Exploration — DOM Events

Introduction

Interactive web applications react to different types of user actions, such as the user moving a mouse or clicking it, or pressing a key on the keyboard. This interactivity in web applications is provided using the concept of **event-driven programming**. When a user takes an action, the browser dispatches an **event** based on the type of user action. We can use JavaScript to register functions called **event handlers**. When an event of particular type occurs, the browser executes the event handler registered for that type of event. We can program event handler functions so that they modify attributes of DOM nodes, modify the DOM tree or send HTTP requests to a webserver, thus providing interactivity in our application. We will study events and event handling in this exploration.

Events & Event Handling

Event handling requires specifying 3 things:

1. What happened, i.e., what is the event of interest?
2. Where did this event occur, i.e., what is the element of interest?
3. What to do, i.e., what JavaScript code to invoke when the event occurs on this element?

An event can be something that a user does or a browser does. A complete list of DOM events can be found at **W3 Schools website** (https://www.w3schools.com/jsref/dom_obj_event.asp). Some of the most important categories include the following:

- **Mouse-related events** (https://www.w3schools.com/jsref/obj_mouseevent.asp). Among others, these include:
  - `click` when the user clicks on an element,
  - `dblclick` when the user double clicks on an element,
  - `mousedown` when the user presses a mouse button on an element,
  - `mouseup` when the user releases a mouse button over an element, and
  - additional events that are raised when a mouse enters or leave an element.
- **Keyboard-related events** (https://www.w3schools.com/jsref/obj_keyboardevent.asp). These include:
  - `keydown` when a user is pressing a key,
  - `keypress` when a user has pressed the key,
  - `keyup` when a user has released the key.
- **Focus-related events** (https://www.w3schools.com/jsref/obj_focusevent.asp): Among others, these include:

  - `blur` when an element loses focus,
  - `focus` when an element gets focus.
- **Form submission event** (https://www.w3schools.com/jsref/event_onsubmit.asp) `submit` is raised when a form is submitted.
- **Input event** (https://www.w3schools.com/jsref/obj_inputevent.asp) `input` is raised when an element gets user input.
- Page related events such as
  - `load` raised for page load,
  - `unload` raised for page unload,
  - `pagehide` raised when the user moves away from a page, and
  - `pageshow` raised when the user navigates to a page.
- Timer events.

Registering an Event Handler

There are two ways to register an event handler to execute when an event occurs on an element.

1. Register the JavaScript code inline, as the value of an attribute with the name of the event. This is not recommended since it mixes JavaScript code with HTML elements.
2. Register the JavaScript code using the DOM API. There are 2 different syntax for this, one that uses the event handler property and the other that calls the `addEventListener` method.
   1. The name of the event handler property for an event has `on` prepended to the name of that event. E.g., we register the function `mouseClick` on the element `myElem` for the event `click` using the statement `myElem.onclick = mouseClick;`.
   2. The statement `myElem.addEventListener('click', mouseClick);` registers the same handler using the `addEventListener` method. This is the most recommended way of registering event handlers. If we call this function multiple times for the same element, we can register multiple functions that will all be called to handle an event.

Example: Registering an Event Handler

In the following HTML document, we register the function `mouseClick` as the event handler for the event `click` for two different elements. We register this function as the event handler on:

- the element with id `registerAsAttribute` by registering the JavaScript inline.
- the element with id `registerWithDOM` by registering the JavaScript code using the DOM API.

```
1  <!doctype html>
2 ▼<html>
3
4 ▼<head>
5      <title>Registering event
   listeners</title>
       <script>
           // The browser can pass the event
   to the event handler
6 ▼        function mouseClick(event) {
                                    use clicked on

                                    event.target
                                    dispatched
                                    e will discuss
                                    f
                                    next section.
                                    nt.target);
                                    is currently
```

Code Editor

This is where you write code. Code describes how programs work. The editor helps you write code, by coloring certain types, and giving you suggestions when you type. Click on one of the examples to see how code looks.

Next

Removing an Event Handler

We remove an event handler from an element by calling the method `removeEventListener` on this element. E.g., the following statement will remove the `mouseClick` event handler for the `click` event from the element `myElem`:

```
myElem.removeEventListener('click', mouseClick);
```

Example: Removing an Event Handler

In the following HTML document, we have two buttons.

- We register event handlers for both these buttons.
- Clicking on the button at the top changes the background of the span element which is right below this button.
- Clicking on the button at the bottom, removes the event handler for the button at the top.
- This means that once the bottom button has been clicked, clicking on the top button does not do anything and the background of the span element no longer changes.

