

# Module 3-8

## Event Handling

# Objectives

- Selecting DOM objects
- `addEventListener()` function
- Event delegation
- Event bubbling
- Default browser behavior
- Event types and elements
- Adding listeners to new DOM elements
- `removeEventListener()`

# Events

- Events are changes that can occur within HTML DOM elements.
- Examples of events:
  - A user hovering over a piece of text with the mouse cursor.
  - A user clicking on a link or button.
  - A HTML page loading for the first time.
  - A user double clicking somewhere on the page.
- We use JS to help define the actions that should take place if these events occur.

# Event Handling

- A style of user interaction that browsers use to allow developers to react and interact with a user's use of their site

## Event Handler

- What to do when a particular event has happened.

## Event Objects

- Object given to every Event Handler that defines properties about that event, such as location, which DOM element the event happened on, key presses, or mouse click information.

# Event Listening


Reacting to events requires three things:

1. A DOM element that we want to listen to events on

```
<button id="newBtn">Try it</button>
```

2. A specific event that we want to listen to

```
<script>  
document.getElementById("newBtn").addEventListener("click", newBtnFunction);
```



3. A function that holds the logic that we want to execute

```
function newBtnFunction()  
{  
  document.getElementById("demo").innerHTML = "Hello World";  
};
```

# Common Events

Event	Description
click	user clicks once
mouseover	when the mouse cursor is over an element
dblclick	user clicks twice in rapid succession
change	user changes the value on a form (if it's an input box, user needs to click somewhere else to complete the event)
focus	When an element is the currently active one, think again about a form, the field you are currently on is the one that's focused.
blur	Opposite of focus, element has lost focus, something else is focused.

# Adding Events

Events are added using the `addEventListener` method, it is a method of a DOM element, consider the following example:

## HTML

```
<button type="button"
id='superBtn'>You are
awesome.</button>
```

## JS

```
let button = document.getElementById('superBtn');
button.addEventListener('click', action);

function action() {
  window.alert('No.... you are awesome!');
}
```

This method is straightforward, the first parameter is the event we are trying to catch. The second parameter is the action it will take, most likely codified in a function.



# Adding Events using Anonymous Functions

Instead of calling a named function, we can write it anonymously (no name):

HTML

```
<button type="button"
id='superBtn'>You are
awesome.</button>
```

JS

```
let button = document.getElementById('superBtn');
button.addEventListener('click', () => {
    window.alert('No.... you are awesome!');
});
```

Best practice is to first write named function, then call named function in the event listener.

JS

```
function awesomeFunction() {
    window.alert('No.... you are awesome!');
};

document.getElementById("superBtn").addEventListener("click", () => {
    awesomeFunction();
});
```

# Event Object

```
function awesomeFunction() {  
  window.alert('No... you are awesome!');  
};  
  
document.getElementById("superBtn").addEventListener("click", (event) => {  
  awesomeFunction();  
});
```



The event object holds some important properties that we can use. Example, if we want to know which mouse button was pressed in a mousedown event, where a mouse event happened (x, y, or page), if the mouse was moved, etc.

an example

# Event delegation and bubbling

- Event delegation – listening to events where you delegate a parent element as the listener for all of the events that happen inside it.
- Event bubbling – an event starts at the element that triggered it, then moves up the DOM tree until each reaches the html element. Also known as propagation.
  - If you have a listener on a child, and also one on the parent, both will be triggered when the child event occurs

# Event Delegation

- Technique for listening to events where you delegate a parent element as the listener for all the events that happen inside it.
- Examples of events:
  - Listening to all clicks in the document
  - Listening to all clicks inside a form
- Events will bubble up from the specific element that triggered it, until it reaches the top parent. Each parent element that has a listener will get triggered.

Let's check it out

# The “DOMContentLoaded” Event

There is a special event that is particularly helpful as it is one that runs every time a HTML page is loaded, the event for when all DOM Elements have been loaded.

