Unit 2: HTML5, JQuery and Ajax Handling Events in JavaScript

An event is an action that occurs in the web browser, which the web browser feedbacks to you so that you can respond to it.

For example, when users click a button on a webpage, you may want to respond to this click event by displaying a dialog box.

Each event may have an event handler which is a block of code that will execute when the event occurs.

An event handler is also known as an event listener. It listens to the event and executes when the event occurs.

Suppose you have a button with the id btn:

```
<button id="btn">Click Me!</button>
```

To define the code that will be executed when the button is clicked, you need to register an event handler using the addEventListener() method:

```
let btn = document.querySelector('#btn');

function display() {

alert('It was clicked!');

}

btn.addEventListener('click',display);
```

How it works.

- First, select the button with the id btn by using the querySelector() method.
- Then, define a function called display() as an event handler.

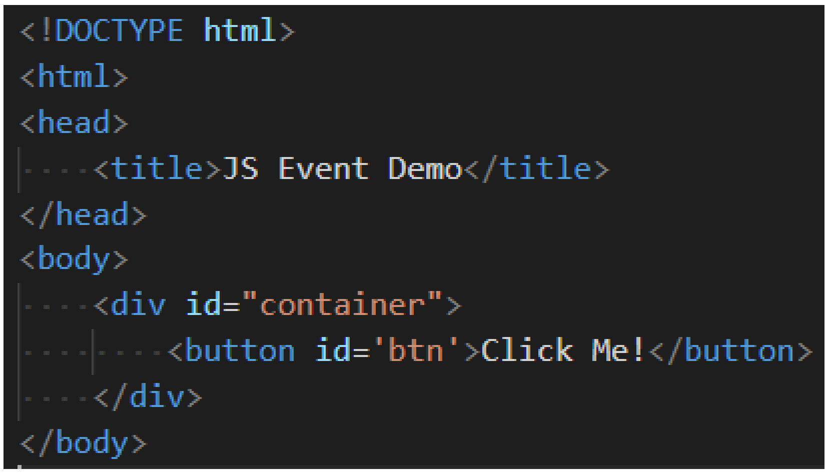
- Finally, register an event handler using the addEventListener() so that when users click the button, the display() function will be executed.

A shorter way to register an event handler is to place all code in an anonymous function, like this:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function() {

  alert('It was clicked!');

});
```

Event flow

Assuming that you have the following HTML document:



```
<!DOCTYPE html>
<html>
<head>
    <title>JS Event Demo</title>
</head>
<body>
    <div id="container">
        <button id='btn'>Click Me!</button>
    </div>
</body>
```

When you click the button, you're clicking not only the button but also the button's container, the div, and the whole webpage.

Event flow explains the order in which events are received on the page from the element where the event occurs and propagated through the DOM tree.

There are two main event models: **event bubbling** and **event capturing**.

Event bubbling

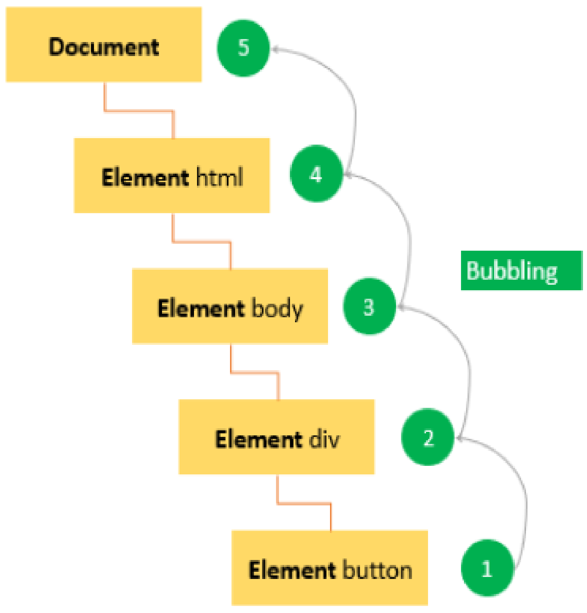
In the event bubbling model, an event starts at the most specific element and then flows upward toward the least specific element (the document or even window).

When you click the button, the click event occurs in the following order:

- button
- div with the id container
- body
- html
- document

The click event first occurs on the button which is the element that was clicked. Then the click event goes up the DOM tree, firing on each node along its way until it reaches the document object. Modern web browsers bubble the event up to the window object.

The following picture illustrates the event bubbling effect when users click the button:



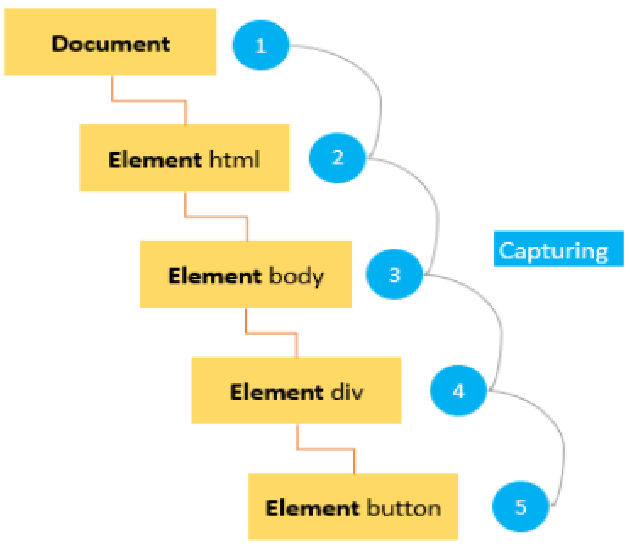
Event capturing

In the event capturing model, an event starts at the least specific element and flows downward toward the most specific element.

When you click the button, the click event occurs in the following order:

- document
- html
- body
- div with the id container
- button

The following picture illustrates the event capturing effect:

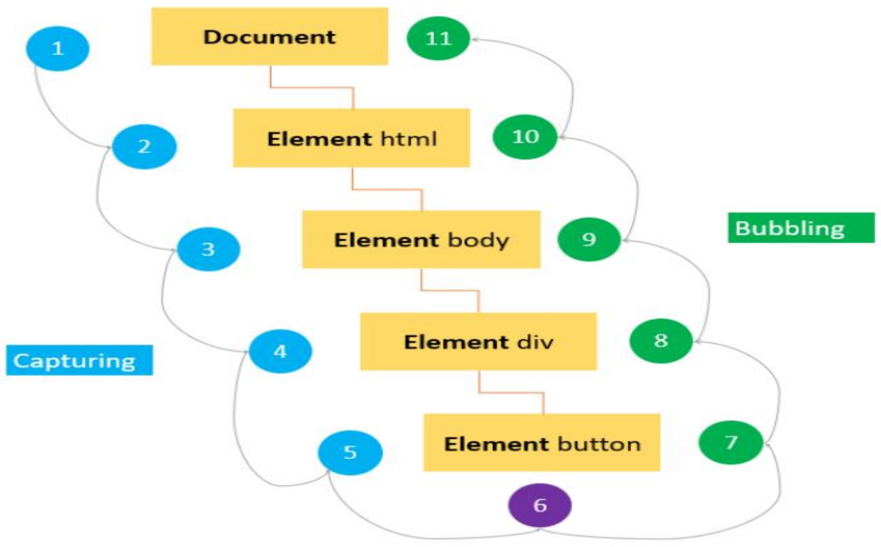


DOM Level 2 Event flow

DOM level 2 events specify that event flow has three phases:

1. First, event capturing occurs, which provides the opportunity to intercept the event.
2. Then, the actual target receives the event.
3. Finally, event bubbling occurs, which allows a final response to the event.

The following picture illustrates the DOM Level 2 event model when users click the button:



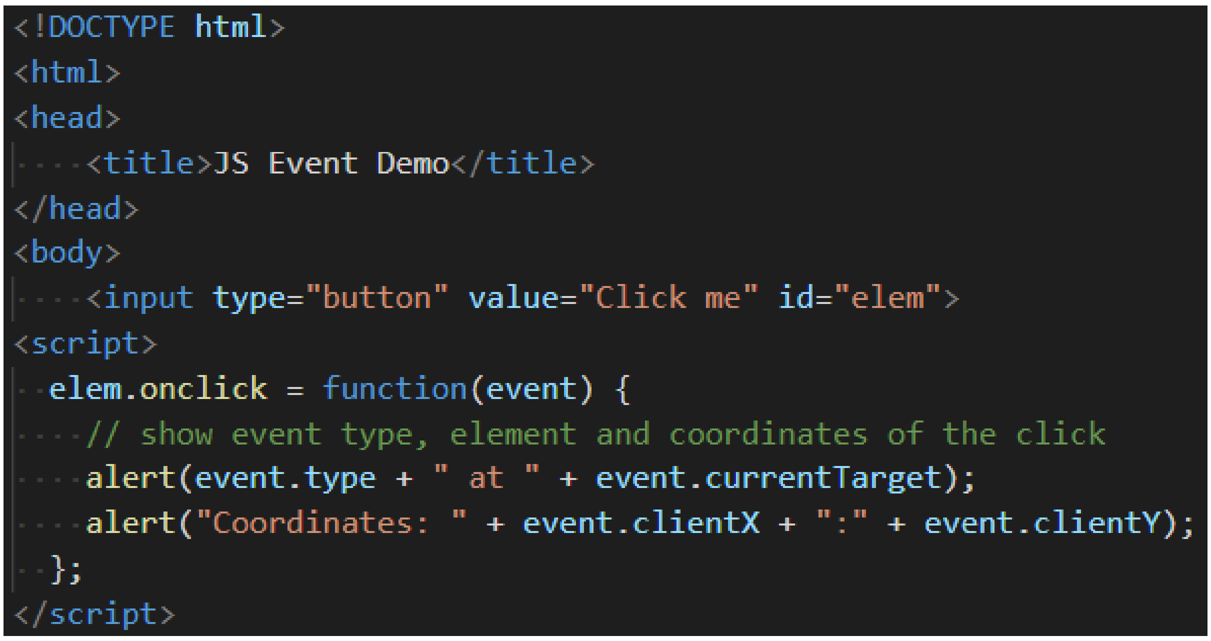
Event object

To handle an event properly we need want to know what's happened when an event occurs. It's not just a "click" or a "keydown", but what were the pointer coordinates at the time and which key was pressed.

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler. A parameter specified with a name such as **event, evt, or simply e**. This is called the **event object**, and it is automatically passed to event handlers to provide extra features and information.

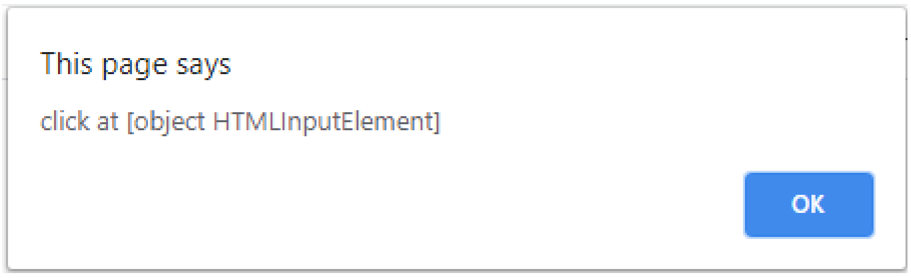
Here's an example of getting pointer coordinates from the event object:

CODE:



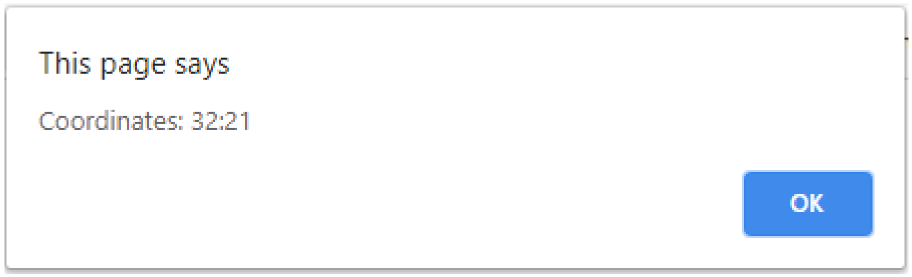
```
<!DOCTYPE html>
<html>
<head>
    <title>JS Event Demo</title>
</head>
<body>
    <input type="button" value="Click me" id="elem">
<script>
  elem.onclick = function(event) {
    // show event type, element and coordinates of the click
    alert(event.type + " at " + event.currentTarget);
    alert("Coordinates: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Results:



This page says

click at [object HTMLInputElement]



This page says

Coordinates: 32:21

Some properties of event object:

event.type

Event type, here it’s "click"

event.currentTarget

Element that handled the event. That’s exactly the same as this, unless the handler is an arrow function, or its this is bound to something else, then we can get the element from event.currentTarget.

event.clientX / event.clientY

Window-relative coordinates of the cursor, for pointer events. When the event occurs, the web browser passed an Event object to the event handler:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click', function(event) {

  console.log(event.type);

});
```

Output:

```
'click'
```

event.target

A handler on a parent element can always get the details about where it actually happened.

The most deeply nested element that caused the event is called a target element, accessible as event.target.

Note the differences from this (=event.currentTarget):

event.target – is the “target” element that initiated the event, it doesn’t change through the bubbling process.

this – is the “current” element, the one that has a currently running handler on it.

For instance, if we have a single handler form.onclick, then it can “catch” all clicks inside the form. No matter where the click happened, it bubbles up to <form> and runs the handler.

In form.onclick handler:

this (=event.currentTarget) is the <form> element, because the handler runs on it.

event.target is the actual element inside the form that was clicked.

The following table shows the most commonly-used properties and methods of the event object:

Property/Method	**Description**
bubbles	true if the event bubbles
cancelable	true if the default behavior of the event can be canceled
currentTarget	the current element on which the event is firing
defaultPrevented	return true if the preventDefault() has been called.
detail	more information about the event
eventPhase	1 for capturing phase, 2 for target, 3 for bubbling
preventDefault()	cancel the default behavior for the event. This method is only effective if the cancelable property is true
stopPropagation()	cancel any further event capturing or bubbling. This method only can be used if the bubbles property is true.
target	the target element of the event
type	the type of event that was fired

The event object is only accessible inside the event handler. Once all the event handlers have been executed, the event object is automatically destroyed.

preventDefault()

To prevent the default behavior of an event, you use the preventDefault() method.

For example, when you click a link, the browser navigates you to the URL specified in the href attribute:

```
<a href="https://www.google.com/">Google</a>
```

However, you can prevent this behaviour by using the preventDefault() method of the event object:

```
let link = document.querySelector('a');

link.addEventListener('click',function(event) {

  console.log('clicked');

  event.preventDefault();

});
```

Note:

The preventDefault() method does not stop the event from bubbling up the DOM. And an event can be canceled when its cancelable property is true.

stopPropagation()

The stopPropagation() method immediately stops the flow of an event through the DOM tree. However, it does not stop the browers default behavior.

Example:

```
let btn = document.querySelector('#btn');
btn.addEventListener('click', function(event) {
console.log('The button was clicked!');
 event.stopPropagation();
});
document.body.addEventListener('click',function(event) {
   console.log('The body was clicked!');
});
```

Without the stopPropagation() method, you would see two messages on the Console window.

The click event never reaches the body because the stopPropagation() was called on the click event handler of the button.

When an event occurs, you can create an event handler which is a piece of code that will execute to respond to that event. An event handler is also known as an event listener. It listens to the event and responds accordingly to the event fires.

An event listener is a function with an explicit name if it is reusable or an anonymous function in case it is used one time.

An event can be handled by one or multiple event handlers. If an event has multiple event handlers, all the event handlers will be executed when the event is fired.

There are three ways to assign event handlers.

1) HTML event handler attributes

Event handlers typically have names that begin with on, for example, the event handler for the click event is onclick.

To assign an event handler to an event associated with an HTML element, you can use an HTML attribute with the name of the event handler. For example, to execute some code when a button is clicked, you use the following:

```
<input type="button" value="Save" onclick="alert('Clicked!')">
```

In this case, when the button is clicked, the alert box is shown.

When you assign JavaScript code as the value of the onclick attribute, you need to escape the HTML characters such as ampersand (&), double quotes ("), less than (<), etc., or you will get a syntax error.

An event handler defined in the HTML can call a function defined in a script. For example:

```
<script>
function showAlert() {
alert('Clicked!');}
</script>
<input type="button" value="Save" onclick="showAlert()">
```

In this example, the button calls the showAlert() function when it is clicked.The showAlert() is a function defined in a separate <script> element, and could be placed in an external JavaScript file.

NOTE:

The following are some important points when you use the event handlers as attributes of the HTML element:

First, the code in the event handler can access the event object without explicitly defining it:

```
<input type="button" value="Save" onclick="alert(event.type)">
```

Second, the this value inside the event handler is equivalent to the event's target element:

```
<input type="button" value="Save" onclick="alert(this.value)">
```

Third, the event handler can access the element's properties, for example:

```
<input type="button" value="Save" onclick="alert(value)">
```

Disadvantages of using HTML event handler attributes

Assigning event handlers using HTML event handler attributes are considered as bad practices and should be avoided as much as possible because of the following reasons:

1. First, the event handler code is mixed with the HTML code, which will make the code more difficult to maintain and extend.
2. Second, it is a timing issue. If the element is loaded fully before the JavaScript code, users can start interacting with the element on the webpage which will cause an error.

For example, suppose that the following showAlert() function is defined in an external JavaScript file:

```
<input type="button" value="Save" onclick="showAlert()">
```

And when the page is loaded fully and the JavaScript has not been loaded, the showAlert() function is undefined. If users click the button at this moment, an error will occur.

2) DOM Level 0 event handlers

Each element has event handler properties such as onclick. To assign an event handler, you set the property to a function as shown in the example:

```
let btn = document.querySelector('#btn');

btn.onclick = function() {

alert('Clicked!');

};
```

In this case, the anonymous function becomes the method of the button element. Therefore, the this value is equivalent to the element. And you can access the element’s properties inside the event handler:

```
let btn = document.querySelector('#btn');

btn.onclick = function() {

 alert(this.id);

};
```

Output:

```
btn
```

By using the this value inside the event handler, you can access the element’s properties and methods.

To remove the event handler, you set the value of the event handler property to null:

```
btn.onclick = null;
```

The DOM Level 0 event handlers are still being used widely because of its simplicity and cross-browser support.

3) DOM Level 2 event handlers

DOM Level 2 Event Handlers provide two main methods for dealing with the registering/deregistering event listeners:

addEventListener() – register an event handler

removeEventListener() – remove an event handler

These methods are available in all DOM nodes.

The addEventListener() method

The addEventListener() method accepts three arguments: an event name, an event handler function, and a Boolean value that instructs the method to call the event handler during the capture phase (true) or during the bubble phase (false). For example:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function(event) {

   alert(event.type); // click

});
```

It is possible to add multiple event handlers to handle a single event, like this:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function(event) {

   alert(event.type); // click

});

btn.addEventListener('click',function(event) {

   alert('Clicked!');
});
```

The removeEventListener() method

The removeEventListener() removes an event listener that was added via the addEventListener(). However, you need to pass the same arguments as were passed to the addEventListener(). For example:

```
let btn = document.querySelector('#btn');

        // add the event listener

let showAlert = function() {

   alert('Clicked!');

};

btn.addEventListener('click', showAlert);

        // remove the event listener

btn.removeEventListener('click', showAlert);
```

Using an anonymous event listener as the following will not work:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',function() {

  alert('Clicked!');

});     // won't work

btn.removeEventListener('click', function() {

  alert('Clicked!');

});
```

Accessing the element: this

The value of this inside a handler is the element. The one which has the handler on it. In the code below button shows its contents using this.innerHTML:

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

We can set an existing function as a handler, the function should be assigned as sayThanks, not sayThanks().

If we add parentheses, then sayThanks() becomes is a function call. So the last line actually takes the result of the function execution, that is undefined (as the function returns nothing), and assigns it to onclick. That doesn't work.

Don't use setAttribute for handlers.

Example:

// a click on <body> will generate errors,

// because attributes are always strings, function becomes a string

```
document.body.setAttribute('onclick', function() { alert(1) });
```

JavaScript Event Delegation

Suppose that you have the following menu:

```
<ul id="menu">
        <li><a id="company"> company </a></li>
        <li><a id="Employee"> Employee </a></li>
        <li><a id="report">report</a></li>
</ul>
```

To handle the click event of each menu item, you may add the corresponding click event handlers:

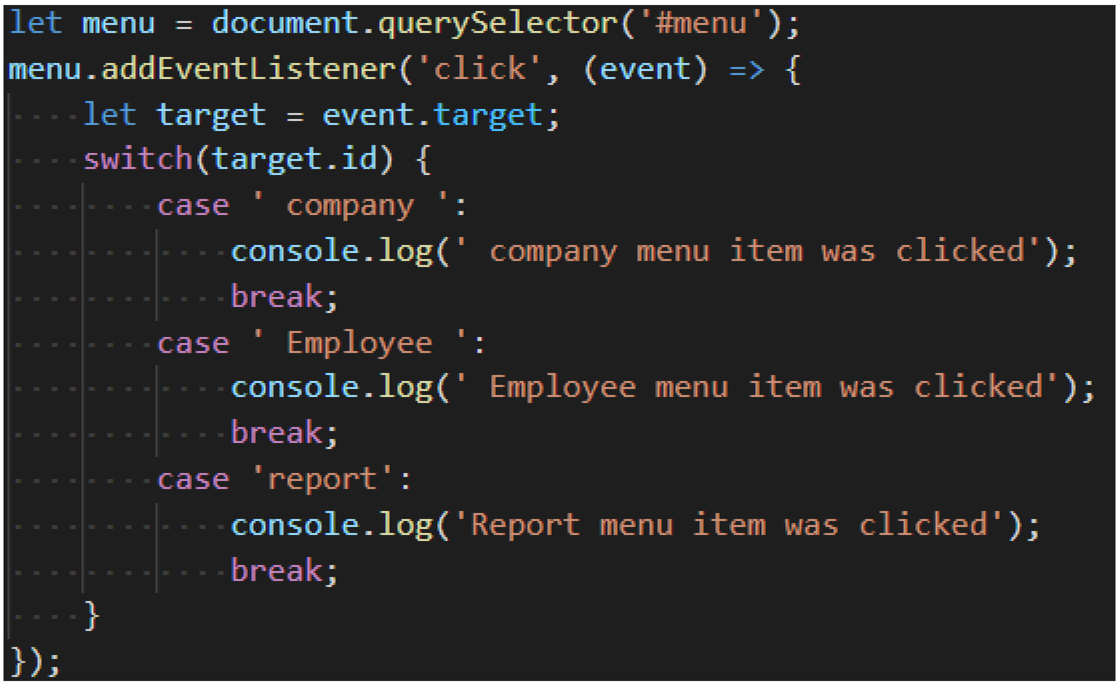
```
let company = document.querySelector('# company’);

company.addEventListener(company,(event) => {

console.log(‘company menu item was clicked');

});

let Employee = document.querySelector('# Employee);

Employee.addEventListener(' Employee ',(event) => {

console.log(‘Employee menu item was clicked');

});

let report = document.querySelector('#report');

report.addEventListener('report',(event) => {

console.log('Report menu item was clicked');

});
```

In JavaScript, if you have a large number of event handlers on a page, these event handlers will directly impact the performance because of the following reasons:

- First, each event handler is a function which is also an object that takes up memory. The more objects in the memory, the slower the performance.
- Second, it takes time to assign all the event handlers, which causes a delay in the interactivity of the page.

To solve this issue, you can leverage the event bubbling.

Instead of having multiple event handlers, you can assign a single event handler to handle all the click events:



```
let menu = document.querySelector('#menu');
menu.addEventListener('click', (event) => {
    let target = event.target;
    switch(target.id) {
        case ' company ':
            console.log(' company menu item was clicked');
            break;
        case ' Employee ':
            console.log(' Employee menu item was clicked');
            break;
        case 'report':
            console.log('Report menu item was clicked');
            break;
    }
});
```

Explanation:

When you click any <a> element inside the <ul> element with the id menu, the click event bubbles to the parent element which is the <ul> element. So instead of handling the click event of the individual <a> element, you can capture the click event at the parent element.

In the click event listener, you can access the target property which references the element that dispatches the event. To get the id of the element that the event actually fires, you use the target.id property.

Once having the id of the element that fires the click event, you can have code that handles the event correspondingly.

The way that we handle the too-many-event-handlers problem is called the **event delegation**.

The event delegation refers to the technique of levering event bubbling to handle events at a higher level in the DOM than the element on which the event originated.

JavaScript event delegation benefits

When it is possible, you can have a single event handler on the document that will handle all the events of a particular type. By doing this, you gain the following benefits:

- Less memory usage, better performance.
- Less time required to set up event handlers on the page.
- The document object is available immediately. As long as the element is rendered, it can start functioning correctly without delay. You don't need to wait for the DOMContentLoaded or load events.

Delegation limitations:

- First, the event must be bubbling. Some events do not bubble. Also, low-level handlers should not use event.stopPropagation().
- Second, the delegation may add CPU load, because the container-level handler reacts on events in any place of the container, no matter whether they interest us or not. But usually the load is negligible, so we don't take it into account.

JavaScript custom events

The following function highlights an element by changing its background color to yellow:

```
function highlight(elem) {

    const bgColor = 'yellow';

   elem.style.backgroundColor = bgColor;

}
```

