Event Handling: Click Events With Event.target

Imagine a user is viewing a Web page showing 300 different products. The user carefully studies the page, makes a selection, and clicks on one of the 300 products. Could we find out through code which element was clicked on? Yes!

Previously we learned how to handle a click event using an element's ID. However, what if we don't know the ID of the clicked element before it's clicked? There is a simple property we can use to discover on which element the click event occurred: `event.target`.

According to the MDN doc on event.target, "the `target`property of the`Event`interface is a reference to the object that dispatched the event. It is different from event.currentTargetwhen the event handler is called during the bubbling or capturing phase of the event." Essentially:

- `event.target`refers to the element on which the event occurred (e.g. a clicked element).
- `event.currentTarget`refers to the element to which the event handler has been attached, which could be the parent element of the `event.target`element. (*Note: We'll talk about this in more detail in the reading on The Bubbling Principle.*)

It is common practice for developers to use `event.target`to reference the element on which the event occurs in an event handling function. Let's practice using this handy property to get the ID of a clicked element.

Use event.target to console.log the ID of a clicked div

Let's say we had an HTML page with 10 `divs`, each with a unique ID, like below. We want to click on any one of these divs and print the clicked div's ID to the console.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="example.css" />
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body>
    <div id="div-1" class="box">1</div>
    <div id="div-2" class="box">2</div>
    <div id="div-3" class="box">3</div>
    <div id="div-4" class="box">4</div>
    <div id="div-5" class="box">5</div>
    <div id="div-6" class="box">6</div>
    <div id="div-7" class="box">7</div>
    <div id="div-8" class="box">8</div>
    <div id="div-9" class="box">9</div>
    <div id="div-10" class="box">10</div>
  </body>
</html>
```

In our linked **example.css**file, we'll add style to the `.box`class to make our `divs`easier to click on:

CSS

```
.box {
  border: 2px solid gray;
  height: 50px;
  width: 50px;
  margin: 5px;
}
```

Now, we'll write Javascript to print the clicked div's ID to the console. Again, we want to wait for the necessary DOM elements to load before running our script using `DOMContentLoaded`. Then, we'll listen for a click event and `console.log`the clicked element's ID.

Javascript

```
// example.js

// Wait for the DOM to load
window.addEventListener("DOMContentLoaded", event => {
  // Add a click event listener on the document's body
  document.body.addEventListener("click", event => {
    // console.log the event target's ID
    console.log(event.target.id);
  });
});
```

If you open up your HTML in a browser, you should see the 10 `divs`. Click on any one of them. Open up the browser console by right-clicking, selecting*Inspect*, and opening the *Console*tab. The ID of the div you clicked should be printed to the console. Click on the other divs randomly, and make sure their IDs print to the console as well.

What we learned:

- The definition of `event.target`
- How `event.target`differs from `event.currentTarget`
- How to console.log the ID of a clicked element using `event.target`

Event Handling: Common Page Events

Event handling is the core of front-end development. When a user interacts with HTML elements on a website, those interactions are known as ***events***. Developers use Javascript to respond to those events. In this reading, we'll go over three common events and do exercises to add functionality based on those events:

- A button click
- A checkbox being checked
- A user typing a value into an input

Handling a button click event

Let's start with a common event that occurs on many websites: a button click. Usually some functionality occurs when a button is clicked -- such as displaying new page elements, changing current elements, or submitting a form.

We'll go through how to set up a click eventlistener and update the click count after each click. Let's say we have a button element in an HTML file, like below:

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <button id="increment-count">I have been clicked <span id="clicked-count">0
  </body>
```

```
</html>
```

We'll write Javascript to increase the value of the content of `span#clicked-count` by one each time `button#increment-count` is clicked. Remember to use the `DOMContentLoaded` event listener in an external script to ensure the button has loaded on the page before the script runs.

