Events

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://developer.mozilla.org/en-US/docs/Web/API/Event
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events

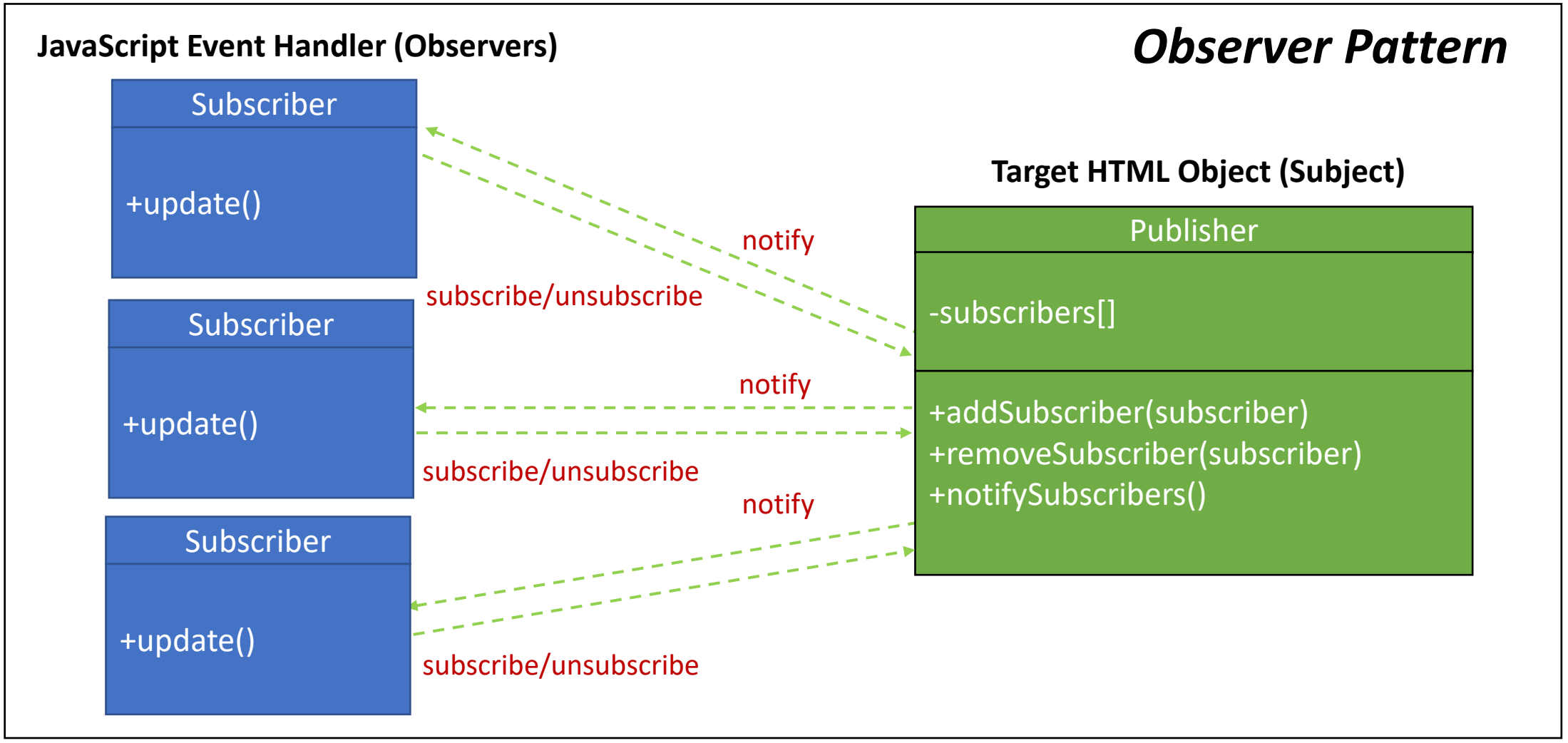
Learning Objectives

- Understanding event propagation: distinguish between event capturing and bubbling and describe how each affects even handling.
- Access and manipulate event object properties such as target, type, currentTarget etc.
- Attach and remove event handlers using JavaScript methods.
- Identify and categorize common event types in JavaScript, including mouse, keyboard, and form events.

