LAB 9

JavaScript 2: DOM and Events

What You Will Learn

- Accessing and modifying DOM HTML elements using JavaScript

- Creating event listeners to react to events

- Tools and tricks to help you develop JavaScript

Approximate Time

The exercises in this lab should take approximately 90 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

1   The starting `lab09` folder has been provided for you (within the zip folder downloaded from Gumroad).

*Note: these labs use the convention of `blue background` text to indicate filenames or folder names and **`bold red`** for content to be typed in by the student.*

THE DOCUMENT OBJECT MODEL (DOM)

In the previous lab, you used the `document.write()` function as a way to manipulate markup using JavaScript. While fine from a learning JavaScript perspective, it is generally not how one typically uses JavaScript. In the first part of this lab, you will learn how to use the Document Object Model (DOM) with JavaScript as a "better" way of manipulating content.

Exercise 9.1 — THE DOCUMENT OBJECT

1   Examine `lab09-ex01.html` in your browser and then in your editor of choice.

2   Display the JavaScript console in your browser and enter the following command in the console:

```
console.dir(document)
```

This provides a hierarchical view of the document object, which allows you to examine the different properties and methods within this object.

3   Examine some of these properties and methods. Notice that some of these properties are hierarchical themselves (e.g., the `all` property, which is a collection of all the elements in the document).

Working with the DOM means working with the `document` object.

Exercise 9.2 — BASIC DOM SELECTION

1   Examine `lab09-ex02.html` in your browser. This is the same file except for the addition of a `<script>` element.

Notice that it references a JavaScript file at the end of the markup. To better understand the rest of the steps in this exercise, you should continually refer back to the markup.

2   Edit `js/lab09-ex02.js` by adding the following and then testing `lab09-ex02.html` in browser.

```
const msg = document.getElementById("msg");
msg.value = "this text is generated by JavaScript";
```

This selects the `<textarea>` element and sets its value.

3   Add the following code and test.

```
const legend = document.getElementById("title");
legend.textContent = "Dynamically generated";
```

The textContent property allows you to modify the content of an element.

4   Replace the two previous lines with the following single line and test:

```
document.getElementById("title").textContent =
    "Dynamically generated";
```

JavaScript developers often try to reduce the number of variables that get added into scope by not defining variables if they are not needed.

5   Comment out the previous line and add the following code and test.

```
const legend = document.getElementById("title");
legend.innerHTML = "Dynamic with <span>HTML</span> added";
```

Notice that you can "inject" HTML into the content of an element. As we shall see below, while easy, we often want to take a more complicated route when adding HTML elements to a document in order to ensure new content gets added to the DOM tree.

6   Modify the code as follows and test.

```
const legend = document.getElementsByTagName("legend");
legend[0].innerHTML = "Dynamic with <span>HTML</span> added";
```

Here you are using a different selection method, that returns a list of <legend> elements. It doesn't matter if there is only a single <legend> element, the getElementsByTagName method always returns a HTMLCollection element (an array-like object that can be iterated like an array).

7   Add the following and test:

```
const labels = document.getElementsByClassName("input-labels");
for (let lab of labels) {
    lab.style.backgroundColor = "#FFF0F0";
}
```

Since getElementsByClassName returns a HTMLCollection, we can iterate through it. In this example, we programmatically alter the background-color CSS property.

8   Add the following code and test.

```
document.querySelector("#msg").style.color = "#E4F0F5";
```

The querySelector method allows you to select a single element using a CSS selector. Unless you need to support older (pre-2012) browsers, we recommend using querySelector and querySelectorAll instead of getElementById, getElementsByClassName, and getElementsByTagName.

9   Add the following code and test.

```
const inputs = document.querySelectorAll("input[type=text]");
for (let inp of inputs) {
    inp.style.backgroundColor = "#E4F0F5";
}
```

The querySelectorAll method allows you to select multiple elements using a CSS selector. It returns a NodeList, which is an array-like object that can be iterated.

10   Add the following code and test. Examine the error message in the console.

```
// this will generate a runtime error ... why?
inputs.style.color = "red";
```

Since querySelectorAll returns a NodeList, the style property will be undefined.

11   Comment out the previous line of code, and then add the following code and test.

```
const notOkay = document.querySelector("#msg000");
alert(notOkay);
```

Notice that if the selector passed to querySelector doesn't match any elements, it returns null.

TEST YOUR KNOWLEDGE #1

Examine `lab09-test01.html` and then open `lab09-test01.js` in your editor. Modify the JavaScript file to implement the following functionality.

1   Use `getElementById` to add a border via CSS to the `<ul>` element with the name "thumb-list".

2   Use `querySelector` to set the value property of the `<textarea>` to the `textContent` of the `<p>` element within the `<fieldset>`.