Javascript

If you open up the HTML file in a browser, you should see the button. If you click the button rapidly and repeatedly, the value of `span#clicked-count` should increment by one after each click.

```
// script.js

window.addEventListener("DOMContentLoaded", event => {
  const button = document.getElementById("increment-count");
  const count = document.getElementById("clicked-count");
  let clicks = 0;
  button.addEventListener("click", event => {
    clicks += 1;
    count.innerHTML = clicks;
  });
});
```

Using addEventListener() vs. onclick

Adding an event listener to the button element, as we did above, is the preferred method of handling events in scripts. However, there is another method we could use here: GlobalEventHandlers.onclick. Check out www.simonewebdesign.it for a breakdown of the differences between using `addEventListener()` and `onclick`. One distinction is

that `onclick` overrides existing event listeners, while `addEventListener()` does not, making it easy to add new event listeners.

The syntax for `onclick` is: `target.onclick = functionRef;` If we wanted to rewrite the button click event example using `onclick`, we would use the following:

```
let clicks = 0;
button.onclick = event => {
  clicks += 1;
  count.innerHTML = clicks;
};
```

There's also yet another way to register the `onclick` event, by putting it directly into the HTML markup and using a named function like so:

```
function clickHandler(event) {
  clicks += 1;
  count.innerHTML = clicks;
}


<button onclick="clickHandler(event)"/>
```

This way suffers from all the problems that the `onclick` property suffers from with the additional problem of relying on the global `window.event` object. You'll noticed that `clickHandler(event)` is passing in an `event` variable. Where is that coming from? Well everytime an event happens, the global variable `window.event` changes to be the current event so that's what is being passed to the function. It's a very messy approach and generally should be avoided.

We'll stick to using `addEventListener()` in our code, but it's important for front-end developers to understand the differences between the methods above and use cases for each one.

Handling a checkbox check event

Another common event that occurs on many websites is when a user checks a checkbox. Checkboxes are typically recorded values that get submitted when a user submits a form, but checking the box sometimes also triggers another function.

Let's practice displaying an element when the box is checked and hiding it when the box is unchecked. We'll pretend we're on a pizza delivery website, and we're filling out a form for pizza toppings. There is a checkbox on the page for extra cheese, and when a user checks that box we want to show a `div` with pricing info. Let's set up our HTML file with a `checkbox` and `div` to show/hide, as well as a link to our Javascript file:

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <h1>Pizza Toppings</h1>
    <input type="checkbox" id="on-off">
    <label for="on-off">Extra Cheese</label>
    <div id="now-you-see-me" style="display:none">Add $1.00</div>
  </body>
</html>
```

Note that we've added `style="display:none"` to the `div` so that, when the page first loads and the box is unchecked, the `div` won't show.

In our `script.js` file, we'll set up an event listener for `DOMContentLoaded` again to make sure the `checkbox` and `div` have loaded. Then, we'll write Javascript to show `div#now-you-see-me` when the box is checked and hide it when the box is unchecked.

Javascript

```
// script.js

window.addEventListener("DOMContentLoaded", event => {
  // store the elements we need in variables
  const checkbox = document.getElementById("on-off");
  const divShowHide = document.getElementById("now-you-see-me");
  // add an event listener for the checkbox click
  checkbox.addEventListener("click", event => {
    // use the 'checked' attribute of checkbox inputs
    // returns true if checked, false if unchecked
    if (checkbox.checked) {
      // if the box is checked, show the div
      divShowHide.style.display = "block";
      // else hide the div
    } else {
      divShowHide.style.display = "none";
    }
  });
});
```

Open up the HTML document in a browser and make sure that you see the `checkbox` when the page first loads and not the `div`. The `div` should show when you check the box, and appear hidden when you uncheck the box.

The code above works. However, what would happen if we had a whole page of checkboxes with extra options inside each one that would show or hide based

on whether the boxes are checked? We would have to call `Element.style.display = "block"` and `Element.style.display = "none"` on each associated `div`.

Instead, we could add a `show` or `hide` class to the `div` based on the checkbox and keep our `display:block` and `display:none` in CSS. That way, we could reuse the classes on different elements, as well as see class names change in the HTML. Here's how the code we wrote above would look if we used CSS classes:

Javascript

```
// script.js
// we need to wait for the stylesheet to load
window.onload = () => {
  // store the elements we need in variables
  const checkbox = document.getElementById("on-off");
  const divShowHide = document.getElementById("now-you-see-me");
  // add an event listener for the checkbox click
  checkbox.addEventListener("click", event => {
    // use the 'checked' attribute of checkbox inputs
    // returns true if checked, false if unchecked
    if (checkbox.checked) {
      // if the box is checked, show the div
      divShowHide.classList.remove("hide");
      divShowHide.classList.add("show");
      // else hide the div
    } else {
      divShowHide.classList.remove("show");
      divShowHide.classList.add("hide");
    }
  });
};
```

CSS

```
.show {
  display: block;
}
.hide {
  display: none;
}
```

HTML (Remove the style attribute, and add the "hide" class)

```
<div id="now-you-see-me" class="hide">Add $1.00</div>
```

Handling a user input value

You've learned a lot about event handling so far! Let's do one more exercise to practice event handling using an input. In this exercise, we'll write JavaScript that will change the background color of the page to cyan five seconds after a page loads unless the field `input#stopper` contains only the text "STOP".

Let's set up an HTML file with the input and a placeholder directing the user to type "STOP": **HTML**

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <input id="stopper" type="text" placeholder="Quick! Type STOP">
  </body>
</html>
```

Now let's set up our Javascript:

Javascript

```
// script.js
// run when the DOM is ready
window.addEventListener("DOMContentLoaded", event => {
  const stopCyanMadness = () => {
    // get the value of the input field
    const inputValue = document.getElementById("stopper").value;
    // if value is anything other than 'STOP', change background color
    if (inputValue !== "STOP") {
      document.body.style.backgroundColor = "cyan";
    }
  };
  setTimeout(stopCyanMadness, 5000);
});
```

The code at the bottom of our function might look familiar. We used`setInterval`along with the Javascript Date object when we set up our current time clock. In this case we're using `setTimeout`, which runs `stopCyanMadness`after 5000 milliseconds, or 5 seconds after the page loads.

What we learned:

- How to add an event listener on a button click
- How to add an event listener to a checkbox
- Styling elements with Javascript vs. with CSS classes
- How to check the value of an input

Event Handling: Form Validation

Everyone has submitted a form at some point. Form submissions are another common action users take on a website. We've all seen what happens if we put in values that aren't accepted on a form -- frustrating errors! Those errors prompt the user to input accepted form values before submission and are the first check to ensure valid data gets stored in the database.

Learning how to implement front-end validation before a user submits a form is an important skill for developers. In this reading, we'll learn how to check whether two password values on a form are equal and prevent the user from submitting the form if they're not.

Validate passwords before submitting a form

In order to validate passwords, we need a form with two password fields: a password field and a confirmation field. We'll also include two other fields that are common on a signup page: a name field and an email field. See the example below:

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <form class="form form--signup" id="signup-form">
      <input class="form__field" id="name" type="text" placeholder="Name" style
      <input class="form__field" id="email" type="text" placeholder="Email" sty
      <input class="form__field" id="password" type="text" placeholder="Passwor
      <input class="form__field" id="confirm-password" type="text" placeholder=
```

```
      <button class="form__submit" id="submit" type="submit">Submit</button>
    </form>
  </body>
</html>
```

Now, we'll set up our `script.js`file with code that will:

- Listen for a form submission event
- Get the values of the two password fields and check for a match
- Alert the user if there's not a match, or submit the form

Javascript

```
// script.js
window.addEventListener("DOMContentLoaded", event => {
  // get the form element
  const form = document.getElementById("signup-form");

  const checkPasswordMatch = event => {
    // get the values of the pw field and pw confirm field
    const passwordValue = document.getElementById("password").value;
    const passwordConfirmValue = document.getElementById("confirm-password")
      .value;
    // if the values are not equal, alert the user
    // otherwise, submit the form
    if (passwordValue !== passwordConfirmValue) {
      // prevent the default submission behavior
      event.preventDefault();
      alert("Passwords must match!");
    } else {
      alert("The form was submitted!");
    }
  };
  // listen for submit event and run password check
  form.addEventListener("submit", checkPasswordMatch);
});
```

In the code above, we prevented the form submission if the passwords don't match using Event.preventDefault(). This method stops the default action of an event if the event is not explicitly handled. We then alerted the user that the form submission was prevented.

What we learned:

- Front-end form validation prevents invalid data from being recorded in the database.
- We use `Event.preventDefault()` to stop form submission.
- Users are typically notified when default behavior is prevented.

Event Handling: Input Focus and Blur

Form inputs are one of the most common HTML elements users interact with on a website. By now, you should be familiar with how to listen for a *click event*and run a script. In this reading, we'll learn about a couple of other events on an input field and how to use them:

- Element: focus event
- Element: blur event

Listening for focus and blur events

According to MDN, the focus eventfires when an element, such as an input field, receives focus (i.e. when a user has clicked on that element).

The opposite of the focus event is the blur event. The blur event fires when an element has lost focus (i.e. when the user clicks out of that element).

Let's see these two events in action. We'll set up an HTML page that includes`<input type="text" id="fancypants">`. Then, we'll write JavaScript that changes the background color of the `fancypants`textbox to `#E8F5E9`when the focus is on the textbox and turns it back to its normal color when focus is elsewhere.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
   <input type="text" id="fancypants">
```

```
  </body>
</html>
```

Javascript

```
// script.js

window.addEventListener("DOMContentLoaded", event => {
  const input = document.getElementById("fancypants");

  input.addEventListener("focus", event => {
    event.target.style.backgroundColor = "#E8F5E9";
  });
  input.addEventListener("blur", event => {
    event.target.style.backgroundColor = "initial";
  });
});
```

In the code above, we changed the background color of the input on `focus` and changed it back to its initial value on `blur`. This small bit of functionality signals to users that they've clicked on or off of an input field, which is especially helpful and more user-friendly when there is a long form on the page. Now you can use `focus` and `blur` on your form inputs!

What we learned:

- How to listen for the focus event on inputs
- How to listen for the blur event on inputs

Event Handling: The Bubbling Principle

Bubbles are little pockets of air that make for an amusing time in the bath. Sometimes, though, bubbles can be annoying -- like when they suddenly pop, or when there are too many and they're overflowing! We can think about Javascript events and their handlers as bubbles that rise up through the murky waters of the DOM until they reach the surface, or the top-level DOM element.

It's important for developers to understand The Bubbling Principle and use it to properly handle events and/or to stop events from bubbling up to outer elements and causing unintended effects.

What is the bubbling principle?

According to this handy bubbling explaineron Javascript.info, The Bubbling Principle means that *when an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors*.Consider the following example HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      window.addEventListener("DOMContentLoaded", event => {
        // Now listen for a click on the body
        document.body.addEventListener("click", event => {
          console.log("The body click event!");
          console.log(event.target.id);
        });
      });
    </script>
  </head>
```

```
  <body>
    <div>
      <div>
        <p id="paragraph">
          If you click on this paragraph, then the function listening on the bo
          will be called.
        </p>
      </div>
    </div>
  </body>
</html>
```

In the `<script>`, you can see the event listener for `DOMContentLoaded`, and inside it, another listener for a `click`event on the `<body>`element of the document accessed through the special property `document.body`. (You could also use `document.querySelector('body')`, too.)

Save the above HTML in a file, and run that file in a browser. Open up the browser console (*right-click -> Inspect -> Console*), click on the `<p>`element, and observe what happens. The message "The body click event!" should appear, then you should see the id `paragraph`printed to the console.

What happened here? The `console.log`shows that an event happened on the `<p>`(i.e. the `event.target`), and yet the click handler on the `<body>`fired -- meaning that the click event on the `<p>`bubbled up to the `<body>`and fired its `click`event, even though there are two levels of `<div>`tags in between!

Whenever you click an element on the page, the browser will *bubble*that event up through every ancestor of the element you clicked on. If you have any event listeners on those elements, those will be called on its way up (all the way up to the window object in fact).

Once the event bubbles all the way back to the top, it actually turns around and goes all the way back down. This is called event `capturing`*and we generally do not use it anymore, it's there because it's the way Internet*

Explorer worked years ago. The function `addEventListener` *doesn't listen for* `capture` *events by default so we mostly don't have to be concerned about these nowadays.*

An event bubbling example

To visualize event bubbling, it might be helpful to watch this short and fun YouTube video on *bubbles inside bubbles inside bubbles*.

Bubble Inside a Bubble Video

We can think of events that happen on nested DOM elements as these nested bubbles. An event that happens on the innermost element bubbles up to its parent element, and that parent's parent element, and so on up the chain. Let's look at another example that demonstrates bubbling.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      window.addEventListener('DOMContentLoaded', event => {
        function handler(e) {
          console.log(e.currentTarget.tagName);
        }
        document.querySelector('main').addEventListener('click', handler);
        document.querySelector('div').addEventListener('click', handler);
        document.querySelector('p').addEventListener('click', handler);
      });
    </script>
  </head>
  <body>
    <main>
      <div>
```

```
        <p>This is a paragraph in a div in a main in a body in an html</p>
      </div>
    </main>
  </body>
</html>
```

If you save this HTML file, open it in a browser, and click on the `<p>`, three different messages should appear in the console: first “P”, second “DIV”, and third “MAIN”. The click event bubbled upwards from the `<p>`element to the`<div>`and finally to the `<main>`.

We could think of this succession of events as bubbles popping. The innermost bubble (the `<p>`element) *popped*(i.e. logged to the console), which caused its parent’s bubble to pop, which caused its parent’s bubble to pop. Since there aren’t any `click`handlers above the `<main>`nothing else happens on the page, but the bubbles would travel all the way up the DOM until they reached the top (`<html>`) looking for event handlers to run.

Stopping event bubbling with stopPropagation()

As stated in the introduction, event bubbling can cause annoying side effects. This MDN doc on Event bubbling and captureexplains what would happen if a user clicked on a `<video>`element that has a parent `<div>`with a show/hide toggle effect. On a click, the video would disappear along with its parent div!

How can you stop this unintended behavior from occurring? The answer is with theevent.stopPropagation()method which stops the bubbling from continuing up the parent chain. Here’s what it would look like on the `<video>`element:

Javascript

```
window.addEventListener('DOMContentLoaded', event => {
  document
    .querySelector('video')
    .addEventListener('click', event => {
      event.stopPropagation();
      video.play();
    });
});
```

Event delegation

While event bubbling can sometimes be annoying, it can also be helpful. The bubbling effect allows us to make use of **event delegation**, which means that we can delegate events to a single element/handler -- a parent element that will handle all events on its children elements.

Say you had an unordered list (`<ul>`) element filled with several list item (`<li>`) elements, and you want to listen for click events on each list item. Instead of attaching a click event listener on each and every list item, you could conveniently attach it to the parent unordered list, like so:

JavaScript

```
window.addEventListener('DOMContentLoaded', event => {
  document
    .getElementById('my-list')
    .addEventListener('click', e => {
      // will print out "This is list item X"
      // depending on which list item is clicked
      console.log(e.target.innerHTML);

      // always prints "my-list"
      console.log(e.currentTarget.id);
```

```
    });
});
```

HTML

```
<ul id="my-list">
  <li>This is list item 1.</li>
  <li>This is list item 2.</li>
  <li>This is list item 3.</li>
  <li>This is list item 4.</li>
  <li>This is list item 5.</li>
</ul>
```

This example is a lot like the first example you saw with the `<p>` inside of a `<div>`, where the click on the `<p>` bubbled up to the `<div>`. In the above example, a click on any `<li>` will bubble up to its parent, the `<ul>`.

When clicked on, a single `<li>` element becomes the event.target-- the object that dispatched the event. The `<ul>` element is the event.currentTarget-- the element to which the event handler has been attached.

Now that you know how to handle events responsibly, go frolic in the bubbles!

What we learned:

- The definition of The Bubbling Principle
- Examples of event bubbling
- How to stop events from bubbling
- How to use bubbling for event delegation