Exercise 12 – Forms

Objectives

To create some simple validation for HTML forms, checking input against constraints and regular expression tests.

Overview

In this exercise, we will test the values of input controls and alert the user to any problems that occur through visual highlighting. In the second part of the exercise, you will write a regular expression to test if the value is a correctly-constructed UK postcode.

This exercise is scheduled to run for around **25 minutes**.

Exercise Set Up

1. In VSCode, open the file called **Forms.html** from the **Starter** folder inside **Ex12** of the **Exercises** folder.
2. If you have rebooted your computer or stopped *live-server* from running since the last exercise, then use VSCode's integrated terminal to navigate to the **Exercises** folder and run the command **live-server**.
3. Open your browser (if it doesn't automatically) and navigate to **http://localhost:8080** and follow the path to open **Ex12→Starter→ Forms.html**.

Part 1 - Validating Form Inputs

Exercise Instructions

This exercise is going to examine how to set up simple form validation. Before we go any further, it is worth stating we could engage in logic and business rules for testing form data for the remainder of the course! Most form validation libraries are vast.

1. Examine the code provided in Forms.html. Look out for the following important sections:
   a) The radio group **name="cardtype"**;
   b) The hidden **** tags; note they are given an id of **xxxError** that matches an id of a nearby input control.
2. Examine the page in your browser and experiment with the inputs.

We are going to start by selecting a radio button and associating a change event using event listeners. We want to select *all radio buttons in a single radio group*. They all share a common attribute of **name=cardtype**. If we use a CSS3 style selector with the **document.querySelectorAll** function, we can examine the name attribute and grab hold of any that match.

3. Within the **<script>**, declare a variable called **cardType** and set it to be *an array of elements* that are an **input** with the **name** of **cardtype**.
4. Use a *loop* (or **forEach** function) to *attach an event listener* to each of the elements returned into the **cardType** array that:

a) Listens for a **change** event;
b) Has a handling function of **radioValue**;
c) Has bubbling/capturing set to **false**.

5. Define the callback function **radioValue** so that if the *radio button's checked property* is **true**, it logs out the **value** associated with it.

The forEach function belongs to the Array object in JavaScript. It executes a provided function once for each array element. The provided function usually takes an element as an argument and can be written as an arrow function.

For more information see:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

We have used the **addEventListener** approach, associating the change event of each radio button to a function called **radioValue**. Note the use of the **this** keyword. Once we have built the event, this will a*lways refer to the specific radio button object that raises the event*. You can now test the page in the browser –when you click on any of the radio buttons, it will log out the value associated with it.

We will use our skills with DOM manipulation to reveal and suppress error messages when things go wrong. To do this, we need *three* functions: one to check the value, one to show the message and one to remove it. We will start with the display and remove error functions.

6. Write a function called **displayError** that:
   a) Takes **element** as an argument;
   b) Sets a variable called **errorMsg** equal to the *element's id plus the string* **"Error"**;
   c) Checks the status of the **style.display** property of **errorMsg** against the string "none" and if this is true:
      i) Sets the **backgroundColor** of the **element** to **"red"**;
      ii) Sets the **display** property of **errorMsg** to **"block"**.

The function takes a DOM element as a parameter then creates a pointer to the accompanying **** element. As you will have spotted back in step 1, the naming convention is important. For example, there is an **<input id="uname">** and a ****. As long as we follow this pattern, this function will change the style properties of the DOM element with the error and reveal a user prompt to help them understand what is wrong.

7. Write a function called **removeError** that *reverses* what **displayError** does.

With our two functions that manipulate the DOM, we can create the logic that checks the values of the data.

8. Write a function called **checkForEmptyValues** with a body that:
   a) Checks to see if the **value** of the *current element* is an *empty string* **OR null** and *passes the element* to the **displayError** function **if true**;

b) Checks to see if the **value** of the *current element* is *NOT an empty string* and *passes the element* to the **removeError** function **if true**.

When we associate this to the appropriate elements, we can check the event raising object's value and cause an error message to display where appropriate.

We will now hook up the events to the appropriate objects. We want to consider the text based input boxes (there are four of them) and add the event. If we check for any elements with a **type=text**, we will create an array of pointers to the corresponding DOM elements.

9. Under the last code you wrote, declare a variable called **textFields** and *collect elements* with **type** of **text** *in an array* as its value.
10. Loop through **textFields** setting the following properties of each text field in the array:
    a) Background colour to **blue**;
    b) An *event listener* for a **change** that calls **checkForEmptyValues** and bubbling/capturing to **false**;
    c) An *event listener* for a **blur** that calls **checkForEmptyValues** and bubbling/capturing to **false**.

The first line of code that switches the background colour to blue is just to show us when we view the page in the browser the elements we are going to be validating. When we come to release our product, we would remove this. We need to blur and change to get a robust validation. **Change** fires when we blur from the control and the value has changed. **Blur** fires when we leave the control irrespectively.

11. Save the page and test it in the browser. If you tab or click out of a control with a blue background, the error functions should fire.

End of Part 1 - Do not go further!

Part 2 - Using Regular Expressions

In the first part of the exercise, one of the things we created was a control that checked for empty values in the postcode field. In this part, we are going to write a function that will check the postcode against a pre-defined pattern. The function will work similarly to the empty value checker we wrote earlier.

1. Under the code you wrote for Part 1 - Instruction 10, implement a function called **isValidPostcode** that:
    a) Sets the *element's background colour* to **green**;
    b) Declares a variable called **postcodeRegEx** and sets it to the Regular Expression given below:

```
/\b((?:(?:gir)|(?:[a-pr-uwyz])(?:(?:[0-9](?:[a-hjkpstuw]|[0-9])?)|(?:[a-hk-y][0-9](?:[0-9]|[abehmnprv-y])?)))) ?([0-9][abd-hjlnp-uw-z]{2})\b/
```

Top Tip: Try to avoid writing your own regular expressions and find a regular expression that works for you on the web. This one was found at: https://www.regextester.com/93715

c) Uses the regular expression's **test** function against the **value** of the element to set the value of a variable called **isValidPostCode**.
d) Checks to see *if a valid postcode has NOT been returned* and calls the **displayError** function with the *current element as its argument* and then **returns** the value of **isValidPostCode**.
e) Otherwise, it calls **removeError** with *the current element as its argument*.

The function creates a regular expression using the short hand notation format then performs a check on the postcode itself. It then tests the value of the event raising DOM element using the **this** keyword and returns a Boolean based upon its decision. In the event of an error, we display the error message, otherwise we remove it.

2. *After the function*, declare a variable called **postCode** that points to the DOM element with an **id** of **postcode**.
3. Associate **postCode's change** event to the **isValidPostcode** function, with bubbling/capturing set to **false**.

The use of **addEventListener** is very important here. Unlike simple or inline event registration, you are able to associate multiple events to a single DOM object. As a result, when we change the value of postcode, we are firing **isValidPostcode** and **checkForEmptyValues** as input validators. Be careful –your functions do not contradict the logic of one another when using this approach.

4. Test the page in the browser and enter an invalid postcode. You should see an error message occurring. Delete the data and tab away and you should still see an error message displaying.