Introduction

- JavaScript's interaction with HTML is handled through events, which indicate when particular moments of interest occur in the document or browser window.
- Events can be subscribed to using **listeners** (also called **handlers**) that execute only when an event occurs.
- This model, called the "***observer pattern***" in traditional software engineering, allows ***a loose coupling*** between the behavior of a page (defined in JavaScript) and the appearance of the page (defined in HTML and CSS).

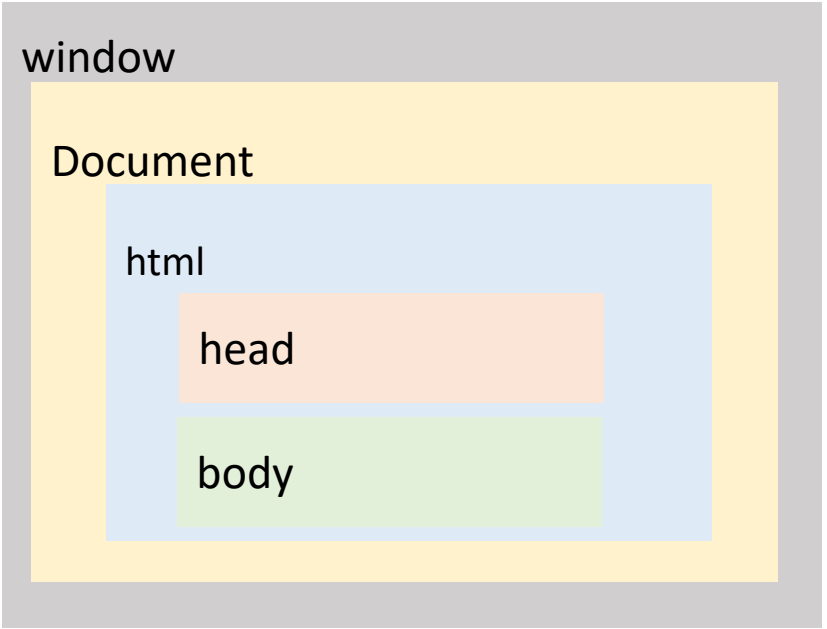
The event and event-handler paradigm in JavaScript is the manifestation of the Observer design pattern. Another name for the Observer pattern is Publisher/Subscriber.