3   Use `querySelectorAll` to add a box shadow to each of the `<img>` elements within the `<ul>` element. The CSS property name is `box-shadow` so the JavaScript DOM property name will be `boxShadow`. To see a sample `box-shadow`, look at the example for the `box` class in `lab09-ex01.css`. Remember that you will need to use a loop. The result should look similar to that shown in Figure 9.1.
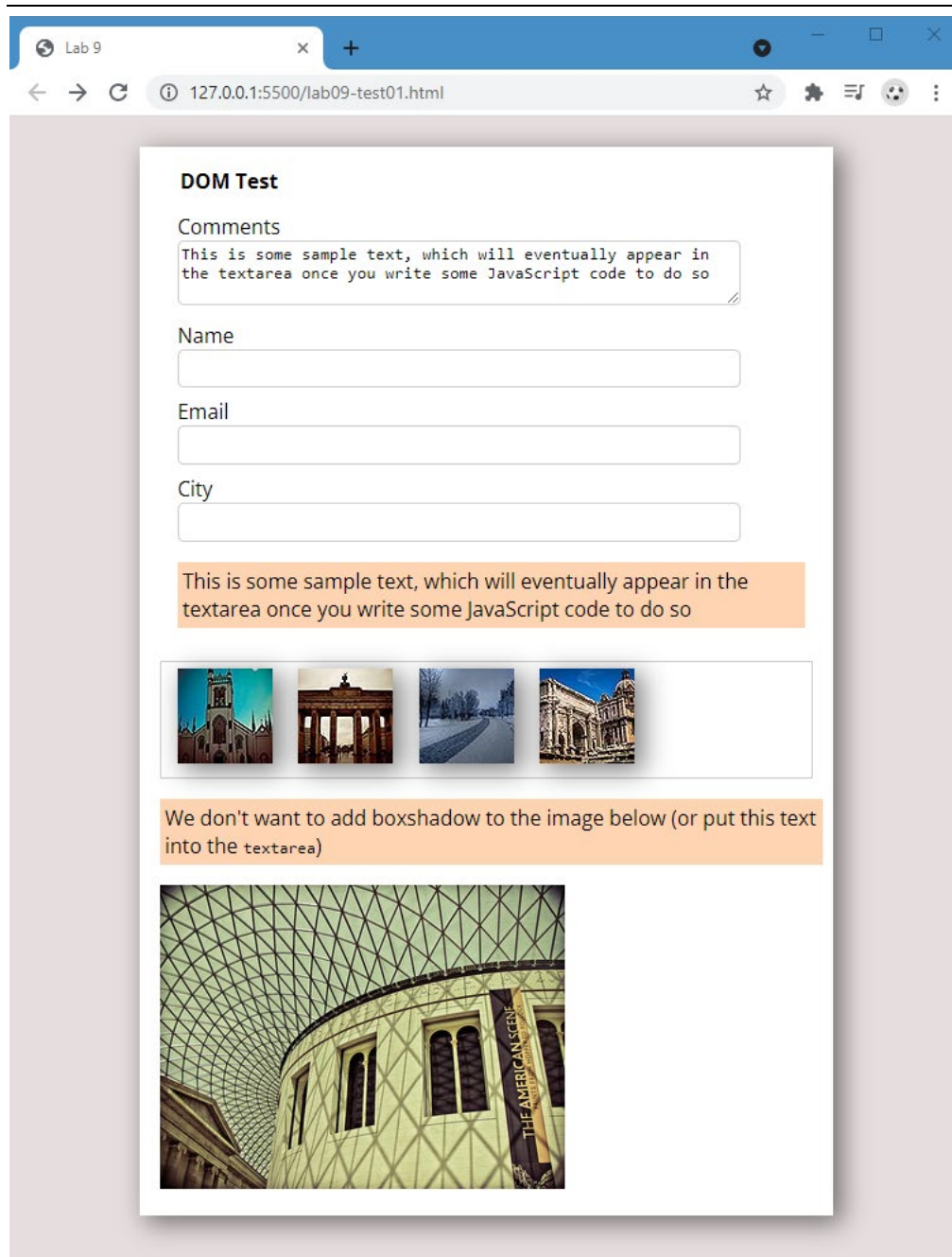
Figure 9.1 – Finished Test Your Knowledge #1

EXERCISE 9.3 — MODIFYING THE DOM

1   Examine `lab09-ex03.html` in your browser. This is the same file as previous exercise. Open `lab09-ex03.js`, add the following, and test.

```
// select the <ul>
const list = document.querySelector("#thumb-list");

// create list item <li>
const item = document.createElement("li");

// create <img> element and set its src attribute
const img = document.createElement("img");
img.setAttribute("src", "images/8710320515.jpg");

// nest the <img> in the <li>
item.appendChild(img);

// add the <li> to the already existing <li> elements
list.appendChild(item);
```

This illustrates the proper way to dynamically add HTML elements to a document. As you can see, this code adds a new image to your list.

2   Comment out the last line in the last step and add the following and test.

```
// add another one at the start of the list
list.insertAdjacentElement('afterbegin',img);
```

As you can see, this adds the new image to the beginning of the list.

3   Add the following lines and test.

```
// remove the last <li>
document.querySelector("#thumb-list li:last-child").remove();
```

This deletes the last <li> elements.

EXERCISE 9.4 — CHANGING CSS CLASSES

1   Examine `lab09-ex04.html` in your browser. In this exercise, you will make the element with the sign-in form appear after a time delay.

2   Modify the `<aside>` element by adding the following class and test.

```
<aside class="centered hidden">
```

The form no longer appears because of the addition of the hidden class.

3   Examine `lab09-ex04.css`. Notice the two classes `hidden` and `visible`.

4   Add the following code to `lab09-ex04.js` and test.

```
setTimeout( function () {
    let aside = document.querySelector('aside');
    aside.className = 'visible';
}, 3000);
```

After a three second delay, this sets the class for the `<aside>` to `visible`. Unfortunately it also removes the centered class since setting className replaces any existing classes.

5   Modify the code as follows and test.

```
setTimeout( function () {
    let aside = document.querySelector('aside');
    aside.classList.remove('hidden');
    aside.classList.add('visible');
}, 3000);
```

The classList() method allows you to specify multiple classes (className only allows one).

6   As an alternate approach, simplify the code using the toggle method as follows and then test.

```
setTimeout( function () {
    let aside = document.querySelector('aside');
    aside.classList.toggle('hidden');
}, 3000);
```

Each call to toggle() either adds or removes the class from the element.

TEST YOUR KNOWLEDGE #2

In Chapter 8, in the fourth Test Your Knowledge, you used the document.write() method to output markup as strings. In this exercise, you will use DOM methods to dynamically add similarly structured content instead so that your output looks similar too that shown in Figure 9.2.

1   Examine lab09-test02.html. It provides the sample markup for a single photo. You will replace that markup with JavaScript in the following steps. For now, either comment out the markup or cut it and paste into a temporary file for referencing later.

2   Modify lab09-test02.js in your editor. You will need to select the <section> element that has an id=parent. Loop through the photos array and create a <figure> element that will get appended to the parent element. You may want to examine photos.json again to reacquaint yourself with it s structure.

3   Within that loop, you will need to dynamically create the <img> element using the appropriate DOM methods. Populate its src and alt attributes using the title property.

4   After the image is created, you will need to create the <figcaption> element. This element will contain the following child elements which will also be dynamically generated from the photo data: <h2>, <p>, and <span> elements. For the <span> elements, you will need to loop through the colors array inside each photo object and set the backgroundColor property of the <span> to the hex value. Append the <figcaption> to the <figure>.

