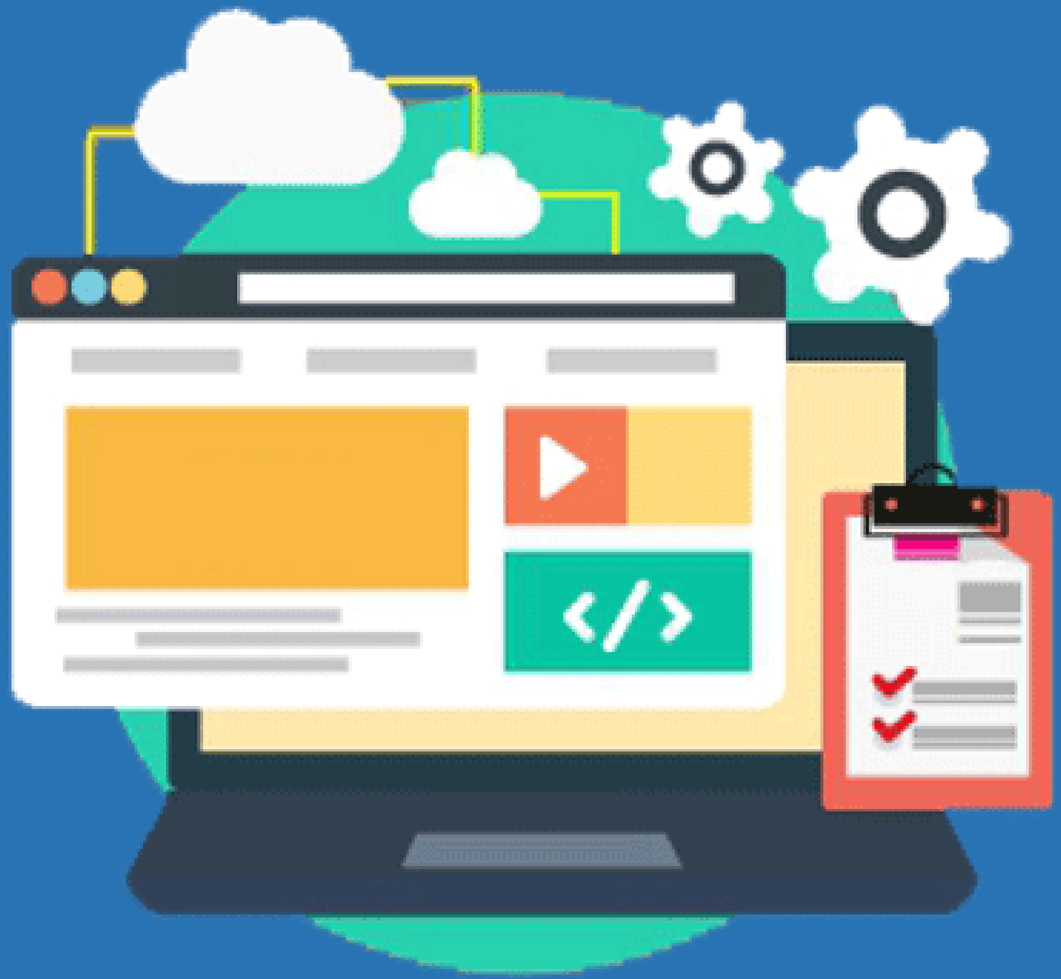
JavaScript Events



www.educba.com

Table of Contents

Introduction

What is Events handling in Javascript?

Event handling is a fundamental concept in JavaScript, allowing you to create dynamic, interactive web pages.

Events are actions or occurrences that happen in the browser, like clicking a button, hovering over an element, typing in a form field, or even the page loading.

JavaScript provides the ability to listen for these events and execute code in response.

Common Events in JavaScript

Events represent actions or occurrences that can be detected by JavaScript. Some common events include:

- Mouse Events
- Keyboard Events
- Form Events
- Window Events

Mouse Events

Common Mouse events are:

- click: Triggered when an element is clicked.
- dblclick: Triggered when an element is double-clicked.
- mouseover: Triggered when the mouse pointer enters an element.
- mouseout: Triggered when the mouse pointer leaves an element.
- mousedown: Triggered when the mouse button is pressed down on an element.
- mouseup: Triggered when the mouse button is released over an element.
- mousemove: Triggered when the mouse pointer moves within an element.