Event Propagation

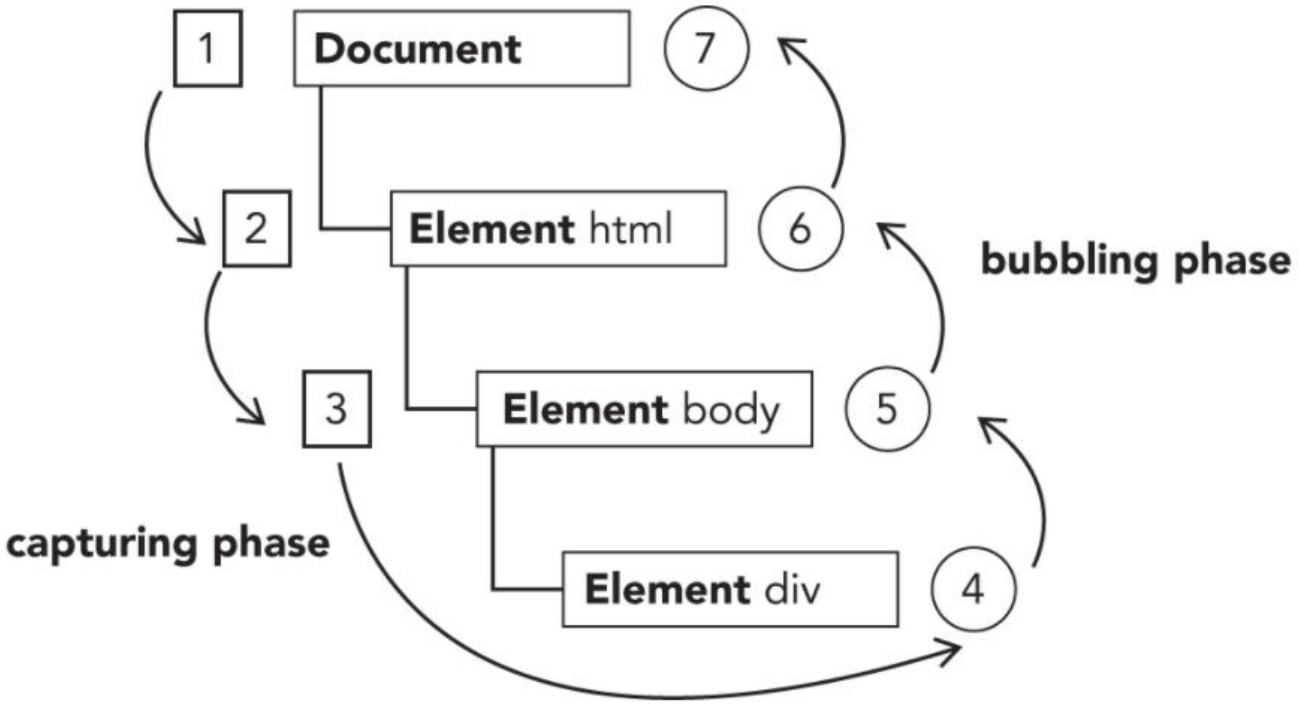
Event Flow Concepts

- Event flow describes the order in which events are received on the page.
    1. Event Bubbling Flow - **start at the most specific element and then flow upward toward the least specific node**.
    2. Event Capturing Flow - **the least specific node should receive the event first and the most specific node should receive the event last**.
- All modern browsers support **event bubbling**, although there are some variations on how it is implemented. **By default**, **all event handlers are registered for the bubbling phase.**
- Modern browsers continue event bubbling up to the `window` object.



Three Phases Event Flow

- When an event is fired on an element that has parent elements, modern browsers run three different phases
  1. In the capturing Phase
  2. In the target Phase
  3. In the Bubbling Phase

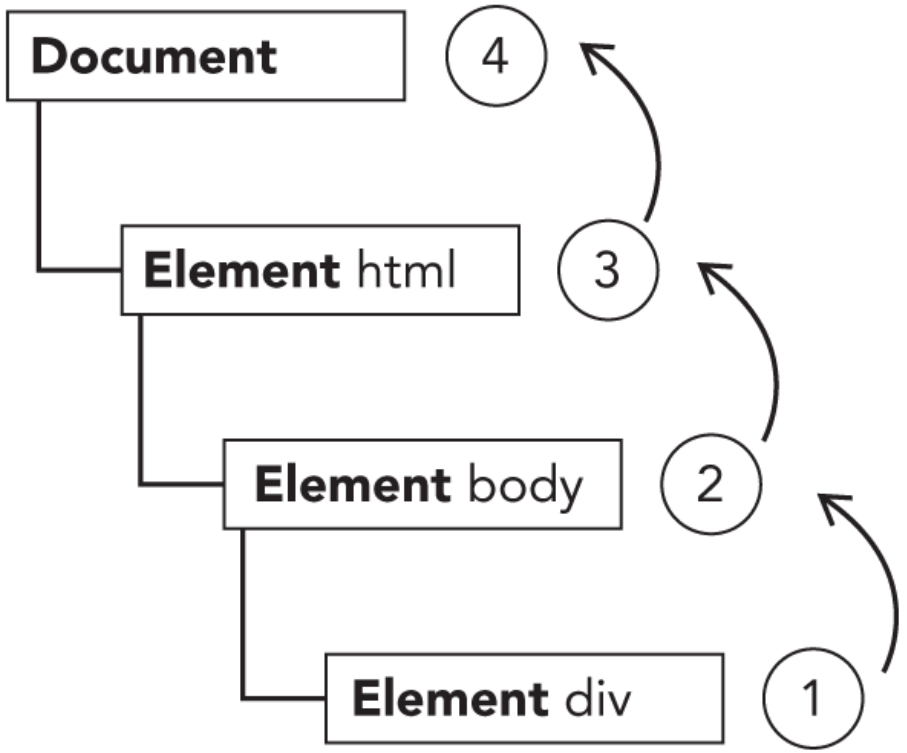


Event Bubbling

- When you click the `<div>` element in the page, the **click** event occurs in the following order:

```
<!DOCTYPE html>
<html>
 <head>
   <title>Event Bubbling Example</title>
 </head>
 <body>
   <div id="myDiv">Click Me</div>
 </body>
</html>
```