5   Be sure to append the created elements to the appropriate parent. Because of all the nested elements, this code can get pretty messy. Be sure to make use of your own functions to keep your code more manageable (and expressive).
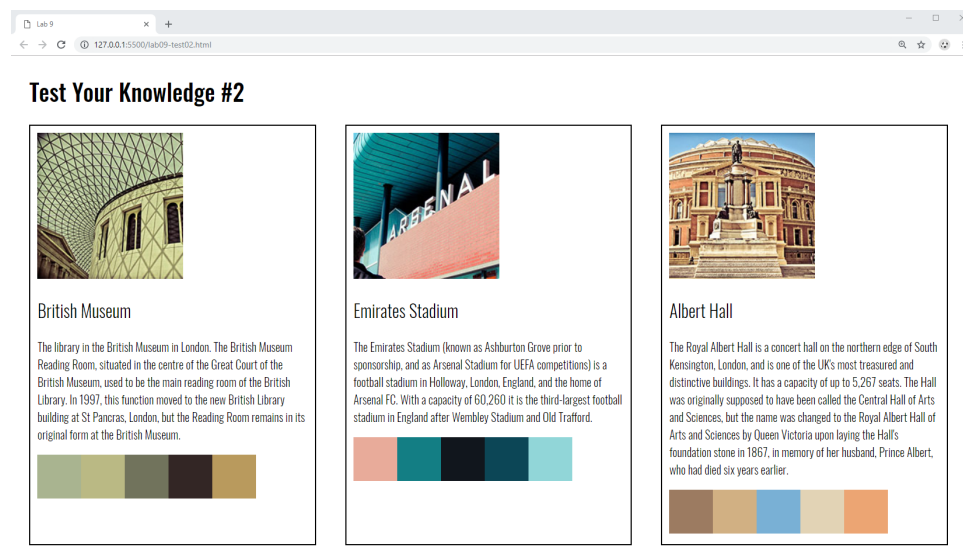
Figure 9.2 – Finished Test Your Knowledge #2

EVENT HANDLING

One of the most common uses of JavaScript is to use it to respond to different user-initiated actions. These are generally referred to as **events** and the JavaScript that responds to these events are called **event handlers**. As the textbook indicates, the preferred approach is to use event listeners.

EXERCISE 9.5 — SIMPLE EVENT HANDLING

1   Examine `lab09-ex05.html` in your browser. Open `lab09-ex05.js` then add the following and test.

```
function simpleHandler(event) {
    alert("button was clicked");
}
const btn = document.getElementById("firstButton");
btn.addEventListener("click", simpleHandler);
```

This defines an event handler function and assigns it to the click event of the first button. It is quite common to instead use an anonymous function for the event handler, as shown in the next steps.

2   Add the following code and test.

```
const btn = document.getElementById("firstButton");
btn.addEventListener("click", function () {
    alert("a different approach but same result");
});
```

While this takes a bit of getting used to, it is an extremely common way of doing JavaScript event coding.

3 What if we wanted the same event handler for all three buttons? Comment out the code added in previous step, then add the following code and test.

```
// illustrates one event handler for multiple events
const buttons = document.querySelectorAll(".card button");
for (let btn of buttons) {
    btn.addEventListener("click", function () {
        alert("now just one handler");
    });
}
```

Notice how this selects a collection of elements and then loops through them to assign event handlers. But what if we wanted to know more about the element that generated the event?

4 Modify the code as follows and test.

```
for (let btn of buttons) {
    btn.addEventListener('click', function (e) {
        let para = e.target.previousElementSibling;
        alert(para.textContent);
    });
}
```

Notice the use of the event object parameter (the variable named e). It provides us with information about the generating event.

5 Add the following code and test by moving mouse in and out of the images.

```
// illustrates the assigning of multiple handlers to the same object
const images = document.querySelectorAll(".card img");
for (let img of images) {
    img.addEventListener('mouseover', function (e) {
        e.target.classList.add('sepia');
    });
    img.addEventListener('mouseout', function (e) {
        e.target.classList.remove('sepia');
    });
}
```

This applies a CSS filter dynamically to the card image when you mouse over the image, and removes the filter when the mouse is moved out of the image. Recall from Exercise 9.4, that you can use classList.toggle() instead.

EXERCISE 9.6 — RESPONDING TO LOAD EVENTS

1 In `lab09-ex06.html` (which is the same file as the finished version of the previous exercise), move the `<script src="js/lab09-ex06.js"></script>` from the end of the document to within the `<head>` element instead.

2 Test. Display the JavaScript console in your browser

Depending on your browser and version, you may or may not get a console error. The code will likely no longer work. Why?

3 The reason it no longer works is that the script is now executed **before** the DOM is fully loaded. To make the script execute **after** the DOM is loaded we must make use of the page `load` event. Try this yourself by moving the event handling code inside of a listener for the `load` event as follows (some code omitted) and then test:

```javascript
// don't set up event handlers until DOM is loaded
document.addEventListener('DOMContentLoaded', function () {

    // illustrates one event handler for multiple events
    const buttons = document.querySelectorAll(".card button");
    for (let btn of buttons) {
        btn.addEventListener('click', function (e) {
            let para = e.target.previousElementSibling;
            alert(para.textContent);
        });
    }
    ...
});
```

EXERCISE 9.7 — EVENT PROPAGATION

1 Examine `lab09-ex07.html` in your browser. Open `lab09-ex07.js` then add the following.

```
document.addEventListener("DOMContentLoaded", function() {

    const containers = document.querySelectorAll('.container');
    for (cont of containers) {
        cont.addEventListener('click', function(e) {
            console.log(`${e.target.nodeName} was clicked –
                Propagated to ${e.currentTarget.nodeName}`);
        });
    }
}
```

This adds this event handler to every element in the document. See Figure 9.13 in the book for explanation of difference between e.target and e.currentTarget.

2 Test by clicking on one of the `<span>` elements and examine the output in the console. Experiment by clicking on other elements. You may want to use the clear console button.

Notice how the click event propagates up the DOM tree to the different ancestors of the element that was clicked.

3 Add the following code (within the `DOMContentLoaded` handler) then test by clicking one of the `<span>` elements.

```
const spans = document.querySelectorAll('span');
for (s of spans) {
    s.addEventListener('click', function (e) {
        console.log('special extra handler for span');
        e.stopPropagation();
    });
}
```

This stops the event from propagating up the ancestor chain.

4 Modify the code as follows.

```
document.addEventListener("DOMContentLoaded", function() {

    const containers = document.querySelectorAll('.container');
    for (cont of containers) {
        cont.addEventListener('click', function(e) {
            console.log(`${e.target.nodeName} was clicked –
                Propagated to ${e.currentTarget.nodeName}`);
        }, { capture: true } );
    }
}
```

This changes the way this event "moves". In the default mode, events propagate from the closest object "up" to its ancestors. By turning on capture, events move in the opposite direction, from the furthest ancestor "down" to the closest object.