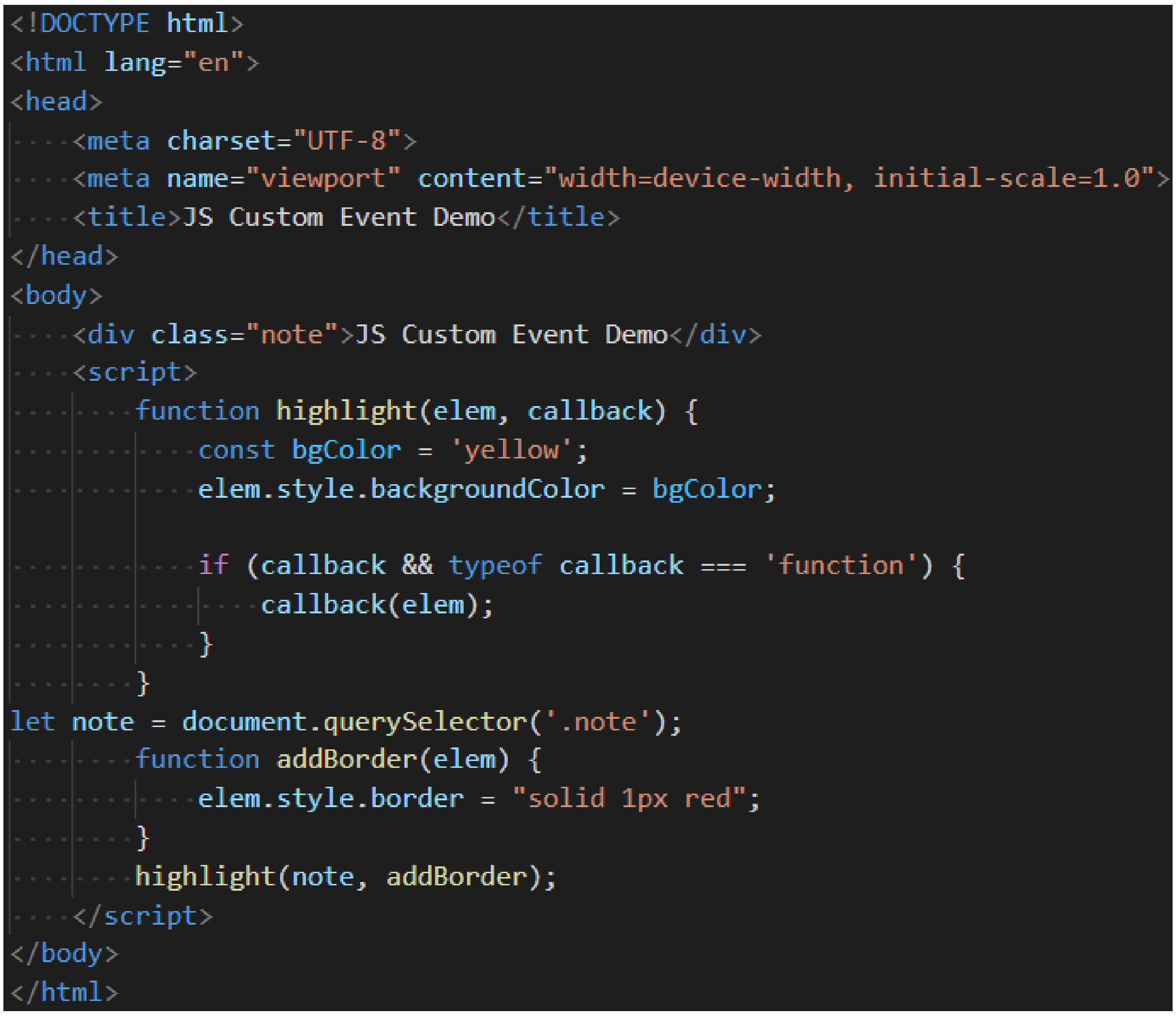
To execute a piece of code after highlighting the element, you may come up with a callback:

```
function highlight(elem, callback) {
```

```
const bgColor = 'yellow';

elem.style.backgroundColor = bgColor;

 if(callback && typeof callback === 'function') {

 callback(elem);

  }

   }
```

The following calls the highlight() function and adds a border to a <div> element:

CODE :



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JS Custom Event Demo</title>
</head>
<body>
    <div class="note">JS Custom Event Demo</div>
    <script>
        function highlight(elem, callback) {
            const bgColor = 'yellow';
            elem.style.backgroundColor = bgColor;

            if (callback && typeof callback === 'function') {
                callback(elem);
            }
        }
let note = document.querySelector('.note');
        function addBorder(elem) {
            elem.style.border = "solid 1px red";
        }
        highlight(note, addBorder);
    </script>
</body>
</html>
```

Results:

JS Custom Event Demo

To make the code more flexible, you can use the custom event.

Creating JavaScript custom events

To create a custom event, you use the CustomEvent() constructor:

let event = new CustomEvent(eventType, options);

The CustomEvent() has two parameters:

- The eventType is a string that represents the name of the event.
- The options is an object has the detail property that contains any custom information about the event.

The following example shows how to create a new custom event called highlight:

let event = new CustomEvent('highlight', {

detail: {backgroundColor: 'yellow'}});

Dispatching JavaScript custom events

After creating a custom event, you need to attach the event to an element and trigger it by using the dispatchEvent() method:

element.dispatchEvent(event);

NOTE :

- Use the CustomEvent() constructor to create a custom event and dispatchEvent() to trigger the event.
- The custom events allow you to decouple the code that you want to execute after another piece of code completes.
- For example, you can separate the event listeners in a separate script. In addition, you can have multiple event listeners to the same custom event.

JavaScript page load events

When you open a page, the following events occur in sequence:

- ***DOMContentLoaded*** – the browser fully loaded HTML and completed building the DOM tree. However, it hasn't loaded external resources like stylesheets and images. In this event, you can start selecting DOM nodes or initialize the interface.
- ***load*** – the browser fully loaded the HTML and also external resources like images and stylesheets.

When you leave the page, the following events fire in sequence:

- ***beforeunload*** – fires before the page and resources are unloaded. You can use this event to show a confirmation dialog to confirm if you really want to leave the page. By doing this, you can prevent data loss in case you are filling out a form and accidentally click a link to navigate to another page.
- ***unload*** – fires when the page has completely unloaded. You can use this event to send the analytic data or to clean up resources.

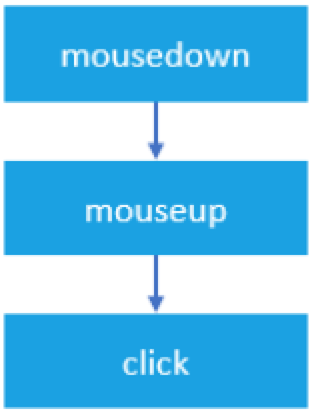
JavaScript mouse events

Mouse events fire when you use the mouse to interact with the elements on the page. DOM Level 3 events define nine mouse events.

mousedown, mouseup, and click

When you click an element, there are no less than three mouse events fire in the following sequence:

- The mousedown fires when you depress the mouse button on the element.
- The mouseup fires when you release the mouse button on the element.
- The click fires when one mousedown and one mouseup detected on the element.



If you depress the mouse button on an element and move your mouse off the element, and then release the mouse button. The only mousedown event fires on the element.

Likewise, if you depress the mouse button, and move the mouse over the element, and release the mouse button, the only mouseup event fires on the element.

In both cases, the click event never fires.

dblclick

In practice, you rarely use the dblclick event. The dblclick event fires when you double click over an element.

It takes two click events to cause a dblclick event to fire. The dblclick event has four events fired in the following order:

- mousedown
- mouseup
- click
- mousedown
- mouseup
- click
- dblclick

As you can see, the click events always take place before the dblclick event. If you register both click and dblclick event handlers on the same element, you will not know exactly what user actually has clicked or double-clicked the element.

mousemove

The mousemove event fires repeatedly when you move the mouse cursor around an element. Even when you move the mouse one pixel, the mousemove event still fires. It will cause the page slow, therefore, you only register mousemove event handler only when you need it and immediately remove the event handler as soon as it is no longer used, like this:

```
element.onmousemove = mouseMoveEventHandler;

// ...

//  later, no longer use

element.onmousemove = null;
```

mouseover / mouseout

The *mouseover* fires when the mouse cursor is outside of the element and then move to inside the boundaries of the element.

The *mouseout* fires when the mouse cursor is over an element and then move another element.

mouseenter / mouseleave

The *mouseenter* fires when the mouse cursor is outside of an element and then moves to inside the boundaries of the element.

The *mouseleave* fires when the mouse cursor is over an element and then moves to the outside of the element’s boundaries.

Both mouseenter and mouseleave does not bubble and does not fire when the mouse cursor moves over descendant elements.

Registering mouse event handlers

To register a mouse event, you use these steps:

First, select the element by using querySelector() or getElementById() method.

Then, register the mouse event using the addEventListener() method.

For example, suppose that you have the following button:

```
<button id="btn">Click Me!</button>
```

To register a mouse click event handler, you use the following code:

```
let btn = document.querySelector('#btn');

btn.addEventListener('click',(event) => {

  console.log('clicked');});
```

or

you can assign a mouse event handler to the element's property:

```
let btn = document.querySelector('#btn');

btn.onclick = (event) => {

console.log('clicked');};
```

In legacy systems, you may find that the event handler is assigned in the HTML attribute of the element:

```
<button id="btn" onclick="console.log('clicked')">Click Me!</button>
```

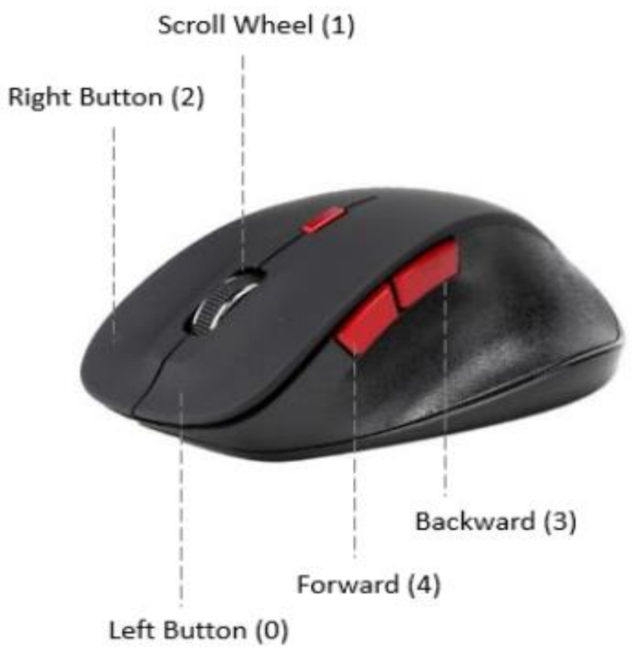
It's a good to always use the addEventListener() to register a mouse event handler.

Detecting mouse buttons

The event object passed to the mouse event handler has a property called button that indicates which mouse button was pressed on the mouse to trigger the event.

The mouse button is represented by a number:

- 0: the main mouse button pressed, usually the left button.
- 1: the auxiliary button pressed, usually the middle button or the wheel button.
- 2: the secondary button pressed, usually the right button.
- 3: the fourth button pressed, usually the Browser Back button.
- 4: the fifth button pressed, usually the Browser Forward button.



Modifier keys

When you click an element, you may press one or more modifier keys: ***Shift, Ctrl, Alt, and Meta.***

Esc
Tab
Caps
Shift
Fn
Ctrl
Alt

Note:

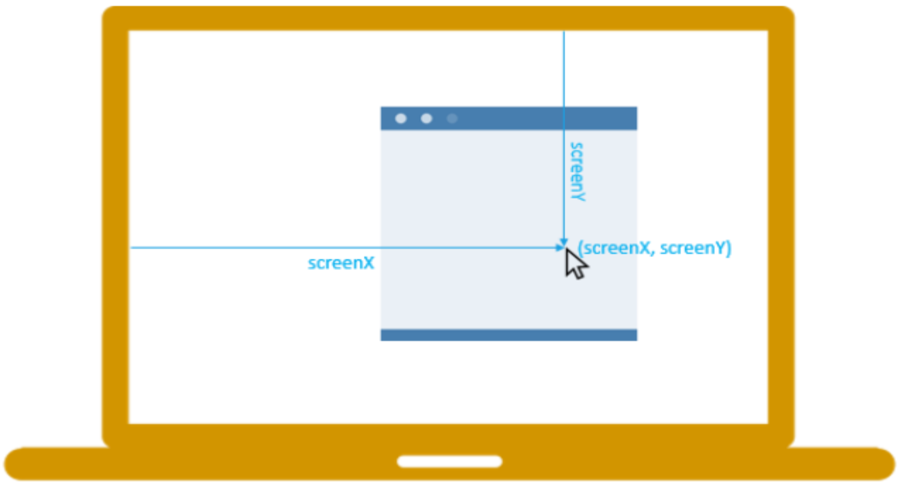
The Meta key is the Windows key on Windows keyboards and the Command key on Apple keyboard.

To detect if these modifier keys have been pressed, you can use the event object passed to the mouse event handler.

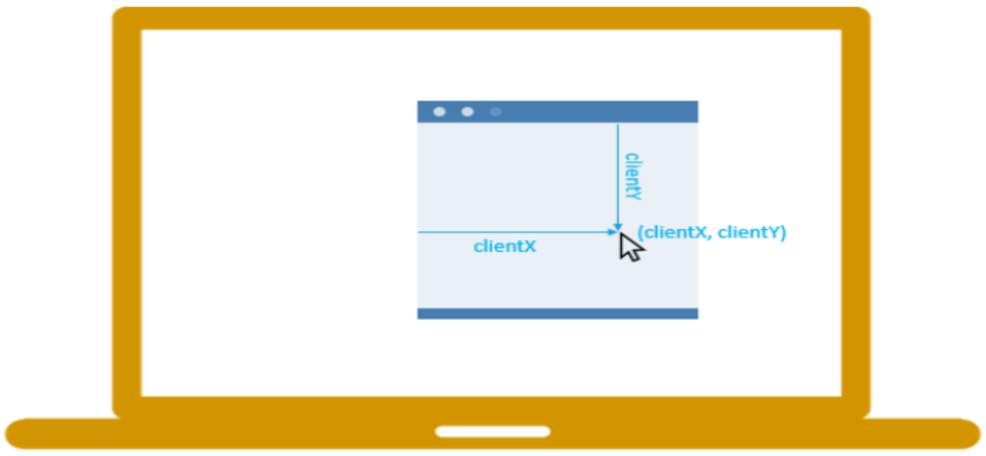
The event object has four Boolean properties, where each is set to true if the key is being held down or false if the key is not pressed.

Screen Coordinates

The *screenX* and *screenY* properties of the event passed to the mouse event handler return the screen coordinates of the location of the mouse in relation to the entire screen.



On the other hand, the *clientX* and *clientY* properties provide the horizontal and vertical coordinates within the application's client area at which the mouse event occurred:



NOTES:

- Use addEventListener() method to register a mouse event handler.
- The event.button indicates which mouse button was pressed to trigger the mouse event.
- The modifier keys: alt, shift, ctrl, and meta (Mac) can be obtained via properties of the event object passed to the mouse event handler.
- The screenX and screenY properties return the horizontal and vertical coordinates of the mouse pointer in screen coordinates.
- The clientX and clientY properties of the event object return horizontal and vertical coordinates within the application's client area at which the mouse event occurred.

keyboard events

When you interact with the keyboard, the keyboard events are fired. There are three main keyboard events:

- ***keydown*** – fires when you press a key on the keyboard and it fires repeatedly while you holding down the key.

- ***keyup*** – fires when you release a key on the keyboard.
- ***keypress*** – fires when you press a character keyboard like a,b, or c, not the left arrow key, home, or end keyboard, … The keypress also fires repeatedly while you hold down the key on the keyboard.

The keyboard events typically fire on the text box, through all elements support them.

When you press a character key once on the keyboard, three keyboard events are fired in the following order:

- keydown
- keypress
- keyup

Both keydown and keypress events are fired before any change made to the text box, whereas the keyup event fires after the changes have made to the text box. If you hold down a character key, the keydown and keypress are fired repeatedly until you release the key.

When you press a non-character key, the keydown event is fired first followed by the keyup event. If you hold down the non-character key, the keydown is fired repeatedly until you release the key.

Handling keyboard events

To handle a keyboard event, you follow these steps:

First, select the element on which the keyboard event will fire. Typically, it is a text box.

Then, use the element.addEventListener() to register an event handler.

Suppose that you have the following text box with the id message:

```
<input type="text" id="message">
```

The following example illustrates how to register keyboard event listeners:

```
let msg = document.getDocumentById('#message');

msg.addEventListener("keydown", (event) => {

// handle keydown });

msg.addEventListener("keypress", (event) => {

// handle keypress });

msg.addEventListener("keyup", (event) => {

// handle keyup });
```

If you press a character key, all three event handlers will be called.

The keyboard event properties

The keyboard event has two important properties: key and code. The key property returns the character that has been pressed whereas the code property returns the physical key code.

For example, if you press the z character key, the event.key returns z and event.code returns KeyZ.

NOTE :

- When you press a character key on the keyboard, the keydown, keypress, and keyup events are fired sequentially. However, if you press a non-character key, only the keydown and keyup events are fired.
- The keyboard event object has two important properties: key and code properties that allow you to detect which key has been pressed.
- The key property returns the value of the key pressed while the code represents a physical key on the keyboard.

JavaScript focus events

The focus events fire when an element receives or loses focus. These are the two main focus events:

- ***focus*** fires when an element has received focus.
- ***blurfires*** when an element has lost focus.
- The ***focusin*** and ***focusout*** fire at the same time as focus and blur, however, they bubble while the focus and blur do not.

The following elements are focusable:

- The window gains focus when you bring it forward by using Alt+Tab or clicking on it and loses focus when you send it back.
- Links when you use a mouse or a keyboard.
- Form fields like input text when you use a keyboard or a mouse.
- Elements with tabindex, also when you use a keyboard or a mouse