Keyboard Events

Common Keyboard events are:

- keydown: Triggered when a key is pressed down.
- keyup: Triggered when a key is released.
- keypress: Triggered when a key is pressed and held down (deprecated in favor of keydown and keyup).

Form Events

Common Form events are:

- submit: Triggered when a form is submitted.
- reset: Triggered when a form is reset.
- input: Triggered when the value of an input element changes.
- change: Triggered when the value of an input, select, or textarea element changes after losing focus.
- focus: Triggered when an element gains focus.
- blur: Triggered when an element loses focus.

Window Events

Common Window events are:

- load: Triggered when the entire page and its resources have loaded.
- resize: Triggered when the window is resized.
- scroll: Triggered when the document or an element is scrolled.
- unload: Triggered when the user navigates away from the page.

Event Handling in JavaScript

To handle events, you can use event listeners or HTML event attributes. Event listeners are generally preferred because they separate JavaScript from HTML, promoting better code organization.

Adding event Listeners

An event listener is a function that waits for a specific event to occur on an element.

You can add event listeners using the addEventListener() method.

Syntax:

```
element.addEventListener(event, function, useCapture);
```

Explanation:

- element: The DOM element to attach the event listener to.
- event: The event type, e.g., "click".
- function: The function to execute when the event occurs.
- useCapture: A boolean indicating whether to use event capturing or bubbling (optional).

