Events In JavaScript

What is an event?

Events are things that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs, and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs. Events are fired inside the browser window, and tend to be attached to a specific item that resides in it. This might be a single element, a set of elements, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events that can occur.

For example:

- The user selects, clicks, or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or ends.
- An error occurs.

JavaScript Event Types

These are the top 8 types of JavaScript Event discussed below:

1. User Interface events

These occur as the result of any interaction with the browser window rather than the HTML page. In these events, we attach the event listener to the window object, not the document object. The various UI events are as follows.

- Load : The load event fires when the webpage finishes loading. It can also fire on nodes of elements like *images, scripts, or objects.*
- Unload : This event fires before the users leave the page, i.e., the webpage is unloading. Page unloading usually happens because a new page has been requested.
- Error : This event fires when the browser encounters a **JavaScript error** or an asset that doesn't exist.
- Resize : It fires when we resize the browser window. But browsers repeatedly fire this event, so avoid using this event to trigger complicated code; it might make the page less responsive.

- Scroll : This event fires when the user scrolls up/down on the browser window. It can relate to the entire page or a specific element on the page.

2. Focus and blur events

These events fire when the HTML elements you can interact with gain/ lose focus. They are most commonly used in forms and especially helpful when you want to do the following tasks:

- To show tips or feedback to users as they interact with an element within a form. The tips are usually shown in the elements other than the one the user is interacting with.
- To trigger form validation as a user moves from one control to the next without waiting to submit the form.

The different focus and blur events are as follows:

- Focus : This event fires, for a specific DOM node, when an element gains focus.
- Blur : This fires, for a specific DOM node, when an element loses focus.
- Focusin : This event is the same as the focus event. But Firefox doesn't yet support the focusin event.
- Focusout : This is the same event as the blur event. This is a new event type in JavaScript, thus not supported in Firefox right now.

3. Mouse events

These events fire when the mouse moves or the user clicks a button. All the elements of the page support these events and use the bubbling approach. These actions work differently on touchscreen devices. Preventing the default behavior of mouse events can cause unexpected results. The various mouse events of JavaScript are as follows:

- Click : This event fires when the user clicks on the primary mouse button (usually the left button). This event also fires if the user presses the Enter key on the keyboard when an element has focus.

  Touch-screen: A tap on the screen acts like a single left-click.

- Dblclick : This event fires when the user clicks the primary mouse button, in quick succession, twice.

  Touch-screen: A double-tap on the screen acts like a double left-click.

  Accessibility: You can add the above two events to any element, but it's better to apply it only on the items that are usually clicked, or it will not be accessible through keyboard navigation. All the mouse events discussed below cannot be triggered by the keyboard.

- Mousedown : It fires when the user clicks down on any mouse button.

  Touch-screen: You can use the **touchstart** event.

- Mouseup : It fires when the user releases a mouse button.

  Touch-screen: You can use the **touchend** event.

  We have separate **mousedown** and **mouseup** events to add drag-and-drop functionality or controls in game development. Don't forget a **click** event is the combination of **mousedown** and **mouseup** events.

- Mouseover : It fires when the user moves the cursor, which was outside an element before, inside the element. We can say that it fires when we move the cursor over the element.

- Mouseout : It fires when the user moves the cursor, which was inside an element before, outside the element. We can say that it fires when the cursor moves off the element.

  The mouseover and mouseout events usually change the appearance of graphics on our webpage. A prefered alternative to this is to use the **CSS: hover** pseudo-class.

- Mousemove : It fires when the user moves the cursor around the element. This event is frequently triggered.

4. Keyboard events

These events fire on any kind of device when a user interacts with a keyboard.

- Input : This event fires when the value of an **<input>** or a **<textarea>** changes (doesn't fire for deleting in IE9). You can use keydown as a fallback in older browsers.

- Keydown : It fires when the user presses any key in the keyboard. If the user holds down the key, this event fires repeatedly.

- Keypress : It fires when the user presses a key that results in printing a character on the screen. This event fires repeatedly if the user holds down the key. This event will not fire for the enter, tab, or arrow keys; the **keydown** event would.

- Keyup : The keyup event fires when the user releases a key on the keyboard.

  The keydown and keypress events fire before a character appears on the screen, the keyup fires after it shows.

  To know the key pressed when you use the keydown and keypress events, the event object has a keyCode property. This property, instead of returning the letter for that key, returns the ASCII code of the lowercase for that key.

5. Form events

These events are common while using forms on a webpage. In particular, we see the submit event mostly in form of validation (checking form values). If the users miss any required information or enter incorrect input, validation before sending the data to the server is faster. The list below explains the different **form of** events available to the user.