We can use this event to write JS code that “preps” anything on the page before a user goes in and starts clicking on things. Some examples:

- Set some DOM elements (i.e. page titles, lists)
- Add event handlers for elements on the page

# DOMContentLoaded Example

Consider the following code:

```
<html><body>
  <div id='content'>
    <input id='theButton' type='button' value='Surprise!'/>
  </div>
```

HTML

```
<script src="thisScript.js"></script>
</body></html>
```

```
document.addEventListener('DOMContentLoaded', doAfterDOMLoads);
```

1

```
function doAfterDOMLoads() {
```

2

```
  let btn = document.getElementById('theButton');
```

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

3

```
}

function buttonAction() {
  window.alert('surprise!');
```

4

JS

1. First thing that happens, an event listener is defined that will be triggered by DOMContentLoaded to run the method **doAfterDOMLoads**.
2. Once the DOMContentLoaded event fires, the method **doAfterDOMLoads** is executed.
3. Note that in turn, this method defines another event handler for the button. If the button is clicked, the method **buttonAction** will execute.
4. If someone clicks the button, this method executes.



# DOMContentLoaded Example

You can structure the previous block of JS code with anonymous functions if you want. Note that these two blocks of code are identical:

```
document.addEventListener('DOMContentLoaded', doAfterDOMLoads);
```

```
function doAfterDOMLoads() {
```

JS

```
    let btn = document.getElementById('theButton');
```

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

```
}
```

```
function buttonAction() {
```

```
    window.alert('surprise!');
```

```
}
```

```
document.addEventListener('DOMContentLoaded',
```

```
() => {
```

JS

```
    let btn = document.getElementById('theButton');
```

```
    btn.addEventListener('click',
```

```
        () => {
```

```
            window.alert('surprise!');
```

```
        }
```

```
    );
```

```
}
```

```
);
```

# The “DOMContentLoaded” Event

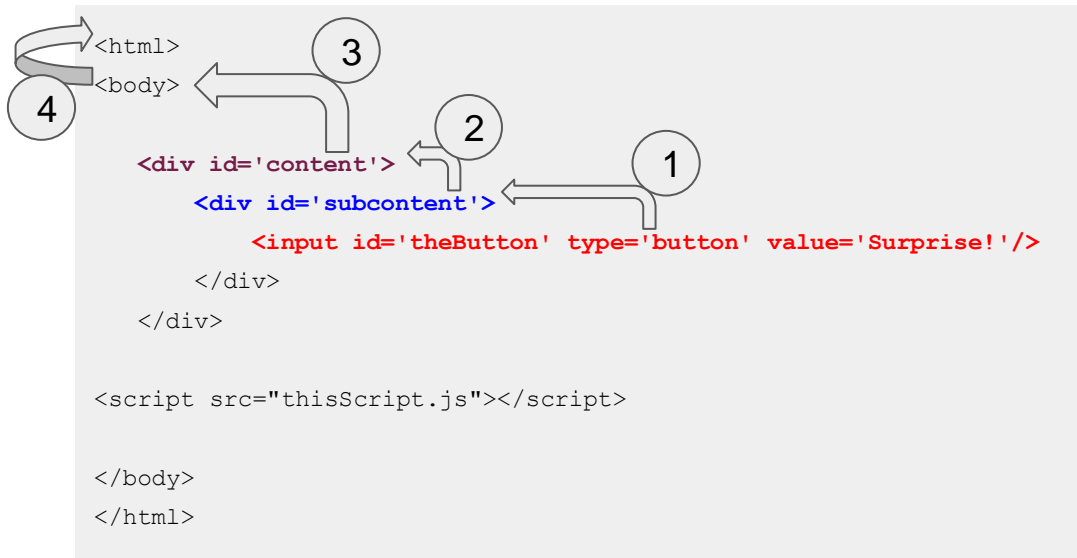
This is the standard way we want to setup event handlers:

1. Add an event listener that responds to the DOMContentLoaded event.
2. To the method attached to step 1, add all other event listeners for other page elements (i.e. buttons, form elements, etc.)