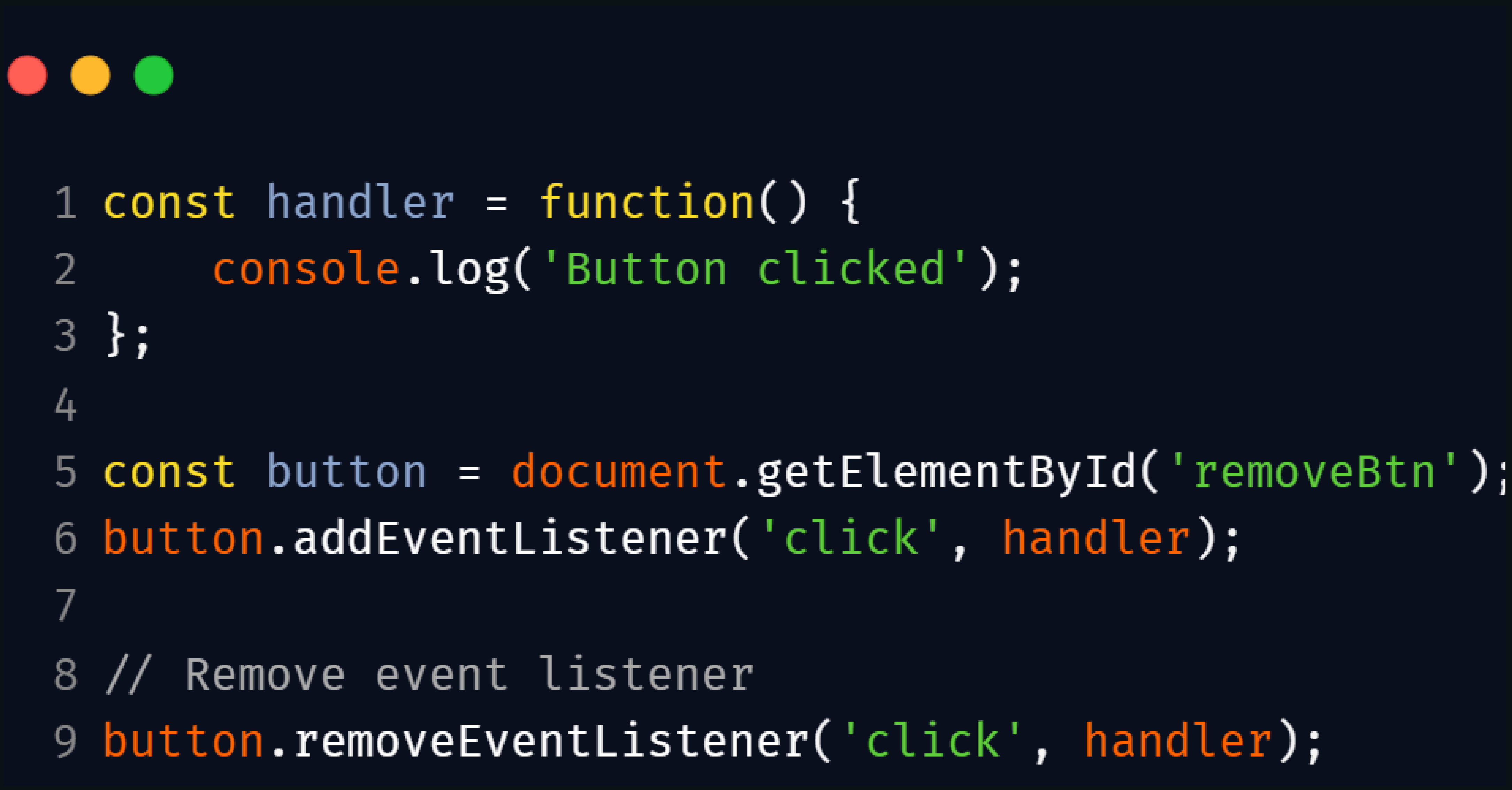
Removing event Listeners

Event listeners can be removed using the removeEventListener() method.

Syntax:

```
element.removeEventListener(event, function, useCapture);
```

Adding and removing Event Listeners Example:



```javascript
const handler = function() {
    console.log('Button clicked');
};

const button = document.getElementById('removeBtn');
button.addEventListener('click', handler);

// Remove event listener
button.removeEventListener('click', handler);
```

Event Propagation: Bubbling and Capturing

When an event occurs, it doesn’t just trigger on the element it’s directly attached to. The event can also propagate (or travel) through the DOM tree. This process can happen in two phases:

1. Capturing Phase (Event Capturing)
2. Bubbling Phase (Event Bubbling)

Capturing Phase (Trickle Down)

During the capturing phase, the event starts from the root of the DOM and goes down to the target element. This phase is rarely used in practice.

Bubbling Phase (Trickle Up)

In the bubbling phase, the event starts from the target element and bubbles up to its parent elements.