- Submit : This event fires on the node representing the **<form>** element when a user submits a form.
- Change : It fires when the status of various form elements change. This is a better option than using the click event because clicking is not the only way users interact with the form.
- Input : The input event is very common with the <input> and the <textarea> elements.

  We often use the **focus** and **blur** events with forms, but they are also available in conjunction with other elements like **links.**

6. Mutation events and observers

Whenever the structure of the DOM tree changes, it triggers a **mutation event**. The change in the tree may be due to the addition or removal of a DOM node through your script. But these have an alternative that will replace them: **mutation observers**. The following are the numerous mutation events in JavaScript.

- DOMNodeInserted : It fires when the script inserts a new node in the DOM tree using *appendChild(), replaceChild(), insertBefore(), etc.*
- DOMNodeRemoved : This event fires when the script removes an existing node from the tree using *removeChild(), replaceChild(), etc.*
- DOMSubtreeModified : It fires when the structure of the DOM tree changes i.e. the above two events occur.
- DOMNodeInsertedIntoDocument : This event fires when the script inserts a node in the DOM tree as the descendant of another node already in the document.
- DOMNodeRemovedFromDocument : This event fires when the script removes a node from the DOM tree as the descendant of another node already in the document.

  The problem with the mutation events is that lots of changes to your page can make your page feel slow or unresponsive. These can also trigger other event listeners, modifying DOM and leading to more mutation events firing. This is the reason for introducing mutation observers to the script.

  Mutation observers wait until the script finishes its current task before reacting, then reports the changes in a batch (not one at a time). This reduces the number of events that fire when you change the DOM tree through your script. You can also specify which changes in the DOM you want them to react to.

7. HTML5 events

These are the page-level events included in the versions of the HTML5 specialization. New events support more recent devices like phones and tablets. They respond to events such as gestures and movements. You will understand them better after you master the above concepts, thus they are not discussed for now. Work with the events below for now and when you are a better developer, you can search for other events available. The three HTML5 events we will learn are as follows:

- DOMContentLoaded : This event triggers when the DOM tree forms i.e. the script is loading. Scripts start to run before all the resources like *images, CSS, and JavaScript loads*. You can attach this event either to the **window** or the **document** objects.
- Hashchange : It fires when the URL hash changes without refreshing the entire window. Hashes (#) link specific parts (known as **anchors**) within a page. It works on the **window** object; the event object contains both the **oldURL** and the **newURL** properties holding the URLs before and after the hashchange.
- Beforeunload : This event fires on the window object just before the page unloads. This event should only be helpful for the user, not encouraging them to stay on the page. You can add a dialog box to your event, showing a message alerting the users like their changes are not saved.

8. CSS events

These events trigger when the script encounters a CSS element. As CSS is a crucial part of web development, the developers decided to add these events to js to make working with CSS easier. Some of the most common CSS events are as follows:

- Transitionend : This event fires when a CSS transition ends in a program. It is useful to notify the script of the end of transition so that it can take further action.
- Animationstart : These events fire when CSS animation starts in the program.
- Animationiteration : This event occurs when any CSS animation repeats itself. With this event, we can determine the number of times an animation iterates in the script.
- Animationend : It fires when the CSS animation comes to an end in the program. This is useful when we want to act just after the animation process finishes.

Handling events

elem.addEventListener(type, fn)

- type is the event to handle (e.g. click)
- fn is a function to handle the event

Note: functions can be passed as values!

Best practice: semantic elements Use the right element, e.g. don't add click handler to paragraph Otherwise, may be impossible to use with keyboard/touch/screen reader

```
const handleClick = (event) => {
        alert("Button was clicked!");
};
let button = document.body.clickme;
button.addEventListener("click", handleClick);
```

Event argument

event.currentTarget

The element the listener was added to that triggered the event

Recommendation: **event.target** is slightly different; stick to currentTarget

Note: I will use event.target only.

```
const handleClick = (event) => {
        let elem = event.currentTarget;
        elem.textContent = "I was clicked!";
};
```

event.type : It's used to find the type of the event

Event: timeStamp property

event.timeStamp

The timeStamp event property returns the number of milliseconds from the document was finished loading until the specific event was created.

Not all systems provide this information, therefore, timeStamp may be not available for all systems/events.

This value is the number of milliseconds elapsed from the beginning of the time origin until the event was created. If the global object is Window, the time origin is the moment the user clicked on the link, or the script that initiated the loading of the document. In a worker, the time origin is the moment of creation of the worker.