Let's work on a more comprehensive example

# Event Bubbling

If your HTML elements have a hierarchy of parent child relationships, an event tied to a child element will propagate up (bubble), possibly activating any events tied to the parent. Consider the following:



Under the hood, an event attached to the button travels upward in the following direction:

1. From the button to its immediate parent (subcontent)
2. From *subcontent* to *content*
3. From *content* to *body*
4. From *body* to *html*

# Event Bubbling

If the following JS code were in place, we'd see the popup appear three times:

```
document.addEventListener('DOMContentLoaded', doAfterDOMLoads);

function doAfterDOMLoads() {
  let btn = document.getElementById('theButton');
  let sub = document.getElementById('subcontent');
  let main = document.getElementById('content');
  btn.addEventListener('click', buttonAction1);
  sub.addEventListener('click', buttonAction2);
  main.addEventListener('click', buttonAction2);
}

function buttonAction1() {
  window.alert('surprise!');
}

function buttonAction2() {
  window.alert('surprise again!');
}
```

# How to stop event bubbling (propagation)

If the following JS code were in place, we'd see the popup appear three times:

```
document.addEventListener('DOMContentLoaded', doAfterDOMLoads);

function doAfterDOMLoads() {
  let btn = document.getElementById('theButton');
  let sub = document.getElementById('subcontent');
  let main = document.getElementById('content');
  btn.addEventListener('click', buttonAction1);
  sub.addEventListener('click', buttonAction2);
  main.addEventListener('click', buttonAction2);
}

function buttonAction1(event) {
  window.alert('surprise!');
  event.stopPropagation();
}

function buttonAction2() {
  window.alert('surprise again!');
}
```

# Objectives

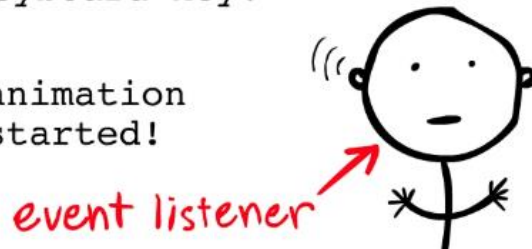
- Selecting DOM objects

a button  
has been  
clicked!

a file has  
finished  
loading!

someone pressed  
a keyboard key!

the animation  
has started!



event listener

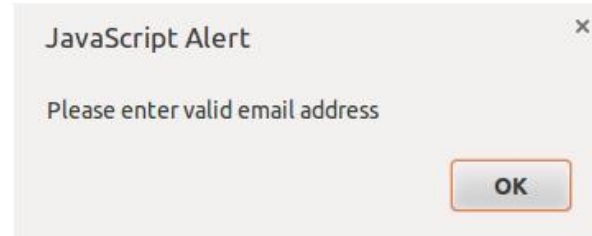
# Objectives

- Selecting DOM objects
- `addEventListener()` function

Email:

User Name:

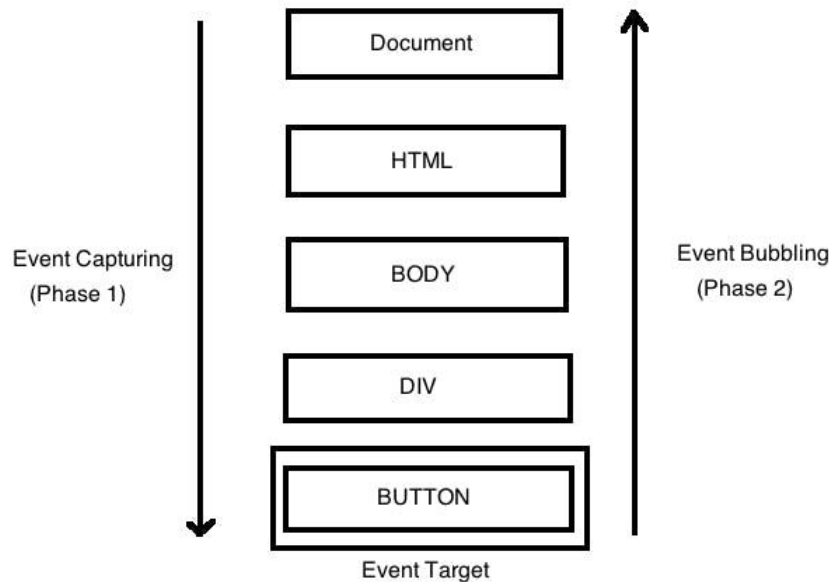
Password:





# Objectives

- Selecting DOM objects
- `addEventListener()` function
- Event delegation



# Objectives

- Selecting DOM objects
- `addEventListener()` function
- Event delegation
- Event bubbling

To do this, we'll rely on DOM event bubbling — the process by which an event progresses from its direct target up through the target's node ancestry until it reaches the `window` object. The graphick below may help to illustrate the flow: In this case, we'll count on the fact that a click or a mouseover on (A) the text node *within* an `<li>` progresses to the `<li>` element itself (B), then to the `<ul>` element (C), and then to the list's parent `<div>` (D) and so on up the document:

```
<div id="container" D>
  C<ul id="list">
    <li id="li-1">List Item 1</li>
    <li id="li-2">List Item 2</li>
    B<li id="li-3">List Item 3</li>
    A<li id="li-4">List Item 4</li>
    <li id="li-5">List Item 5</li>
    <li id="li-6">List Item 6</li>
  </ul>
</div>
```

# Objectives

- Selecting DOM objects
- `addEventListener()` function
- Event delegation
- Event bubbling
- Default browser behavior

## Control of Default behavior

Sometimes a default scenario of event processing includes some additional behavior: bubbling and capturing or displaying context menu.

If you don't need a default behavior, you can cancel it. Use object *event* and next methods for this purpose:

`e.preventDefault();`



for aborting default browser behavior.

[2]

`e.stopPropagation();`

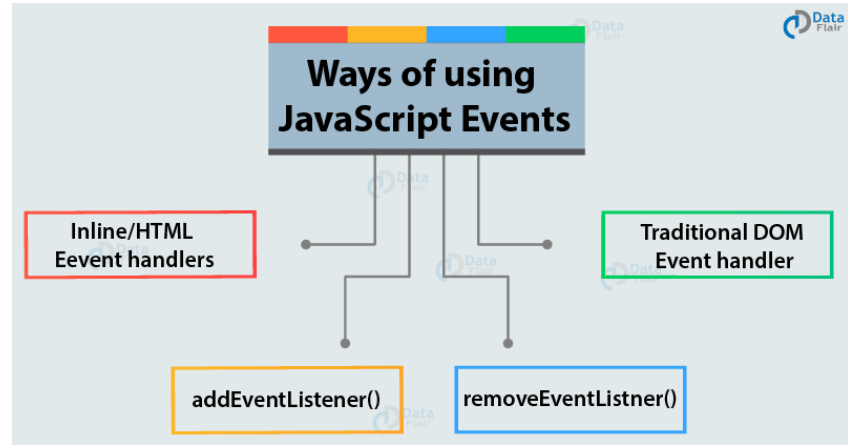


for discarding *bubbling* and *capturing*.

[1]

# Objectives

- Selecting DOM objects
- `addEventListener()` function
- Event delegation
- Event bubbling
- Default browser behavior
- Event types and elements



# Objectives

- Selecting DOM objects
- addEventListener() function
- Event delegation
- Event bubbling
- Default browser behavior
- Event types and elements
- Adding listeners to new DOM elements

```
1 var elements = document.getElementsByClassName("classname");
2
3 var myFunction = function() {
4     var attribute = this.getAttribute("data-myattribute");
5     alert(attribute);
6 };
7
8 for (var i = 0; i < elements.length; i++) {
9     elements[i].addEventListener('click', myFunction, false);
10 }
11
12 // If you have ES6 support you can replace your last line with:
13
14 Array.from(elements).forEach(function(element) {
15     element.addEventListener('click', myFunction);
16 });
```

# Objectives

- Selecting DOM objects
- `addEventListener()` function
- Event delegation
- Event bubbling
- Default browser behavior
- Event types and elements
- Adding listeners to new DOM elements
- `removeEventListener()`

The `removeEventListener` method, called with arguments similar to `addEventListener`, removes a handler.

```
1 <button>Act-once button</button>
2 <script>
3   var button = document.querySelector("button");
4   function once() {
5     console.log("Done.");
6     button.removeEventListener("click", once);
7   }
8   button.addEventListener("click", once);
9 </script>
```



Act-once button

Done.

To be able to unregister a handler function, we give it a name (such as `once`) so that we can pass it to both `addEventListener` and `removeEventListener`.