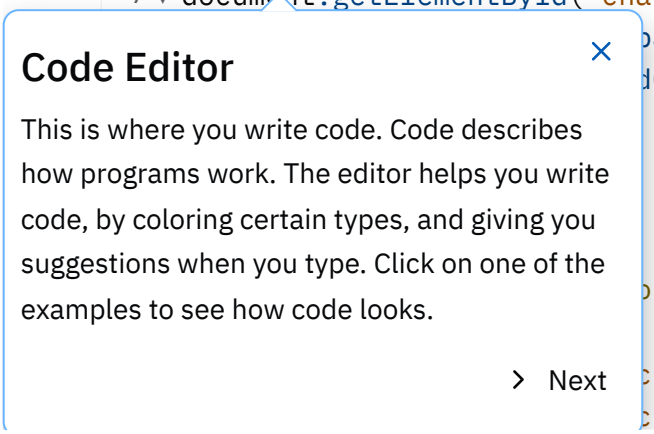
```
<!doctype html>
<html>

<head>
    <title>Removing an event
listener</title>
    <script>
        function changeColor() {
            let toUpdate =
document.getElementById('changeMe');
                                    backgroundColor
                                    dColor ===

                                     'red';


                                    on() {

                                    ckgroundChanger
                                    ck',
```



Event Object

When an event is raised, the browser can pass an object corresponding to this event to our handler. To do this we need to add an argument to our handler in its declaration. We actually did this in our previous example where we added an argument named `event` to our event handler `mouseClick`. **This event object (https://developer.mozilla.org/en-US/docs/Web/API/Event)** created by the browser has a very large number of properties and methods. Some of the most important properties of this object include the following:

- `type`: the name of the event, e.g., `click`, `mouseup`, etc.
- `timeStamp`: the time that the event was created.
- `target`: the element that dispatched the event.
- `currentTarget`: the element on which the event listener was registered. The difference between `target` and `currentTarget` will be explained when we discuss event propagation later in this exploration.

In addition to the general event, subclasses have been defined for specific types of events. For example, subclasses exist for **mouse events (https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent)**, **keyboard events (https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent)**, etc.

Load Events

One other important event type to know about are the load events. These events are called when a resource and all of its dependent resources have finished loading. When we want some JavaScript code to execute as soon as a resource is loaded, we can register a handler with that JavaScript code to the corresponding load event.

A common use case is when we want to manipulate a webpage as soon as it is loaded. One way to achieve this is to register the handler with the window's `load` event. You may have noticed this being done in some of our examples. However, in general, it is not a good idea to use this event if our goal is to manipulate the DOM. The reason is that the `load` event is raised only after everything in the page, including any large images, has been loaded. If our goal is to have the handler execute as soon as the page's DOM is parsed, we can instead use the event `DOMContentLoaded`, as show below:

```
<script>
  document.addEventListener("DOMContentLoaded", function(event) {
    console.log("DOM fully loaded and parsed");
  });
</script>
```

Timers

Timers (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Timeouts_and_intervals) provide another mechanism for event-driven programming in JavaScript. These timers are used in client-side code for animations or for automatic page refreshes.

- The function `setTimeout(myFunc, delay)` can be used to call another function after a specified number of milliseconds. We can pass arguments to the function being called by adding them after the delay time. For example:
  - `setTimeout(myFunc, 100)` will call `myFunc()` after 100 milliseconds.
  - `setTimeout(myFunc, 100, arg1, arg2)` will call `myFunc(arg1, arg2)` after 100 milliseconds.
- The function `setInterval(myfunc, interval)` can be used to keep calling another function after an interval of the specified number of milliseconds. This function returns an `intervalID` that uniquely identifies this timer. We can pass arguments to the function being called similarly to how it is done for `setTimeout`. For example,
  - `setInterval(myFunc, 100)` will keep calling `myFunc()` after every 100 milliseconds.
  - `setInterval(myFunc, 100, arg1, arg2)` will keep calling `myFunc(arg1, arg2)` after every 100 milliseconds.
- We can cancel a timer running at an interval by calling the function `clearInterval` and passing it the `intervalID` of the timer.

Exercise: Timers

In the following example, we use `setTimeout` to turn the background color of an element to red, 10 seconds after the DOM has been loaded. We also use `setInterval` to toggle the background color of another element after every 3 seconds. Your task: add code to this example so that 20 seconds after the DOM has been loaded, the toggling of the background color is permanently stopped.

```
1  <!doctype html>
2  <html>
3
4  <head>
5      <title>Use timers</title>
       <script>
6          function turnBackgroundRed() {
7
   document.getElementById("turnRed").style.b
   ackgroundColor = 'red';
```

Code Editor

This is where you write code. Code describes how programs work. The editor helps you write code, by coloring certain types, and giving you suggestions when you type. Click on one of the examples to see how code looks.