JS

```
function getTime(event) {
        const time = document.getElementById("time");
       time.firstChild.nodeValue = event.timeStamp;
}
document.body.addEventListener("keypress", getTime);
```

Event: preventDefault() method

event.preventDefault()

The preventDefault() method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.

For example, this can be useful when:

- Clicking on a "Submit" button, prevent it from submitting a form
- Clicking on a link, prevent the link from following the URL

Note: Not all events are cancelable. Use the cancelable property to find out if an event is cancelable.

Note: The preventDefault() method does not prevent further propagation of an event through the DOM. Use the stopPropagation() method to handle this.

Event: clientX/Y, pageX/Y and screen/Y

1. **pageX/Y** gives the coordinates relative to the <html> element in CSS pixels.
2. **clientX/Y** gives the coordinates relative to the viewport in CSS pixels.
3. **screenX/Y** gives the coordinates relative to the screen in device pixels.

Regarding your last question if calculations are similar on desktop and mobile browsers... For a better understanding - on mobile browsers - we need to differentiate two new concept: the **layout viewport** and **visual viewport**. The visual viewport is the part of the page that's currently shown onscreen. The layout viewport is synonym for a full page rendered on a desktop browser (with all the elements that are not visible on the current viewport).

On mobile browsers the pageX and pageY are still relative to the page in CSS pixels so you can obtain the mouse coordinates relative to the document page. On the other hand clientX and clientY define the mouse coordinates in relation to the **visual viewport**.

JS

```
document.addEventListener('click', function(e) {
  console.log(
    'page: ' + e.pageX + ',' + e.pageY,
    'client: ' + e.clientX + ',' + e.clientY,
    'screen: ' + e.screenX + ',' + e.screenY)
}, false);
```

Event: ctrlKey property

The **KeyboardEvent.ctrlKey** read-only property returns a boolean value that indicates if the control key was pressed (true) or not (false) when the event occurred.

JS

```
function geek (event) {
        if (event.ctrlKey) {
           document.getElementById("gfg").innerHTML = "Ctrl key is pressed.";
        }
        else {
           document.getElementById("gfg") .innerHTML= ("Ctrl key is not pressed.");
        }
    }
```

Event: shiftKey property

The **KeyboardEvent.shiftKey** read-only property is a boolean value that indicates if the shift key was pressed (true) or not (false) when the event occurred.

The pressing of the shift key may change the key of the event too. For example, pressing B generates key: "b", while simultaneously pressing Shift generates key: "B".

JS

```
function geek (event) {
    if (event.shiftKey) {
        document.getElementById("gfg").innerHTML= "Shift key is pressed.";
    }
    else {
        document.getElementById("gfg").innerHTML= "Shift key is not pressed.";
    }
}
```

Event: altKey property

The **KeyboardEvent.altKey** read-only property is a boolean value that indicates if the alt key was pressed (true) or not (false) when the event occurred.

JS

```
function geek(event) {
    if (event.altKey) {
        document.getElementById("gfg")
            .innerHTML = "Alt key is pressed.";
    } else {
        document.getElementById("gfg")
            .innerHTML ="Alt key is not pressed.";
    }
}
```

Event: code property

The **KeyboardEvent.code** property represents a physical key on the keyboard (as opposed to the character generated by pressing the key). In other words, this property returns a value that isn't altered by keyboard layout or the state of the modifier keys.

If the input device isn't a physical keyboard, but is instead a virtual keyboard or accessibility device, the returned value will be set by the browser to match as closely as possible to what would happen with a physical keyboard, to maximize compatibility between physical and virtual input devices.

This property is useful when you want to handle keys based on their physical positions on the input device rather than the characters associated with those keys; this is especially common when writing code to handle input for games that simulate a gamepad-like environment using keys on the keyboard. Be aware, however, that you can't use the value reported by KeyboardEvent.code to determine the character generated by the keystroke, because the keycode's name may not match the actual character that's printed on the key or that's generated by the computer when the key is pressed.

For example, the code returned is "KeyQ" for the Q key on a QWERTY layout keyboard, but the same code value also represents the ' key on Dvorak keyboards and the A key on AZERTY keyboards. That makes it impossible to use the value of code to determine what the name of the key is to users if they're not using an anticipated keyboard layout.

To determine what character corresponds with the key event, use the **KeyboardEvent.key** property instead.

JS

```
window.addEventListener(
  "keydown",
  (event) => {
    const p = document.createElement("p");
    p.textContent = `KeyboardEvent: key='${event.key}' | code='${event.code}'`;
    document.getElementById("output").appendChild(p);
    window.scrollTo(0, document.body.scrollHeight);
  },
  true,
);
```

This code will print the key pressed and keycode simultaneously.