Lesson | Wednesday

Introduction to Programming (/introduction-to-programming) / JavaScript and Web Browsers (/introduction-to-programming/javascript-and-web-browsers) / Event Handling with Event Listeners

Text | Cheat sheet

So far, we've worked with event handler properties to handle events in our code. These properties represent events for the object it belongs to (an HTML element, `document`, or `window` object). An event is just a signal that something has happened in the browser. To use an event handler property, we set its value to a function, and every time the corresponding event happens, the function gets called and all of the code inside of the function runs. In this way, we use event handler properties to write code to react to events that happen in our webpage.

In this lesson, we'll learn a new method of event handling: we'll create **event listeners** in our code to listen for and react to events that happen in our webpage. This process is called **event listening** and it does everything that our event handler properties do, but it has an advantage that makes event listening the preferred method of handling events.

Note that **event handling** is really an umbrella term that describes the processes and tools that developers use to write code that responds to events. In web development, event handling encompasses event handler properties, the concept of event listening with event listeners, and the outmoded form of handling events with HTML event handler attributes.

Let's dive into the new concept of event listening!

Event Listeners

So how do we create an event listener? We'll use object methods that are built-in to the Web APIs that make browser functionality available to web developers. Let's look at an example using the Mad Libs project. We previously used the `onsubmit` event handler property to react to the submission event on our form. Here's the code snippet we'll work with:

js/scripts.js

```
// These are the scripts from the Mad Libs project.
window.onload = function() {
  let form = document.querySelector("form");
  form.onsubmit = function(event) {
    // omitted code to get the value for each form input
    ...
    // omitted code to set the story variables to the value
s we got from the form
    ...
    document.querySelector("div#story").removeAttribute("cl
ass");
    event.preventDefault();
  };
};
```

Take note that we've left some code out, the code that gets the form input values and sets the values of our Mad Libs story. Anytime we leave out code we'll use ellipses `...` in its place. We just want to focus in on the event handler property for the form submission event and learn how to use an event listener instead. Check out our updated code below — the biggest change when using event listeners is using a built-in method instead of a property.

js/scripts.js

```
// These are the scripts from the Mad Libs project.
window.onload = function() {
  let form = document.querySelector("form");
  // new code below
  form.addEventListener("submit", function(event) {
    ...
    ...
    document.querySelector("div#story").removeAttribute("cl
ass");
    event.preventDefault();
  // new code below
  });
};
```

Let's break down this new code:

- We call the method `addEventListener()` on our `form` object to create the event listener. Just like with event handler properties, we target the object that we want to attach the event handler to. This process is often called **registering** the event listener.
- We pass in two arguments to the `addEventListener()` method:
  - `"submit"` is the first argument. This first argument is the name of the event that we want to target, and its data type is always a string. It doesn't have the usual `on` that we add to our event handler properties. Here, we only reference the name of the event.

    - The second argument is a function. This is called the **handler function** that gets called and run when the corresponding event happens. It might surprise you that functions can be arguments just like strings and numbers but this is actually a very powerful feature of JavaScript. In our example code, we're specifically using an anonymous function expression, but we can use other types of functions here instead, like a function declaration (we'll see examples of this in future lessons).
    - Note that this second argument is more generally referred to as a **callback function**, which is any function that is passed into a method or function call as an argument. We'll learn more about callbacks later on in this lesson.
- We close the event listener with `});` :
    - With the closing curly bracket `}` we're closing the function that we pass in as the second argument.
    - With the closing parenthesis `)` we're closing the `addEventListener()` method call.
    - With the semi colon `;` we're following JS convention by adding semicolons at the end of our statement `form.addEventListener(...);`, where the ellipsis `...` represents the two arguments we pass into our event listener method.

We use the same `addEventListener()` method for any event listener that we want to set up. Changing the event that we're targeting is a matter of changing the first argument that we pass into the `addEventListner()` method. Let's see another example. This time, we'll update our `window.onload` event handler property to use an event listener instead.

js/scripts.js

```
// new code below
window.addEventListener("load", function() {
  let form = document.querySelector("form");
  form.addEventListener("submit", function(event) {
    ...
    ...
    document.querySelector("div#story").removeAttribute("cl
ass");
    event.preventDefault();
  });
// new code below
});
```

Just like in our initial example, we've called our `addEventListener()` method on the `window` object, passing in the string `"load"` as the first argument to target the load event, and an anonymous function expression as the second argument as the event handler. Finally, we've made sure to close our `addEventListener()` method call after the end of the function expression close with the `});` series of closing brackets, parens, and semicolon.

Since we're well practiced with event handler properties, function expressions, and using object methods, moving to use event listeners should be a more comfortable stretch for us. Everything we've learned about targeting specific objects (like `window`, `document`, or an HTML element object) to attach an event handler to remains unchanged with event listeners. The same goes for the types of events we can target.

Also, all of the "onevent" properties can be easily translated to the syntax we use to target events with event listeners: simply remove the "on" from the event property and make it into a string. Let's see some more examples:

- `onclick` event handler property is the `"click"` event in our event listener.

- `onmouseover` event handler property is the `"mouseover"` event in our event listener.
- `onkeydown` event handler property is the `"keydown"` event in our event listener.

Let's turn now to the advantages of using event listeners. Then we'll review more information about callback functions.

The Benefits of Using Event Listeners

The main difference between event handler properties and event listeners is that multiple event handlers can be added (or removed as we'll learn in a different lesson) using event listeners. This means that for one event on the same object (usually an HTML element), we can register multiple event handlers.

Let's look at an example. In this example, we'll extend our Mad Libs website by adding new functionality to it. Say we not only wanted our submit event handler to get the form values for our Mad Libs story, but also display a new button that resets the form, and trigger an alert with an advertisement (yuck!). Now we have 3 different reactions to the submit event that we need to handle in our code. Let's see how we would do this with event listeners.

First, we need to add a new button to our HTML. We'll add it right below the closing `</form>` tag:

mad-libs.html

```
...
</form>
<br />
<button type="button" class="hidden" id="reset">Reset</butt
on>
...
```

We've also included `<br />` tag before the new button to create a line break.

Notice that the `type` attribute on the button is set to `"button"`. This attribute value makes the button have no default behavior and do nothing when pressed. We can target a click event on a button with `type="button"`, but not a form submission event, so we cannot use `type="button"` in a button element within an HTML form.

This is in contrast to setting `type="submit"` on a button element, which is meant to be used in form elements and responds specifically to a submission event and has a default behavior of refreshing the page.

Next, let's look at the JS. Pay attention to the comments added to the code snippet below that describe the newly added code.

js/scripts.js

```
window.addEventListener("load", function() {
  let form = document.querySelector("form");
  // we've accessed our button and story elements at the to
p level
  // of the window load event listener to reuse these eleme
nts
  // in multiple locations
  let resetBtn = document.querySelector("button#reset");
  let story = document.querySelector("div#story");

  // the original form submission event listener
  form.addEventListener("submit", function(event) {
    ...
    ...
    // we've updated our code to use the new story variable
    story.removeAttribute("class");

    // take note that we only need to call event.preventDef
ault();
    // once, even though there are 3 different event listen
ers for the
    // form submission event
    event.preventDefault();
  });

  // new event listener for form submit event to show reset
button
  form.addEventListener("submit", function() {
    reset.removeAttribute("class");
  });

  // new event listener for form submit event to show adver
tisement
  form.addEventListener("submit", function() {
    window.alert("Do you need a new computer? Visit www.sup
erextracomputersales.com to find the best deals!");
  });

  // new event listener for click event on reset button to
  // reset form values
  resetBtn.addEventListener("click", function() {
```

```
    story.setAttribute("class", "hidden");
    document.getElementById("person1Input").value = null;
    document.getElementById("person2Input").value = null;
    document.getElementById("animalInput").value = null;
    document.getElementById("exclamationInput").value = nul
l;
    document.getElementById("verbInput").value = null;
    document.getElementById("nounInput").value = null;
  });
});
```

As we can see in the code snippet, we've added 3 separate event listeners on the form submission event. The reason why using the `onsubmit` event handler property would fail in this case is because the property can only be set to one value, and anytime we reset the value, it overwrites the previous one.

So, if we updated the same code above to use event handler properties, only the last `onsubmit` event handler property would be registered. Let's see what this looks like. Pay attention to the comments as you read the following code snippet.

js/scripts.js

```
window.addEventListener("load", function() {
  let form = document.querySelector("form");
  let resetBtn = document.getElementById("reset");
  let story = document.getElementById("story");
  // the original form onsubmit event handler
  form.onsubmit = function(event) {
    ...
    ...
    story.removeAttribute("class");
    event.preventDefault();
  };

  // this onsubmit event handler overwrites the one above i
t
  form.onsubmit = function() {
    resetBtn.removeAttribute("class");
  };

  // this onsubmit event handler overwrites the one above i
t
  // this is the only remaining onsubmit event handler for
the form
  form.onsubmit = function() {
    window.alert("Do you need a new computer? Visit www.sup
erextracomputersales.com to find the best deals!");
  };

  resetBtn.addEventListener("click", function() {
    story.setAttribute("class", "hidden");
    document.getElementById("person1Input").value = null;
    document.getElementById("person2Input").value = null;
    document.getElementById("animalInput").value = null;
    document.getElementById("exclamationInput").value = nul
l;
    document.getElementById("verbInput").value = null;
    document.getElementById("nounInput").value = null;
  });
});
```

In the above code snippet, we should be able to track that the first two `onsubmit` event handlers (for getting the form values/setting the story values and showing the reset button) have both been overwritten by the very last `onsubmit` event handler. Now when we submit the form, our webpage will show the advertisement, but it won't display the story or reset button.

You may be thinking, so what! Couldn't we just combine each reaction to the form submission into one event handler function? Something like this:

js/scripts.js

```
window.addEventListener("load", function() {
  let form = document.querySelector("form");
  let resetBtn = document.getElementById("reset");
  let story = document.getElementById("story");
  // the original form submission event handler
  form.onsubmit = function(event) {
    // get form values and set story values
    ...
    ...
    story.removeAttribute("class");
    event.preventDefault();

    // show reset button
    resetBtn.removeAttribute("class");

    // show ad
    window.alert("Do you need a new computer? Visit www.sup
erextracomputersales.com to find the best deals!");
  };

  resetBtn.addEventListener("click", function() {
    story.setAttribute("class", "hidden");
    document.getElementById("person1Input").value = null;
    document.getElementById("person2Input").value = null;
    document.getElementById("animalInput").value = null;
    document.getElementById("exclamationInput").value = nul
l;
    document.getElementById("verbInput").value = null;
    document.getElementById("nounInput").value = null;
  });
});
```

Yes, we can do this, and we can use code commentary to label the different reactions we are making to the form submission.

We can also do this using the `addEventListener()` method, just like the following code snippet demonstrates. Note that ellipses `...` represent unchanged code that's omitted from the example.

```
window.addEventListener("load", function() {
  let form = document.querySelector("form");
  let resetBtn = document.getElementById("reset");
  let story = document.getElementById("story");
  // the original form submission event handler
  form.addEventListener("submit", function(event) {
    // get form values and set story values
    ...
    ...
    story.removeAttribute("class");
    event.preventDefault();

    // show reset button
    resetBtn.removeAttribute("class");

    // show ad
    window.alert("Do you need a new computer? Visit www.sup
erextracomputersales.com to find the best deals!");
  });
  ...
});
```

If we can do this, it begs the question: why would we NOT do this? The answer has to do with how we organize our code so that it scales well and it is easy to understand.

When we talk about code organization and choosing between putting code into one function or into multiple functions (or event handlers), we should be asking ourselves, "what would make my code easier to read and understand?". There will be some differences when we answer this question, but a good guideline for code organization is to separate code by what it does in our application.

With our Mad Libs website's expanded functionality, we now have 3 reactions to the form submission: get and set the Mad Lib story values, show an ad, and display a reset button. As far as organizing our code, we have two main options:

- Organize the three reactions into one submit event handler function based on its shared purpose of responding to the submit event.
- Organize the three reactions into three separate submit event handler functions based on recognizing that each reaction has a different purpose in our webpage (like showing an ad versus displaying the Mad Libs story).

So what's the correct choice? Well, both options work when our scripts are as small as ours are. That's a fact.

However, if our submit event triggered 5 or 10 different reactions, then our code will likely be easier to understand if we separate our reactions into multiple event listeners instead of using one very large event listener. This is an example of using multiple event listeners to improve code organization and how readable it is.

Or, what if we wanted the advertisement to show only once, and not every time the form is submitted — well, we'd need to separate the advertisement into its own event listener that we could cancel after it has been run once. If we had originally chosen to separate each reaction into its own event listener, then we could easily update our code to address the new requirement (to only show the ad once). This is an example of using multiple event listeners to improve how well our code scales as requirements change.

Because using event listeners with the `addEventListener()` method allow us to attach multiple handlers to the same event, they are considered the best choice for applications that scale (get larger and change in requirements); they gives us the flexibility at any point to change and expand the functionality of our website while maintaining good code organization. So, **going forward, you should choose to use event listeners over event handler properties.**

You should also pause periodically to ask yourself "what would make my code easier to read and understand?". We've just started to learn about new ways to organize our code, so if you don't feel

comfortable with this decision making process quite yet, that's entirely normal and expected. The idea is to start reflecting on these questions, but not to get hung up on finding the most efficient or "perfect" code organization. It takes practice as well as actually building larger and more complex applications to get the hang of good code organization.

Also take note that while there are always multiple solutions when it comes to code organization (and benefits and drawbacks to consider for each), there are helpful standbys to always incorporate into your code:

- Use code comments to describe the different functions in your scripts. You don't need to go overboard, but it can be helpful when there are longer chunks or when you have a long function that does a lot of things.
- Use descriptive variable and function names.
- Use correct spacing, new lines, and indentation. This helps immensely for readability.

More About Callback Functions

Callbacks or **callback functions** is a concept in computer programming that many languages implement. A callback function (or just "callback") is any function that is passed to another function or method as an argument. This callback function isn't called immediately, though. The function/method that receives the callback function as an argument is expected to call ("call back to") the function at a later time.

We can see this process at work in the `addEventListener()` method. Let's review this method. `addEventListener()` has two required parameters. In pseudocode, this looks like:

```
// this is pseudocode!
target.addEventListener(eventName, callbackFunc);
```

- `target` is the object we are targeting — an HTML element, the `document` object, or the `window` object.
- `eventName` is the first parameter. Here we'll pass in a string with the name of the event we're creating the listener for.
- `callbackFunc` is the second parameter. Here we'll pass in a function that contains all of the code that we want to run in reaction to the event.

Even though we are passing a callback function as the second argument of the `addEventListener()` method, this function only gets called when the corresponding event happens. Not immediately, but at a later time. This timing distinction is important and a big reason why callbacks exist, which is to handle asynchronous JavaScript code. We won't get into that now, but we will revisit asynchrony and callbacks later in the program, so there will be more opportunity to become familiar with this concept.

Also, take note that categorizing functions as "callbacks" does not have to do with the type of a function (declaration, expression, or another — how we define the function). Instead it has to do with a function's application — where we are using it in our code. Any function that is used as an argument for another function automatically becomes a callback function.

It will be unlikely that you ever create a custom function in the program that takes a function as an argument. There won't be a lot of use cases for that. However, you will work with some built-in methods and functions (of JavaScript or Web APIs) that require a function for an argument (a callback).

If this sounds complicated and confusing, that's totally expected — especially if you're new to coding. There are a lot of complicated things going on under the hood. Learning to drive is one thing —

and it's not too hard. But learning how to build and maintain a car is a lot more work! We'll revisit JavaScript terminology and the innerworkings of both JS and our web browser regularly, so there will be more opportunity for review and practice.

Summary and Completed Code for the Mad Libs Project

In this lesson we learned about event listening. Some key topics we covered are:

- Event listening is another method of event handling. **Event handling** is an umbrella term that describes the processes and tools that developers use to write code that responds to events.
- **Event listening** is the process of creating event listeners in our code to listen for and react to events that happen in our webpage.
- When we call `addEventListener()` on a target (a DOM element, `window`, or `document`), this process is called **registering** the event listener.
- We can create an event listener with the `addEventListener()` method, which takes two arguments:
  - The first argument is the event name as a string, like `"click"` or `"submit"`.
  - The second argument is the **handler function**, the function that handles reacting to the event.
- The benefit of using event listeners is that we can create multiple handlers for the same event on the same target (a DOM element, `window`, `document`, or otherwise). This improves our code organization and how easy it is to read and understand, and it also makes our code able to scale well. Event listeners are considered the recommended way to set up event handling in applications.
- Any function that is passed into another function/method as an argument is called a **callback function**.

- The "handler" function that we pass into the `addEventListener()` method is a callback function.
- Callback functions are all about the application of functions — where they are being used in our code — and they are important to asynchronous JavaScript, which we'll learn about down the road.

To view the completed code for our Mad Libs project visit the cheat sheet. The code includes the new functionality of the reset button and the advertisement, and a separate event listener for each reaction to the form submission event.

Previous (/introduction-to-programming/javascript-and-web-browsers/practice-more-branching)
Next (/introduction-to-programming/javascript-and-web-browsers/using-function-declarations-in-event-handling)

Lesson 63 of 75
Last updated March 24, 2023

disable dark mode

(http://www.epicodus.com)