to the function.

- The `addEventListener()` function registers a function (the second argument) to be called whenever a specific event (the first argument) occurs.
    - You can call `addEventListener()` on the window itself to listen for all events:

      ```
      window.addEventListener('click', function() {
          ...
      });
      ```

    - You can also call addEventListener() on a specific DOM element to listen only for events related to that element:

      ```
      var button = document.querySelector('.button');
      button.addEventListener('click', function() {
          ...
      });
      ```

- The `removeEventListener()` function removes a specific event handler. The arguments are similar to those of `addEventListener()`:

```
var button = document.querySelector('.button');
function callOnce() {
    console.log('Button clicked!');
```

```
    button.removeEventListener('click', callOnce);
}
button.addEventListener('click', callOnce);
```

  - The above event handler will only be called once.
  - The function being removed needs to be referred to by name, so we can't use an anonymous function as an event handler if we want to use `removeEventListener()` on that handler.

- Typically, when an element is removed from the DOM, we want to remove any event listeners registered to it. Otherwise, we may generate a memory leak.

- It is possible to assign event handlers to DOM elements by setting properties like `onclick`, `onsubmit`, etc., e.g.:

```
someElement.onclick = function (event) {...};
```

- This is a less desirable way to assign event handlers than using `addEventListener()`.
  - For example, only one handler can be assigned using a property like `onclick`, whereas with `addEventListener()`, we can assign many handlers for the same event to the same object.
  - In general, it is best to stick with `addEventListener()`.

Event Objects

- Every event handler function is passed an ***Event*** object as an argument. This object contains information about the specific event that triggered the handler.

- The Event objects associated with specific types of events have special properties specific to that event type.
  - e.g. the Event object for click events has a `button` property designating which mouse button was clicked.
  - We'll see some of these properties for specific event types below.

- Every Event object also has a `type` property specifying what kind of event was triggered, as a string (e.g. `'click'` or `'mousedown'`).

- Most Event objects also have a `target` property, which represents the specific DOM element on which the event was triggered:

```
var button = document.querySelector('.button');
button.addEventListener('click', function (event) {
     event.target.style['background-color'] = 'red';
});
```

- If an event handler is registered to an element with children, `target` will point to the specific child that triggered the event.
    - To access the element to which the handler was registered, you may use the `currentTarget` property.

Event propagation

- Events bubble up from children to parents and then to ancestors.
    - e.g. a click on a button will bubble up and trigger an event on the button's container, then on the button's container's container, and so forth, up through the `window`.
    - If ancestor elements have a handler registered for an event that occurs on a descendant element, the ancestor's handler will be triggered after the handler(s) of its descendants.
    - In all of these handlers, `event.target` will always point to the same element, the lowest element in the DOM that triggered the event.
    - In each handler, `event.currentTarget` will always point to the element on which the handler was registered.

- An event handler may call the `stopPropagation()` method on the `event` object to stop the event from continuing to bubble up:

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

Default actions

- Many DOM elements have default actions associated with them. For example, an `<a>` element's default click action is to open the associated link.

- In some situations, we will not want the default action to occur because we are assigning our own special action in an event listener. In these situations we can prevent the default action from occurring by calling `event.preventDefault()`.

- For example, to prevent the default action from occurring when some link is clicked, we could do this:

```
var someLink = document.querySelector('a.no-default');
someLink.addEventListener(function (event) {
    event.preventDefault();
    // Do some special event handling for this link.
});
```

Delegation

- In many cases, we will want to register essentially the same event handler on many similar DOM elements (e.g. a handler for the “like” button on all posts in a Facebook feed).

- In these cases, we could simply use a method like `querySelectorAll()` to find all of the elements to which we wanted to assign an event handler, loop through them, and assign the handler function to handle events on each one.

- Unfortunately, there are problems with this approach:
    - If a new element is added to the DOM that should have the same listener assigned to it, we need to remember to do that upon its addition to the DOM.
    - If an element with the event listener assigned to it is removed from the DOM, we need to remember to remove the event listener before the element is removed in order to avoid a memory leak.
    - If a page has too many event handlers all listening for events, this can result in a slowdown of the page, as the browser has to loop over many event handlers for each event triggered to see if they apply in the current case.
    - More event handlers being registered results in higher memory usage by the browser.

- To avoid these problems, we can use a technique called ***delegation***, where we listen for events at a *higher level* than the individual elements for which we'd like to handle events. Then, when the event is triggered, we rely on the fact that it propagates up the DOM tree and use `event.target` to determine which (if any) of the elements of interest actually triggered the event.

- For example, to listen for a click event on a set of 1000 similar buttons, we could assign an event listener to the first common ancestor of those buttons. Within that listener, we could check `event.target` to see if the click event originated from one of the buttons we were interested in and, if so, take the same action we would have taken if we'd been listening on that individual button.