Next

Prevent Default Behavior (Optional)

We can call the method `preventDefault()` on an event to prevent the default action for the event from taking place. An example use case is calling this method on a `submit` event to prevent the submission of a form if the form validation fails, as shown in the following code snippet.

```
let form = document.getElementById('myForm');

form.onsubmit = function(event) {
  // Validate the form.
  if ( ...) {
    // If validation fails, prevent form submission
    event.preventDefault();

   // Show a message to the user
        ...
  }
}
```

MDN has a good example (https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault#stopping_keystrokes_from_reaching_an_edit_field) of using this method to prevent the user from entering invalid characters in an input field.

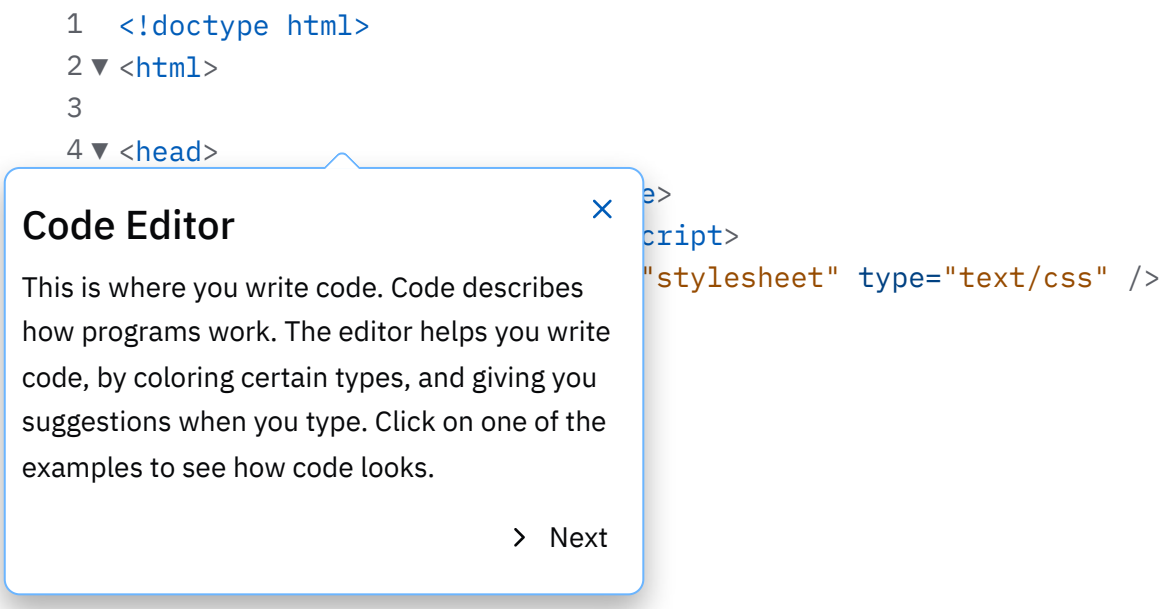
Event Propagation (Optional)

When an event is fired at an element that has parent elements, what happens if event handlers are defined at both the element and its parent? The default behavior is that the event **bubbles up** the DOM tree until it gets to the root element, i.e., the `html` element. Any event handlers defined for this event type along the path from this element to the root element will be executed for the event, starting with any event handler defined on the element, then any defined on the element's parent, then any defined on the element's grandparent, and so on. This is called the **bubbling phase**. The event handler on any element can stop an event from bubbling up the DOM tree by calling the `stopPropagation` method of the event in question.

Note that it is also possible to define event handlers which are executed on the DOM tree starting with the `html` element and going down the tree until the element on which the event occurred is reached. This is called the **capturing phase** and the event is said to **trickle-down** the DOM tree. Using `stopPropagation` stops an event from trickling down the DOM tree, similar to how this method stops an event from bubbling up. By default, event handlers are registered for the bubbling phase and registering handlers for the capturing phase is a lot less common.