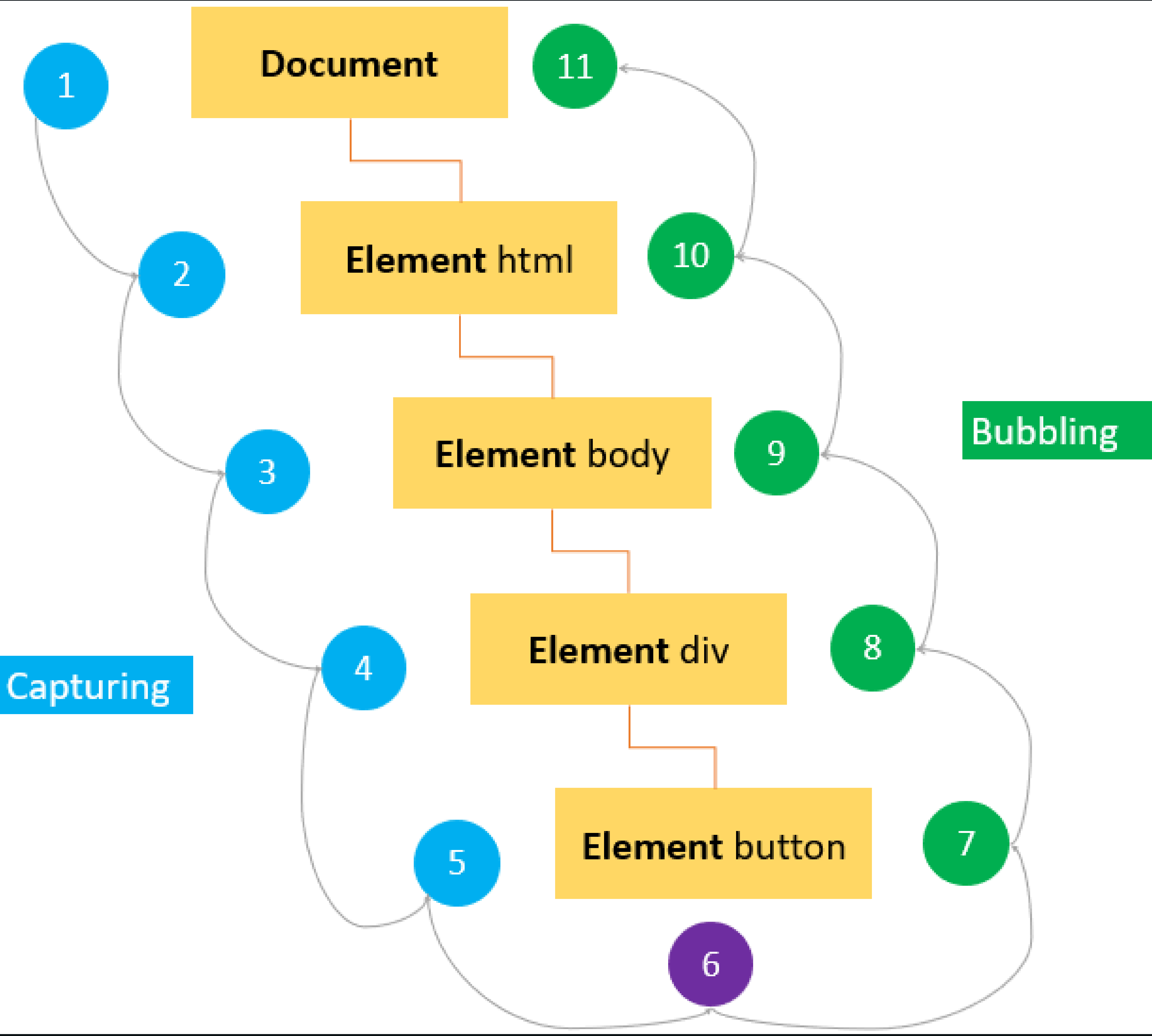
For example, if you click a button inside a div, the click event first triggers on the button, then on the div, and continues up the DOM tree.

```
// html
<div id="parent">
  <button id="child">Click me</button>
</div>

// javascript
<script>
  document.getElementById("parent").addEventListener("click", function() {
      console.log("Parent DIV clicked");
  });

  document.getElementById("child").addEventListener("click", function() {
      console.log("Button clicked");
  });
</script>
```

kodemaven-portfolio.vercel.app



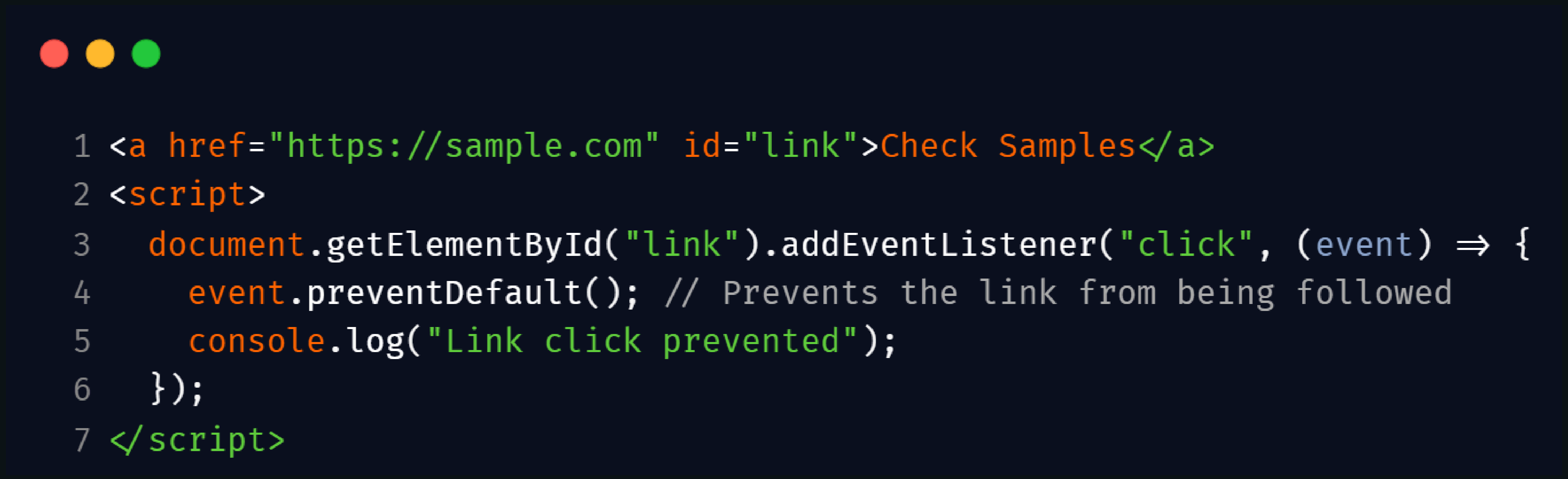
Pictoral representation of Events propagation