5   Add the following code (within the `DOMContentLoaded` handler) then test by clicking one of the `<span>` elements.

```
const section = document.querySelector('section');
section.addEventListener('click', function (e) {
    e.stopPropagation();
}, {capture: true});
```

Now the event "stops" when it reaches the `<section>` element and never makes it to the "lower" event handler for the `<span>` elements.

EXERCISE 9.8 — EVENT DELEGATION

1   Examine `js/stocks.json`. Recall from Chapter 8 that JSON format is often used to transport data for JavaScript consumption. Here we will simply be reading the JSON data from a file. Next chapter we will be receiving it from an external API.

2   Add the following code to `lab09-ex08.js` and then test by viewing `lab09-ex08.html` in the browser.

```
const stocks = JSON.parse(stockData);
console.dir(stocks);
```

You should be able to see stocks is an array of objects, each one representing a single stock.

3   Add the following.

```
document.addEventListener("DOMContentLoaded", function() {

    const main = document.querySelector('main');
    // loop through stock data and output each one in a div
    for (let stock of stocks) {
        // create img and div container for img
        let img = document.createElement('img');
        img.setAttribute('src',
                `images/logos/${stock.symbol}.svg`);
        img.setAttribute('title', stock.symbol);
        let div = document.createElement('div');
        div.setAttribute('class','item');

        // add img to div and div to <main>
        div.appendChild(img);
        main.appendChild(div);
    }
});
```

You should now see about 70 company logos.

4 When the user clicks on a logo, we want to display more information about that company. Instead of looping through each image and assigning event handlers to each, we will instead use event delegation. Add the following after the `for` loop from previous step:

```javascript
// use delegation to handle click events for all the images
document.querySelector('main')
  .addEventListener('click', function (e) {
    // verify user has clicked on image within <main>
    if (e.target && e.target.nodeName.toLowerCase() == 'img') {
      populateAside(e);
    }
  }
);
```

5 Now write the action for the click. In our case, you are going to populate the table in the `<aside>` element with the relevant company data contained in our `stocks` array. Add the following and test.

```javascript
function populateAside(e) {
  // determine the clicked symbol name from clicked image
  let clickedSymbolName = e.target.getAttribute('title');

  // search through stocks array looking for symbol that matches
  const foundSymbol = stocks.find(function(element) {
      return element.symbol === clickedSymbolName;
  });
  // display aside (hidden initially)
  let aside = document.querySelector('aside');
  aside.style.display = 'block';

  let logo = document.querySelector('#logo img');
  let symbol = document.querySelector('#symbol');
  let name = document.querySelector('#name');
  let sector = document.querySelector('#sector');
  let sub = document.querySelector('#sub');

  // populate table with data
  logo.setAttribute('src',
      `images/logos/${foundSymbol.symbol}.svg`);
  symbol.textContent = foundSymbol.symbol;
  name.textContent = foundSymbol.name;
  sector.textContent = foundSymbol.sector;
  sub.textContent = foundSymbol.subIndustry;
}
```
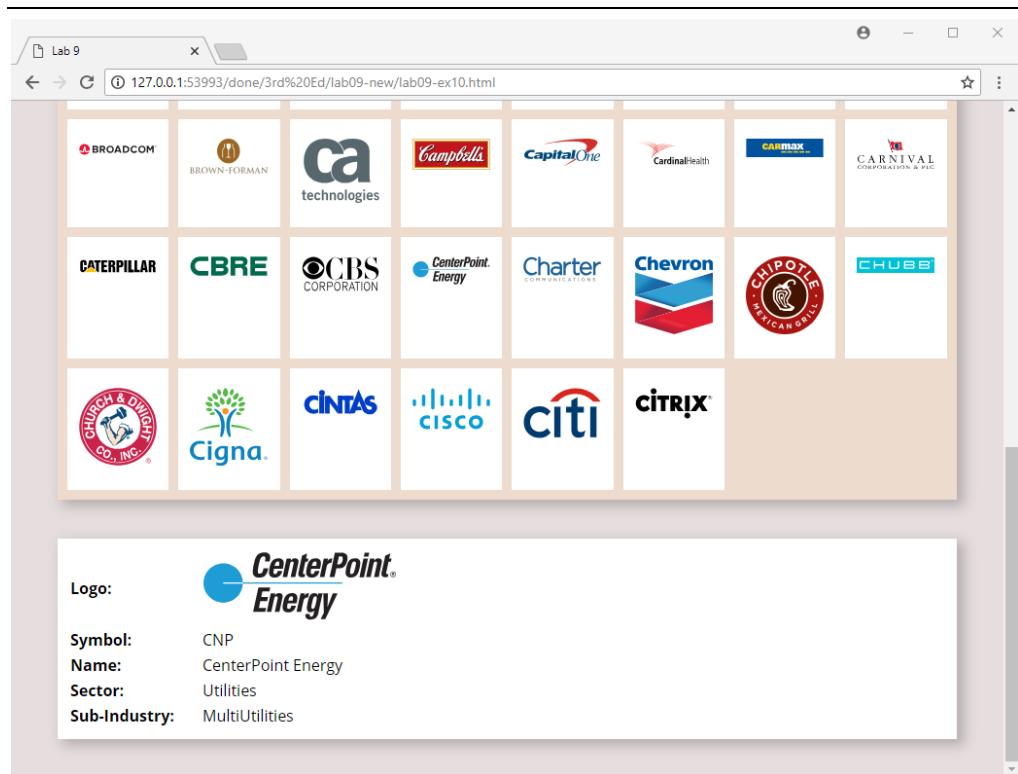
Figure 9.3 – Finished Exercise 9.8

TEST YOUR KNOWLEDGE #3

Examine `lab09-test03.html`, view in browser, and then open `lab09-test03.js` in your editor. Modify the JavaScript file to implement the following functionality.

1 Add an event handler for the click event of each `<div>` with the `panel` class (that is, **don't** use event delegation for this exercise). Be sure to assign this event handler after the DOM is loaded (i.e., after the `DOMContentLoaded` event).

In this event handler you are going to either add or remove the class 'open' from the clicked panel (this will either expand or shrink the panel back to its original size. This can be achieved easily using the `toggle()` method of the `classList` property. The result should look similar to Figure 9.4 when panel is opened with a click.

This exercise inspired from Wes Bos's JavaScript 30 sample project (https://javascript30.com/), and is used with permission.
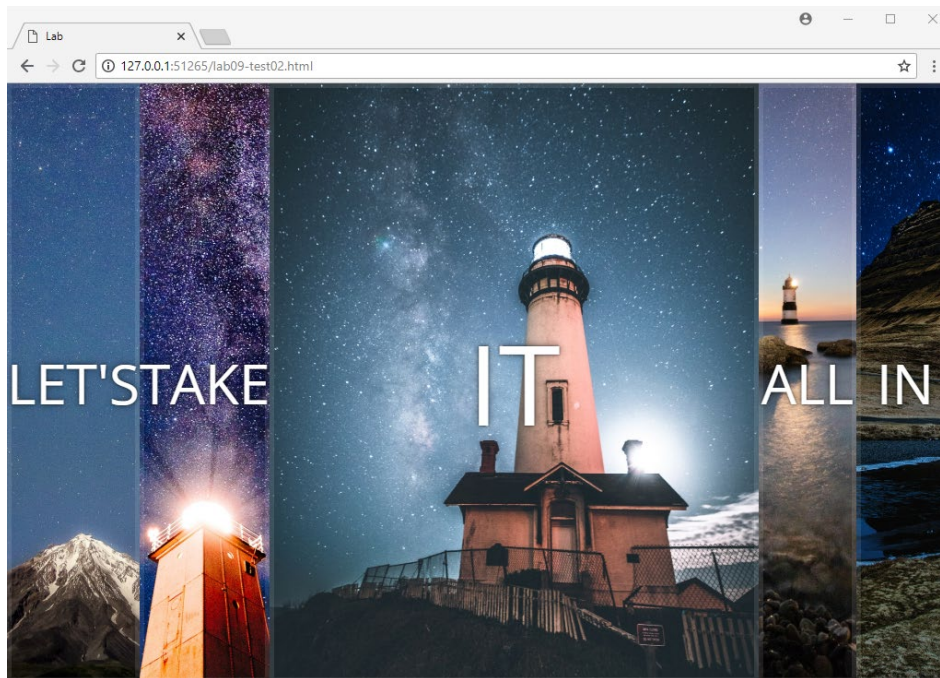
Figure 9.4 – Finished Test Your Knowledge #3

EXERCISE 9.9 — RESPONDING TO KEYBOARD EVENTS

1 Examine `lab09-ex09.html` in your browser.

Notice the series of <audio> elements. Each of these elements contains a data-key attribute. The data-* attribute is a mechanism within HTML5 for adding custom information to an element. In this example, it will be used by our JavaScript for looking up the appropriate sound to play.

2 Open `lab09-ex09.js` and add the following code and test by pressing the 'g' key.

```javascript
document.addEventListener("DOMContentLoaded", function() {

    // when key is pressed play a sound if correct key
    document.addEventListener('keydown', playSound);

    function playSound(e) {
        // initially let's just play the sound for data-key=g
        const audio = document.querySelector(
            `audio[data-key="g"]`);

        // since audio sound takes some time to play, we can reset
        // the current time to zero so that it is more responsive to
        // multiple fast key presses
        audio.currentTime = 0;

        // now play sound at beginning
        audio.play();
    }
});
```

There is not a lot of keyboard handling here: instead the focus is on selecting an <audio> element and playing the sound file specified in the element.

3 Modify the code as follows and test by pressing the keys a, s, d, f, g, h, j, k, l

```
function playSound(e) {
    // first determine the key that was pressed from event
    const keyPressed = e.key;

    // select the <audio> element for the pressed key
    const audio =
        document.querySelector(
            `audio[data-key="${keyPressed}"]`);

    // if key doesn't have <audio> element then exit
    if (!audio) return;
    ...
}
```

All nine sounds sound play when their associated key is pressed. Right now, there is no visual cue for the key being pressed. The next steps will make use of some already existing CSS classes to add some visual transition effects.

4 Modify the code as follows and test.

```
    // if key doesn't have <audio> element then exit
    if (!audio) return;

    // select the <div> element corresponding to the pressed key
    const div =
        document.querySelector(
            `div.key[data-key="${keyPressed}"]`);

    // add glowing box class to it
    div.classList.add('playing');
```

A glowing yellow box should now be showing up around the `<div>` on screen associated with the pressed key. The `<div>` also grows a bit in size. However the effects never goes away.

5 You can make the yellow box go away by responding to another event. Before doing that, open `lab09-ex09.css` and look at the `key` and `playing` CSS classes. Notice the `transition` property in the `key` class? It specifies the length of time for all changes in visual state in the element assigned to this class. It is assigned to the `<div>` elements with the key and sound name (e.g., `<div data-key="g" class="key">`). The code in the previous step dynamically assigned the `.playing` class to the same `<div>` element, which means the `scale`, `border-color`, and `box-shadow` properties transitioned across .07sec. The ending of that transition is an event that can be responded to in JavaScript, which is covered in the next step.

6   Modify the code as follows and test:

```
window.addEventListener("DOMContentLoaded", function() {
   // when key is pressed play a sound if correct key
   document.addEventListener('keydown', playSound);

   // to remove the box around the div for the key, we will listen
   // for any transition-end event (in this case, used by the .key
   // class elements)
   const keys = document.querySelectorAll('.key');
   for (let key of keys) {
      key.addEventListener('transitionend', removeTransition)
   }

   function removeTransition(e) {
      // skip if not a transform transition
      if (e.propertyName !== 'transform') return;
      // right transition so remove the .playing class from .key
      this.classList.remove('playing');
   }

   ...
```

*This exercise inspired from Wes Bos's JavaScript 30 sample project
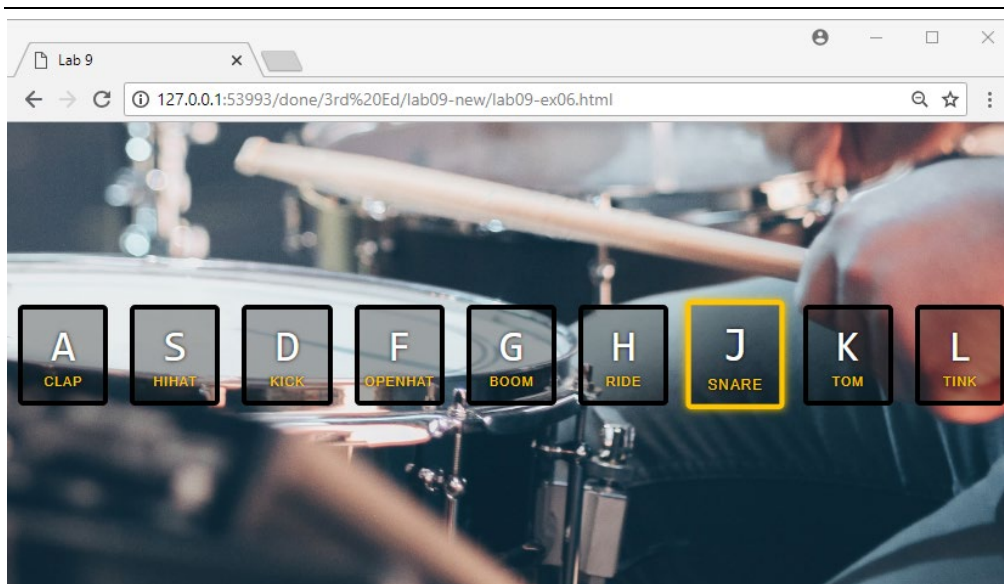(https://javascript30.com/), and is used with permission.*

Figure 9.5 – Finished Exercise 9.6

EXERCISE 9.10 — DEBUGGING EVENTS

1 You are going to continue working with the file from the previous exercise.

2 In Chrome, to access the debugger, you have to click on the Sources tab within the Console and then open the JavaScript file (`lab09-ex09.js`) that you wish to debug.

You will now add a breakpoint to your script by clicking to the left of the first non-comment line within `playSound()` function.

3 Try refreshing the page and then pressing the 'g' key. The line with the breakpoint will be highlighted as shown in Figure 9.4. You can now examine the state of local and global variables.
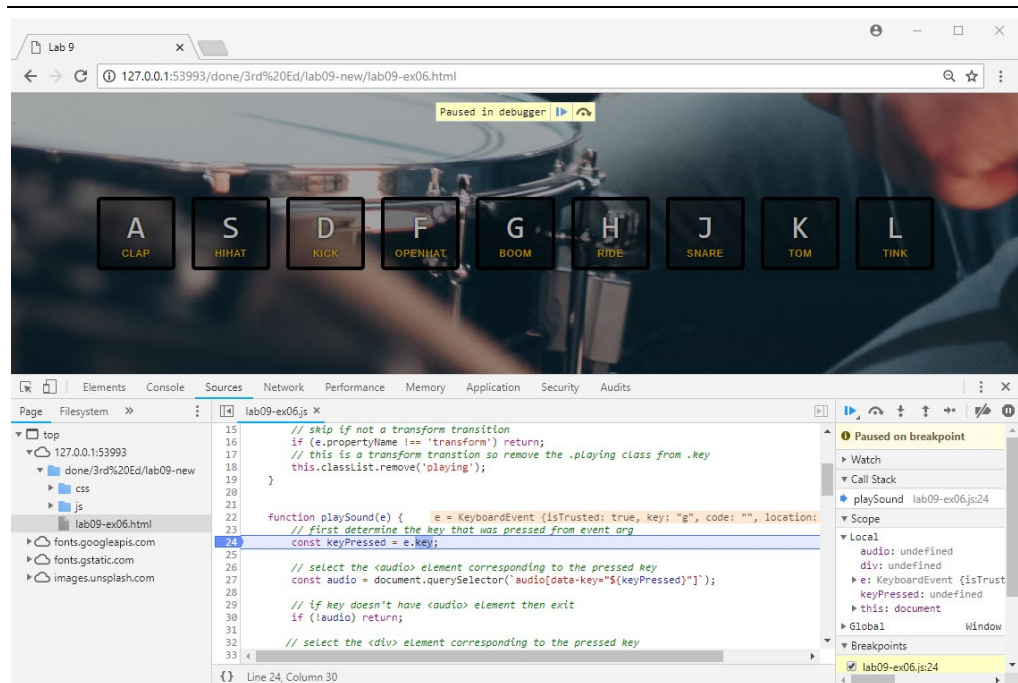

Figure 9.6 – Chrome Debugging

4 Once you have set a breakpoint, you can resume execution of the script or continue execution line-by-line using the execution buttons:

5 Execute the next line of the code by pressing the Step button ⇥• (or pressing F11).

Notice the keyPressed variable in the Scope panel now shows a value.

6 Add another breakpoint on the last line of the function.

7 Press the Resume button ▶ (or press F8).

The intervening lines will be executed and will stop at the new breakpoint. Notice how you can open up and examine the object variables audio and div.

8 Remove breakpoints by clicking on each of the blue breakpoint indicators and refresh the page.

EXERCISE 9.11 — MEDIA EVENTS

1 Examine `lab09-ex11.html` in your browser. You will be adding some event handlers to turn this into a functional video player.

2 Add the following to `lab09-ex11.js` within the `DOMContentLoaded` handler and test.

```
document.addEventListener("DOMContentLoaded", function() {
    const video = document.querySelector('#vidPlayer');
    const playBtn = document.querySelector('#play');

    playBtn.addEventListener('click', playOrPause);
    video.addEventListener('click', playOrPause);

    function playOrPause() {
        if (video.paused) {
            video.play();
        } else {
            video.pause();
        }
    }
}
```

Notice that you are responding not only to the play/pause button clicks, but also to any click in the video itself.

3 To make the play/pause button change depending on its state, add the following and test.

```
document.addEventListener("DOMContentLoaded", function() {

    ...

    video.addEventListener('play', updateButton);
    video.addEventListener('pause', updateButton);

    function updateButton() {
        const icon = video.paused ? symbolPlay : symbolPause;
        playBtn.textContent = icon;
    }
```

4 To make the volume range slider work, add the following and test.

```
const vol = document.querySelector('#volume');
vol.addEventListener('input', changeVolume);

function changeVolume() {
    video.volume = vol.value;
}
```

Notice that volume is a property of the video element.