Example: Prevent Default & Event Propagation (Optional)

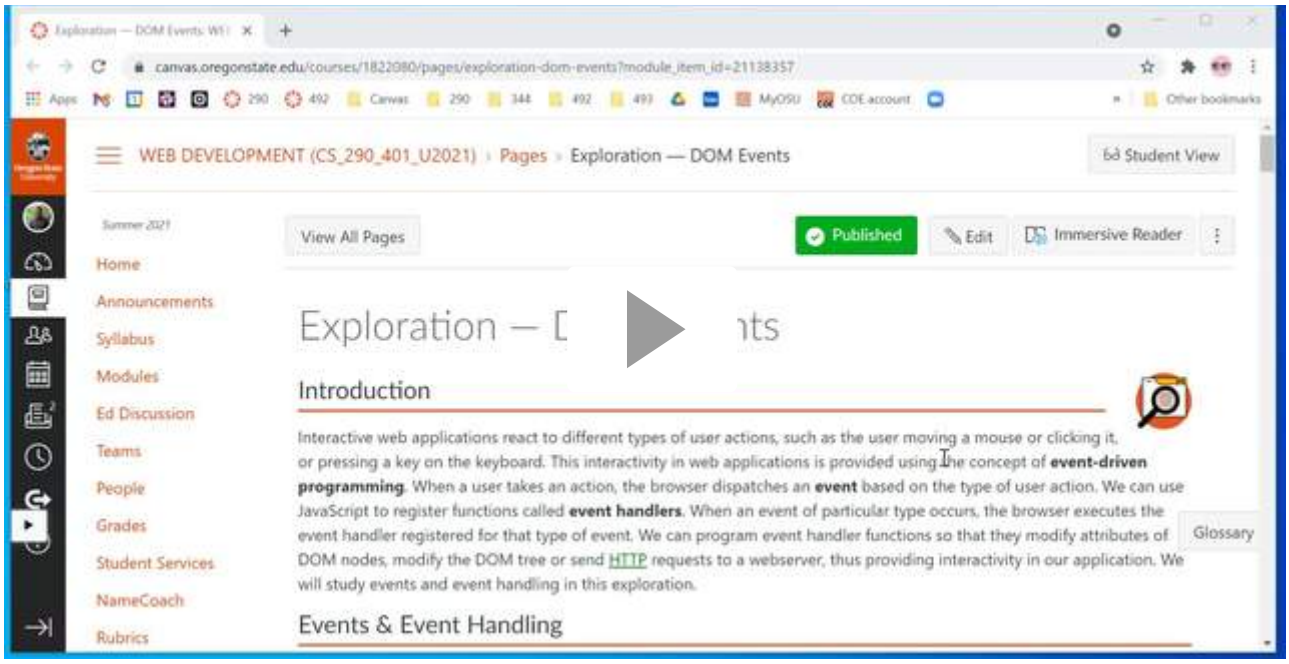
The following example shows the use of the methods `preventDefault` and `stopPropagation`.



In this example we use the checkboxes in the “Event Controls” form to control event handlers for the certain elements in the HTML page.

- The page has a button with id `nestedButton` which is inside a `div` element with id `buttonHolderDiv`.
  - The default behavior of clicking the `nestedButton` is that first the event handler at `nestedButton` is called and then the event bubbles up so that the event handler at `buttonHolderDiv` is called.
  - The behavior of clicking this button is controlled by the checkbox `stopPropagationCheckbox` and if we check `stopPropagationCheckbox`, then only the event handler on `nestedButton` is called.
  - This happens because the event handler on `nestedButton` examines the state of `stopPropagationCheckbox`, and if it is checked the event handler calls `stopPropagation()` to prevent the event from bubbling up to its parent `div` element.
- The page has a checkbox with id `defaultOrNotCheckbox`.
  - The default behavior of clicking a checkbox toggles its state, i.e.,
    - If the checkbox was unchecked, the click makes it checked, and
    - If the checkbox was checked, the click makes it unchecked.

- The behavior of this checkbox is controlled by the checkbox `preventDefaultCheckbox` and if we check `preventDefaultCheckbox`, then the state of `defaultOrNotCheckbox` cannot be modified.
- This happens because the event handler for clicks on `defaultOrNotCheckbox` examines the state of `preventDefaultCheckbox`, and if it is checked the event handler calls `preventDefault()` to prevent the default event handler to be invoked. Thus the state of the checkbox cannot be modified until `preventDefaultCheckbox` is unchecked.



0:00 / 16:19 1x

Summary

We have made it. We can now create dynamic interactive pages. Previously you may have been used to programs with very linear flow. Making the transition to event driven programming can be something of a strange shift from those programs. Make sure you know why events are useful, how they are triggered, how we can register handlers, and how events are sent to various handlers.

Additional Resources

- Events and event handling is discussed in **Chapter 15 (https://eloquentjavascript.net/15_event.html)** of the freely available book *Eloquent JavaScript*.
- An introduction to events and event handling is presented at MDN's **Introduction to events (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events)**.
- More discussion of bubbling and capturing phases can be **found here on MDN (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events#event_bubbling_and_capture)**.