Capturing trickles down while Bubbling goes up

Preventing Default Behavior and Stopping Propagation

Preventing Default Behaviors

The preventDefault() method is used to prevent the default action that is associated with an event from being executed. This is particularly useful when you want to override the default behavior of certain HTML elements or events, such as preventing a form submission, stopping a link from navigating, or disabling right-click context menus.



```
<a href="https://sample.com" id="link">Check Samples</a>
<script>
  document.getElementById("link").addEventListener("click", (event) => {
    event.preventDefault(); // Prevents the link from being followed
    console.log("Link click prevented");
  });
</script>
```

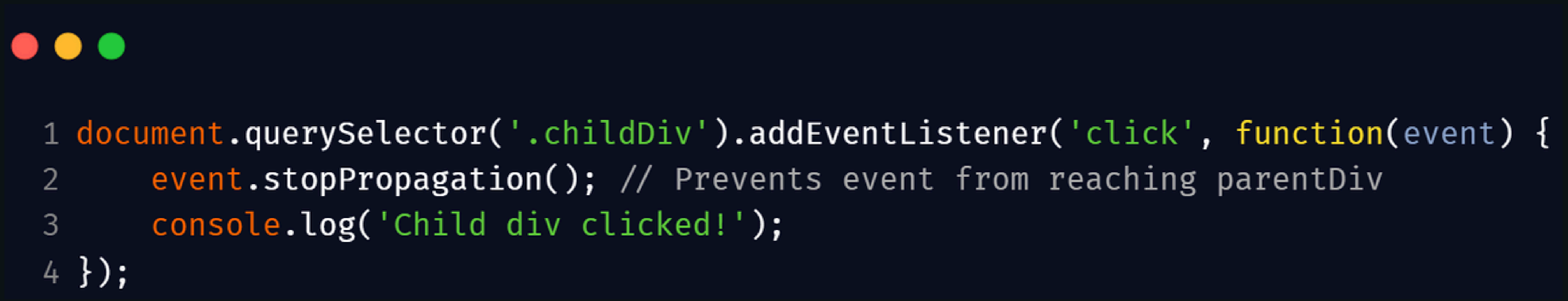
The method event.preventDefault orevents the link from being exposed or followed.

kodemaven-portfolio.vercel.app

Preventing Propagation

By default, most events propagate in the bubbling phase. However, there are situations where you might want to prevent an event from continuing to propagate through the DOM, either during capturing or bubbling. This is where stopPropagation() comes into play.

The stopPropagation() method is used to stop the event from propagating further through the DOM. This means that if an event handler calls stopPropagation(), the event won't continue to travel up (or down) the DOM tree. It effectively halts the propagation of the event, preventing any parent elements from receiving the event.



```javascript
document.querySelector('.childDiv').addEventListener('click', function(event) {
    event.stopPropagation(); // Prevents event from reaching parentDiv
    console.log('Child div clicked!');
});
```

kodemaven-portfolio.vercel.app

Event Object and Event Properties

The Event Object is a central part of event handling in JavaScript. It provides details about the event and methods to control the event's behavior. Depending on the type of event (e.g., mouse, keyboard, form), the Event Object can have different properties.