5   Finally, to make the progress indicator work, add the following and test.

```
const progressBar = document.querySelector('#progressFilled');
video.addEventListener('timeupdate', handleProgress);

function handleProgress() {
    const percent = (video.currentTime / video.duration) * 100;
    progressBar.style.flexBasis = `${percent}%`;
}
```

Notice that the percentage is calculated from two properties of the video element: its duration and its current time position.

EXERCISE 9.12 — FRAME EVENTS

1   Examine `lab09-ex12.html` in your browser and then examine in text editor.

Notice that after the first six images, the `<img>` elements don't have a `src` attribute but instead have a `data-src` element instead. By doing so, the browser will not request the image files. In this exercise, you will programmatically set the `src` attribute when the img elements become visible in the browser.

Instead of using local copies of the images, this example uses versions that are on Cloudinary storage to guarantee some network latency thereby making the fetch visible when testing.

This example based on an online example created by Rahul Nanwani, which is available at https://imagekit.io/blog/lazy-loading-images-complete-guide/.

2   Add the following to `lab09-ex12.js`.

```
document.addEventListener("DOMContentLoaded", function() {
    // lazy class indicates images to be loaded eventually
    const images = document.querySelectorAll("img.lazy");

    // do lazy loading for scroll, resize, and window change events
    document.addEventListener("scroll", lazyload);
    window.addEventListener("resize", lazyload);
    window.addEventListener("change", lazyload);

    // flag used to indicate whether we should ignore event
    let lazyloadThrottleTimeout = false;

    function lazyload () {
      // we will add code to this function in the next step
    }
});
```

3   Now add the following code to the `lazyload()` function.

```
function lazyload () {
  /* unfortunately, scroll events get triggered too
     frequently, so for performance reasons, we throttle its
     speed so it only runs every 20ms */
  if (lazyloadThrottleTimeout) {
     clearTimeout(lazyloadThrottleTimeout);
  }

  lazyloadThrottleTimeout = setTimeout(function() {
     const scrollTop = window.pageYOffset;
     for (let img of images) {
        // is this image visible?
        if (img.offsetTop < (window.innerHeight + scrollTop)) {
           // yes, then sets its src property, which will make
           // browser request image
           img.src = img.dataset.src;
           img.alt = img.dataset.alt;
           // once it's loaded, no need to load again
           img.classList.remove('lazy');
        }
     }
     // if all images are loaded, no need for this handler anymore
     if (images.length == 0) {
        document.removeEventListener("scroll", lazyload);
        window.removeEventListener("resize", lazyload);
        window.removeEventListener("change",
                   lazyload);
     }
  }, 20);
}
```

4   Test in browser. Be sure to reduce the width of the browser window so only two or three images are visible in a row.

As you scroll, you should see the image is blank and then becomes downloaded as it becomes visible. After scrolling to the end, scroll up. Notice that the images don't re-download.

EXERCISE 9.13 — WORKING WITH FORMS

1   Examine `lab09-ex13.html` in your browser.

2   Add the following function to `lab09-ex13.js`:

```
function setBackground(e) {
  if (e.type == "focus") {
      e.target.style.backgroundColor = "#FFE393";
  }
  else if (e.type == "blur") {
      e.target.style.backgroundColor = "white";
  }
}
```

This function is going to get called every time the focus or blur events are triggered in one of our form's input elements.

3   Add the following code:

```
document.addEventListener("DOMContentLoaded", function() {
  const cssSelector = "input[type=text],input[type=password]";
  const fields = document.querySelectorAll(cssSelector);

  for (let field of fields) {
      field.addEventListener("focus", setBackground);
      field.addEventListener("blur", setBackground);
  }
});
```

The anonymous function on the load event assigns the `setBackground()` to the blur and focus events of the relevant `<input>` elements.

4   Test in browser. Move between the different elements with the tab or mouse.

1 You are going to continue working with the file from the previous exercise.

2 Add the following code to `lab09-ex13.js`.

```
for (let field of fields) {
    field.addEventListener("focus", setBackground);
    field.addEventListener("blur", setBackground);
}
// when user submits form, check for empty fields
document.querySelector("#sampleForm").
    addEventListener("submit", checkForEmptyFields);

// ensures form fields are not empty
function checkForEmptyFields(e) {

    // loop thru the input elements looking for empty values
    const  errorList = [];
    for (let field of fields) {
        if (field.value == null || field.value == "") {
            // since a field is empty prevent the form submission
            e.preventDefault();
            errorList.push(field);
        }
    }

    // now display the error message if any
    let msg = "The following fields can't be empty: ";
    if (errorList.length > 0) {
        for (let errorField of errorList) {
            msg += errorField.id + ", ";
        }
        alert(msg);
    }
}
```

3 Test in browser. Experiment by filling in some fields, leaving others blank. Also try putting just spaces in some fields.

4 Modify the `checkforEmptyFields()` function by adding the following:

```
const errorList = [];
for (let field of fields) {
    if (field.value == null || field.value == "") {
        // since a field is empty prevent the form submission
        e.preventDefault();
        errorList.push(field);
    }
}

// hide the error message element in case it is visible
const errorArea = document.getElementById("errors");
errorArea.className = "hidden";

// now display the error message if any
let msg = "The following fields can't be empty: ";
if (errorList.length > 0) {
    for (let errorField of errorList) {
```

```
        msg += errorField.id + ", ";
    }
    // replace the alert; for this simple example, use innerHtml
    errorArea.innerHTML = "<p>" + msg + "</p>";
    errorArea.className = "visible";
}
```

*Instead of displaying the error message inside an alert box this places it within a <div>
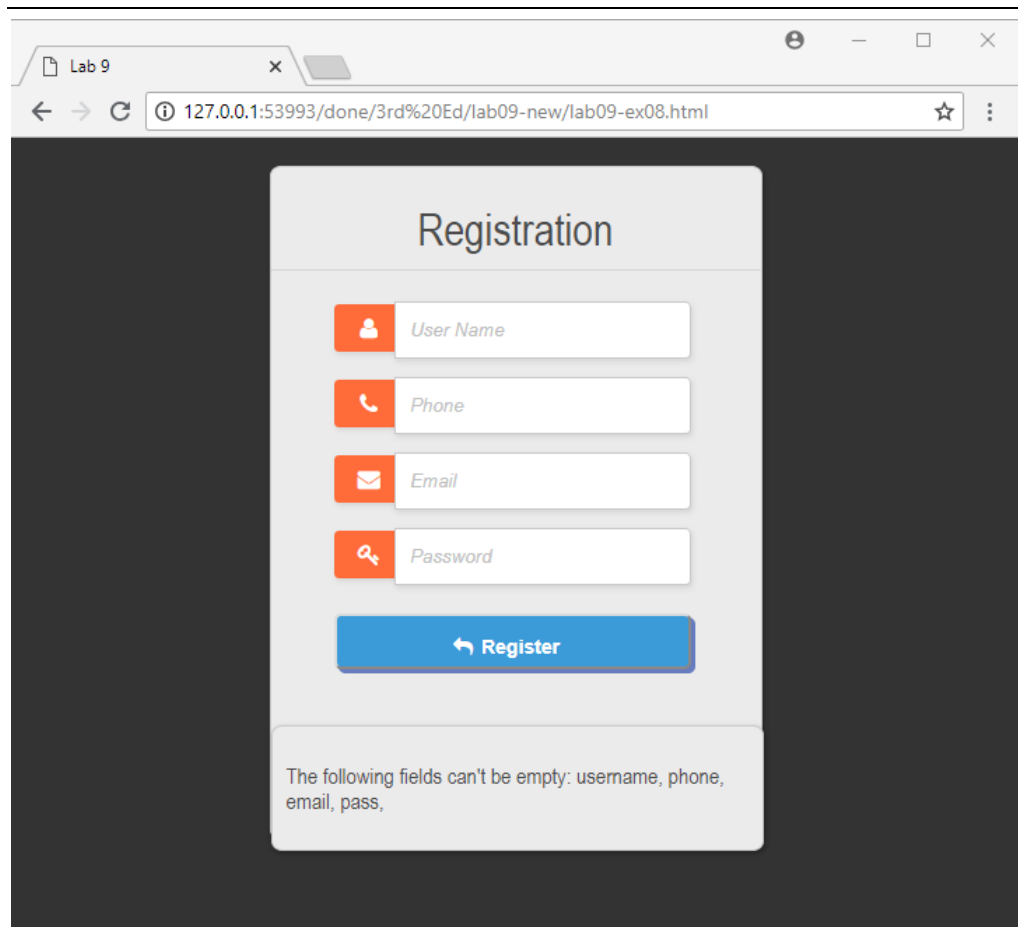element. The result should look similar to that shown in Figure 9.7.*

Figure 9.7 – Finished Exercise 9.14

TEST YOUR KNOWLEDGE #4

Examine `lab09-test04.html`, view in browser, and then open `lab09-test04.js` in your editor. Modify the JavaScript file to implement the following functionality.

1   You are going to need three event handlers. The first will be a click handler for each thumbnail image. In the handler, replace the `src` attribute of the `<img>` element in the `<figure>` so that it displays the clicked thumbnail. Hint: get the `src` attribute of the clicked element and then replace the `small` folder name with `medium` folder name

2   As well, change the `<figcaption>` so that it displays the newly clicked painting's title and artist information. This information is contained within the `alt` and `title` attributes of each thumbnail.

3   Set up an event listener for the input event of each of the range sliders. The code is going to set the `filter` properties on the image in the `<figure>`. Recall from Chapter 7 that if you are setting multiple filters, they have to be included together separated by spaces. **This listener must use event delegation**.

4   Add a listener for the click event of the reset button. This will simply remove the filters from the image by setting the `filter` property to none.

Clicking on thumbnail image replaces the main image in the `<figure>`.

Reset the image and slider values.

Set the appropriate CSS `filter` property whenever the user changes any of the sliders.

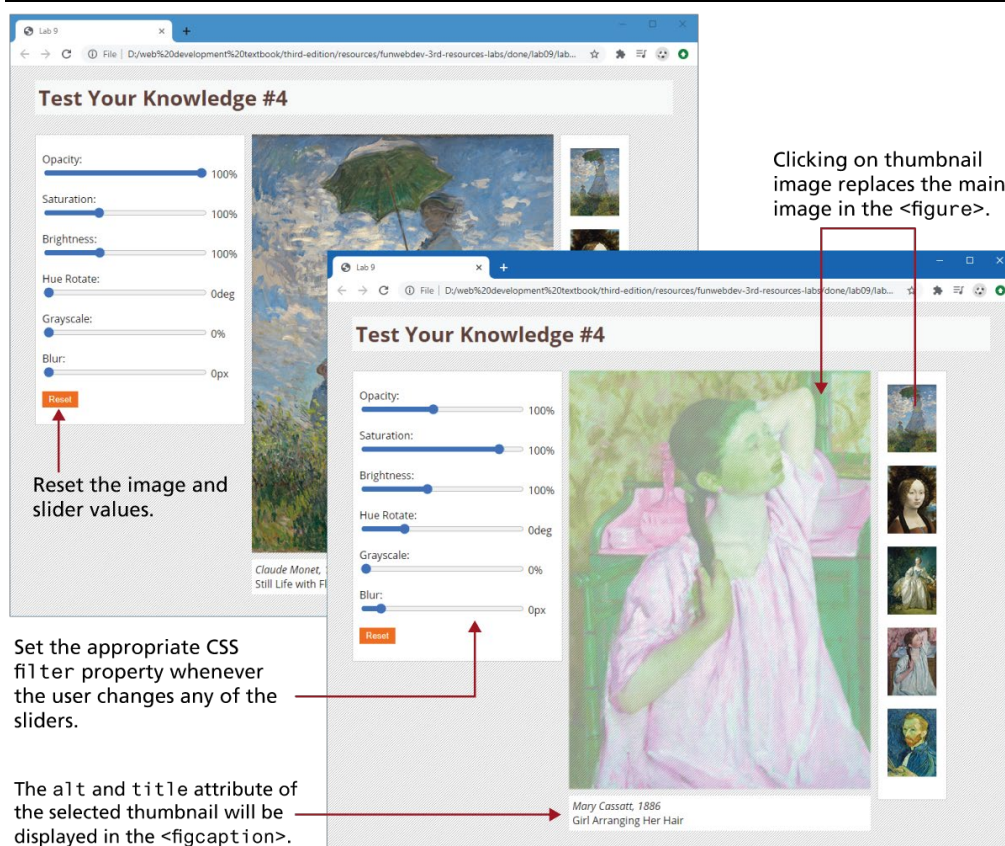The `alt` and `title` attribute of the selected thumbnail will be displayed in the `<figcaption>`.

Figure 9.8 –Test Your Knowledge #4