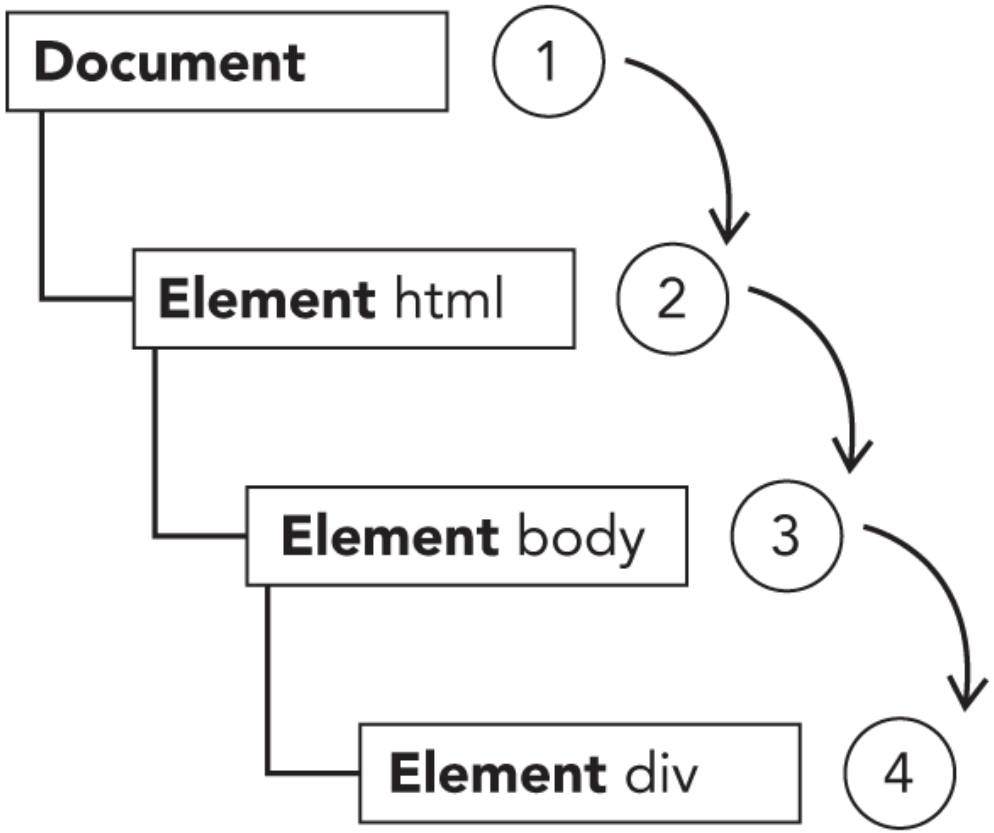
1.
2. <body>
3. <html>
4. document



Event Capturing

```
<!DOCTYPE html>
<html>
 <head>
   <title>Event Bubbling Example</title>
 </head>
 <body>
   <div id="myDiv">Click Me</div>
 </body>
</html>
```

1. Document
2. <html>
3. <body>
4.



Event Handlers

Ways of Using Web Events

1. **Event Handlers Properties** have less power and options, but are easier to use such as `onclick`, `onfocus`, `onmouseover`.

Do not use Inline event handlers, it mixes up your HTML and your JavaScript and becomes unmanageable and inefficient

```
<input type="button" value="Click Me" onclick="console.log('Clicked')"/>
<input type="button" value="Click Me" onclick="showMessage()"/>
```

However, you can use event handlers' properties by writing event handler functions separately

```
const btn = document.getElementById("myBtn")
btn.onclick = function() {
   console.log(btn.id) // "myBtn"
}
```

2. **Adding and removing event handlers**: `addEventListener()` and `removeEventListener()` is more complex, but also more powerful. The main advantages are that:
   - You can remove event-handler code if needed, using `removeEventListener()`.
   - You can add multiple listeners of the same type to elements, if required.

Adding and removing event handlers

- There are two methods to deal with the assignment and removal of event handlers: `addEventListener()` and `removeEventListener()`.
- These methods exist on all DOM nodes and accept **three arguments**:
  1. the *event type's name* to handle
  2. the *event handler or event listener* function
  3. a *Boolean* value indicating whether to call the event handler during the *capture phase* (true) or during the *bubble phase* (false). **Bubble phase is default.**

Adding event handlers: `addEventListener()`

- To add an event handler for the click event on a button, you can use the following code:

```
const btn = document.getElementById("myBtn")
btn.addEventListener("click", () => {
   console.log(btn.id)
}, false)
```

- This code adds an on `click` event handler to a button that will be fired in the bubbling phase (since the last argument is false).

Adding multiple event handlers

- The major advantage is that multiple event handlers can be added. Consider the following example:

```
const btn = document.getElementById("myBtn")
btn.addEventListener("click", () => {
   console.log(btn.id)
}, false)
btn.addEventListener("click", () => {
   console.log("Hello world!")
}, false)
```

- Here, two event handlers are added to the button. The event handlers fire in the order in which they were added, so the first log displays the element's ID and the second displays the message “Hello world!”

Removing event handlers: `RemoveEventListener()`

- Event handlers added via `addEventListener()` can be removed only by using `removeEventListener()` and passing in the same arguments as were used when the handler was added.

```
const btn = document.getElementById("myBtn")
let handler = function() {
  console.log(btn.id)
};
btn.addEventListener("click", handler, false)   // other code here
btn.removeEventListener("click", handler, false) // works!
```

- anonymous functions added using `addEventListener()` cannot be removed,

```
const btn = document.getElementById("myBtn")
btn.addEventListener("click", () => {
   console.log(btn.id)}, false)   // other code here
btn.removeEventListener("click", function() { // won't work!
   console.log(btn.id)}, false)
```

Event Objects

Event Objects

- When an event related to the DOM is fired, all the relevant information is gathered and stored on an object called `event`.
- In DOM-compliant browsers, the `event` object is passed in as the argument to an event handler function.
- This object contains basic information such as the **element that caused the event**, **the type of event that occurred**, and **any other data that may be relevant to the particular event**.
- For example, an event caused by **a mouse action** generates information about **the mouse's position**, whereas an event caused by **a keyboard action** generates information about the **keys that were pressed.**

```
let btn = document.getElementById("myBtn")
btn.addEventListener("click", (event) => {
      console.log(event.type) // "click"}, false)
```

Event Properties and Methods

PROPERTY/METHOD	TYPE	READ/WRITE	DESCRIPTION
`bubbles`	Boolean	Read only	Indicates if the event bubbles.
`cancelable`	Boolean	Read only	Indicates if the default behavior of the event can be canceled.
`currentTarget`	Element	Read only	The element whose event handler is currently handling the event.
`defaultPrevented`	Boolean	Read only	When `true`, indicates that `preventDefault()` has been called (added in DOM Level 3 Events).
`detail`	Integer	Read only	Extra information related to the event.
`eventPhase`	Integer	Read only	The phase during which the event handler is being called: `1` for the capturing phase, `2` for "at target," and `3` for bubbling.
`preventDefault()`	Function	Read only	Cancels the default behavior for the event. If `cancelable` is `true`, this method can be used.

Event Properties and Methods

`stopImmediatePropagation()`	Function	Read only	Cancels any further event capturing or event bubbling and prevents any other event handlers from being called. (Added in DOM Level 3 Events.)
`stopPropagation()`	Function	Read only	Cancels any further event capturing or event bubbling. If `bubbles` is `true`, this method can be used.
`target`	Element	Read only	The target of the event.
`trusted`	Boolean	Read only	When `true`, indicates if the event was generated by the browser. When `false`, indicates the event was created using JavaScript by the developer. (Added in DOM Level 3 Events.)
`type`	String	Read only	The type of event that was fired.
`View`	AbstractView	Read only	The abstract view associated with the event. This is equal to the `window` object in which the event occurred.

The `preventDefault()` method is used to prevent the default action of a particular event.

```
<form action="#" method="post">
      username:<input type="text">
      password <input type="password">
      <div>
        <button id="submit" type="submit" value="Submit New Account">
          Create new account
        </button>
      </div>
    </for>
    <p></p>
```

```
const createBtn = document.getElementById('submit')
createBtn.addEventListener('click', (event) => {
  event.preventDefault()
  const allInputEles = document.querySelectorAll('input')
  const isValidInput = Array.from(allInputEles).every(
    (inputEle) => inputEle.value.length !== 0
  )
  const pEle = document.querySelector('p')
  if (isValidInput) {
    pEle.textContent = 'your account has been created!'
    pEle.style = 'color:green'
  } else {
    pEle.textContent = 'missing some values, please try again'
    pEle.style = 'color:red'
  }
})
```

Event Types

Event Categories

- All major browsers support events specifies the following event groups:
  - **State change events** some events are not triggered directly by user activity but by life cycle or state-related change.
  - **Focus events** are fired when an element gains or loses focus.
  - **Mouse events** are fired when the mouse is used to perform an action on the page.
  - **Keyboard events** are fired when the keyboard is used to perform an action on the page.
  - **Input events** are fired when text is input into the document.

State Change Events

- **DOMContentLoaded** event – DOM is ready, DOM tree is built but external resources such as images or stylesheets may not have loaded.
- **load** event – all external resources such as images are loaded
- **beforeunload** event – the user is leaving the page; we can check if the user saved the changes and ask them whether they really want to leave.
- **Unload (deprecated)** – the user almost left, developers should avoid using this event
- **resize**—Fires on a window or frame when it is resized.
- **scroll**—Fires on any element with a scrollbar when the user scrolls it.

Focus Events

- Focus events are fired when elements of a page receive or lose focus.
- The two primary events of this group are `focus` and `blur`, both of which have been supported in browsers since the early days of JavaScript.
  - **blur**—Fires when an element has lost focus.
  - **focus**—Fires when an element has received focus.

Mouse Events

- A click event can be fired only if a `mousedown` event is fired and followed by a `mouseup` event on the same element; if either `mousedown` or `mouseup` is canceled, then the `click` event will not fire.
- The **mouse events** fire in the following order:

```
mousedown -> mouseup -> click
```

Mouse Events

- **click**—Fires when the user clicks the primary mouse button (typically the left button) or when the user presses the Enter key.
- **mousedown**—Fires when the user *pushes any mouse button down*. This event cannot be fired via the keyboard.
- **mouseup**—Fires when the user *releases a mouse button*. This event cannot be fired via the keyboard.
- **mouseout**—Fires when the mouse cursor is over an element and then the user *moves it over another element*. The element moved to may be outside of the bounds of the original element or a child of the original element. This event cannot be fired via the keyboard.
- **mouseover**—Fires when the mouse cursor is outside of an element and then the user *first moves it inside of the boundaries of the element*. This event cannot be fired via the keyboard.
- **mousemove**—Fires repeatedly as the cursor is being *moved around an element*. This event cannot be fired via the keyboard.

Keyboard Events

- Keyboard events are fired when the user interacts with the keyboard.
- There are three keyboard events, when the user presses a character key once on the keyboard, the `keydown` event is fired first, followed by the `keypress` event, followed by the `keyup` event.

```
keydown -> keypress -> keyup
```

- Note that both `keydown` and `keypress` are fired before any change has been made to the text box, whereas the `keyup` event fires after changes have been made to the text box.

Enter keyup Example

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

```
const inputMessage = document.getElementById('message')
inputMessage.addEventListener('keyup', (event) => {
  if (event.code === 'Enter')
       console.log(event.target.value)
  else
       console.log('no input')
})
```

https://www.freecodecamp.org/news/javascript-keycode-list-keypress-event-key-codes/

Input Events

- The **input** event fires when the value of an `<input>, <select>,` or `<textarea>` element has been changed.
- The **input** event fires just before text is inserted into a text box.

```
<input id="message"/>
<p></p>
```

```
const message = document.querySelector('#message')
const display = document.querySelector('p')
message.addEventListener('input', function () {
        display.textContent = message.value
})
```