Common Properties:

- target: The element that triggered the event.
- type: The type of event (e.g., click, keydown).
- key: The key pressed (for keyboard events).

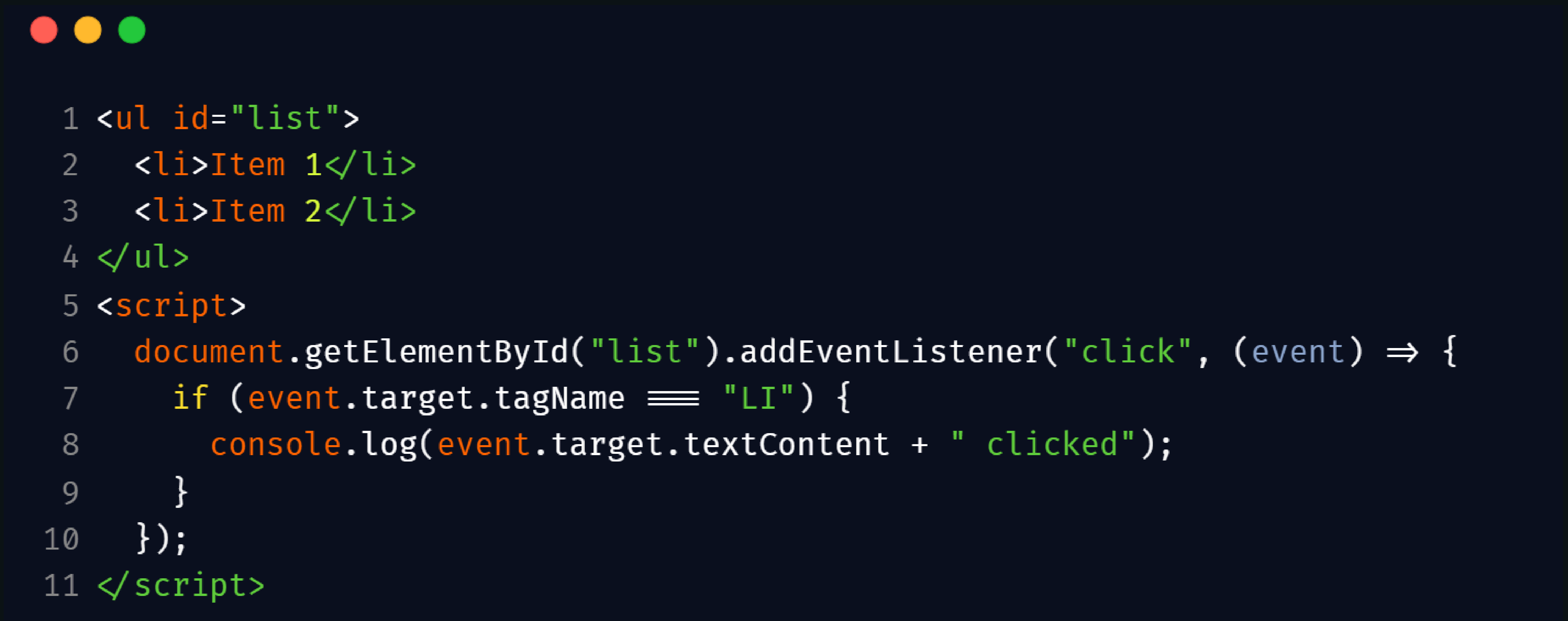
```
<button id="btn">Click Me</button>
<script>
  document.getElementById("btn").addEventListener("click", (event) => {
    console.log(event.target); // The button element
    console.log(event.type); // "click"
  });
</script>
```

Event Delegation

Event delegation is a technique where a single event listener is added to a parent element, allowing it to handle events for all of its child elements. This is efficient for handling events on dynamically added elements.

Why it is used:

- Reduces memory usage.
- Handles events on dynamically added elements.



```
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
<script>
  document.getElementById("list").addEventListener("click", (event) => {
    if (event.target.tagName === "LI") {
      console.log(event.target.textContent + " clicked");
    }
  });
</script>
```

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi