JavaScript Tutorial

JavaScript Form Validation

Performing client-side validation is not only good practice, but is better for your users. This allows the user to correct any mistakes before submitting a form to your web server. Making it less likely that they will have to reload the form and submit it again.

Lesson Goals

- To access data entered by users in forms.
- To validate text fields and passwords.
- To validate radio buttons.
- To validate checkboxes.
- To validate select menus.
- To validate textareas.
- To write clean, reusable validation functions.
- To catch focus, blur, and change events.

Accessing Form Data

All forms on a web page are stored in the `document.forms[]` array. As we learned, JavaScript arrays having a starting index of 0, therefore the first form on a page is`document.forms[0]`, the second form is `document.forms[1]`, and so on. However, it is usually easier to give the forms names (with the `name` attribute) and refer to them that way. For example, a form named `LoginForm` can be referenced as `document.LoginForm`. The major advantage of naming forms is that the forms can be repositioned on the page without affecting the JavaScript.

Like with other elements, you can also give your forms `id`s and reference them with`document.getElementById()`.

Elements within a form are properties of that form and can be referenced as follows:

Syntax

```
document.formName.elementName
```

Text fields and passwords have a `value` property that holds the text value of the field. The following example shows how JavaScript can access user-entered text:

Code Sample:

FormValidation/Demos/FormFields.html

```


<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Form Fields</title>
<script type="text/javascript">
    function changeBg(){
        var userName = document.forms[0].userName.value;
        var bgColor = document.colorForm.color.value;

        document.bgColor = bgColor;
        alert(userName + ", the background color is " + bgColor + ".");
    }
</script>
</head>
<body>
<h1>Change Background Color</h1>
<form name="colorForm">
    Your Name: <input type="text" name="userName"><br>
    Background Color: <input type="text" name="color"><br>
    <input type="button" value="Change Background" onclick="changeBg();">
</form>
</body>
</html>


```

Some things to notice:

1. When the user clicks on the "Change Background" button, the `changeBg()`function is called.
2. The values entered into the `userName` and `color` fields are stored in variables (`userName` and `bgColor`).
3. This form can be referenced as `forms[0]` or `colorForm`. The `userName` field is referenced as `document.forms[0].userName.value` and the `color` field is referenced as `document.colorForm.color.value`.

Basics of Form Validation

When the user clicks on a *submit* button, an event occurs that can be caught with the`form` tag's `onsubmit` event handler. Unless JavaScript is used to explicitly cancel the submit event, the form will be submitted. The `return false;` statement explicitly cancels the submit event. For example, the following form will never be submitted:

```
<form action="Process.html" onsubmit="return false;">
    <!--Code for form fields would go here-->
    <input type="submit" value="Submit Form">
</form>
```

Of course, when validating a form, we only want the form *not* to submit if something is wrong. The trick is to return `false` if there is an error, but otherwise return `true`. So instead of returning `false`, we call a validation function, which will specify the result to return.

```
<form action="Process.html" onsubmit="return validate(this);">
```

The `this` Object

Notice that we pass the `validate()` function the `this` object. The `this` object refers to the current object - whatever object (or element) the `this` keyword appears in. In the case above, the `this` object refers to the `form` object. So the entire `form` object is passed to the `validate()` function. Note, the function name `validate()` is arbitrary. The function could be called `checkForm()` or anything else. Let's take a look at a simple example.

Code Sample:

FormValidation/Demos/Login.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Login</title>
<script type="text/javascript">
function validate(form){
    var userName = form.Username.value;
    var password = form.Password.value;

    if (userName.length === 0) {
        alert("You must enter a username.");
        return false;
    }

    if (password.length === 0) {
        alert("You must enter a password.");
        return false;
    }

    return true;
}
</script>
</head>
<body>
<h1>Login Form</h1>
<form method="post" action="Process.html"
            onsubmit="return validate(this);">

    Username: <input type="text" name="Username" size="10"><br>
    Password: <input type="password" name="Password" size="10"><br>

    <input type="submit" value="Submit">
    <input type="reset" value="Reset Form">
</form>
</body>
</html>
```

1. When the user submits the form, the `onsubmit` event handler captures the event and calls the `validate()` function, passing in the `form` object.
2. The `validate()` function stores the `form` object in the `form` variable.
3. The values entered into the `Username` and `Password` fields are stored in variables (`userName` and `password`).
4. An `if` condition is used to check if `userName` is an empty string. If it is, an alert pops up explaining the problem and the function returns `false`. The function stops processing and the form does *not* submit.
5. An `if` condition is used to check that `password` is an empty string. If it is, an alert pops up explaining the problem and the function returns `false`. The function stops processing and the form does *not* submit.
6. If neither `if` condition catches a problem, the function returns `true` and the form submits.

Cleaner Validation

There are a few improvements we can make on the last example.

One problem is that the `validate()` function only checks for one problem at a time. That is, if it finds an error, it reports it immediately and does not check for additional errors. Why not just tell the user all the mistakes that need to be corrected, so (s)he doesn't have to keep submitting the form to find each subsequent error?

Another problem is that the code is not written in a way that makes it easily reusable. For example, checking for the length of user-entered values is a common thing to do, so it would be nice to have a ready-made function to handle this.

These improvements are made in the example below.

Code Sample:

FormValidation/Demos/Login2.html

```
<!DOCTYPE HTML>
<html>
```

```
<head>
<meta charset="UTF-8">
<title>Login</title>
<script type="text/javascript">
function validate(form){
    var userName = form.Username.value;
    var password = form.Password.value;
    var errors = [];

    if (!checkLength(userName)) {
        errors.push("You must enter a username.");
    }

    if (!checkLength(password)) {
        errors.push("You must enter a password.");
    }

    if (errors.length > 0) {
        reportErrors(errors);
        return false;
    }

    return true;
}

function checkLength(text, min, max){
    min = min || 1;
    max = max || 10000;

    if (text.length < min || text.length > max) {
        return false;
    }
    return true;
}

function reportErrors(errors){
    var msg = "There were some problems...\n";
    var numError;
    for (var i = 0; i<errors.length; i++) {
        numError = i + 1;
        msg += "\n" + numError + ". " + errors[i];
    }
    alert(msg);
}
</script>
</head>
<body>
<h1>Login Form</h1>
<form method="post" action="Process.html"
                onsubmit="return validate(this);">

    Username: <input type="text" name="Username" size="10"><br>
    Password: <input type="password" name="Password" size="10"><br>

    <input type="submit" value="Submit">
```

```
        <input type="reset" value="Reset Form">
        </p>
    </form>
    </body>
    </html>
```

Some things to notice:

1. Two additional functions are created: `checkLength()` and `reportErrors()`.
    - The `checkLength()` function takes three arguments, the text to examine, the required minimum length, and the required maximum length. If the minimum length and maximum length are not passed, defaults of 1 and 10000 are used.
    - The `reportErrors()` function takes one argument, an array holding the errors. It loops through the errors array creating an error message and then it pops up an alert with this message. The `\n` is an escape character for a newline.
2. In the main `validate()` function, a new array, `errors`, is created to hold any errors that are found.
3. `userName` and `password` are passed to `checkLength()` for validation.
    - If errors are found, they are appended to the `errors` array using the `push()`method. The push method accepts any value or variable and appends it to the array.
4. If there are any errors in `errors` (i.e, if its `length` is greater than zero), then`errors` is passed to `reportErrors()`, which pops up an alert letting the user know where the errors are. The `validate()` function then returns `false` and the form is *not* submitted.

5. If no errors are found, the `validate()` function returns `true` and the form is submitted.
6. Notice the use of the default operator in the `checkLength()` function.

By modularizing the code in this way, it makes it easy to add new validation functions. In the next examples we will move the reusable validation functions into a separate JavaScript file called FormValidation.js.

Validating Radio Buttons

Radio buttons that have the same name are grouped as arrays. Generally, the goal in validating a radio button array is to make sure that the user has checked one of the options. Individual radio buttons have the `checked` property, which is `true` if the button is checked and `false` if it is not. The example below shows a simple function for checking radio button arrays.

Code Sample:

FormValidation/Demos/RadioArrays.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Radio Arrays</title>
<script type="text/javascript">
function validate(form){
    var errors = [];

    if ( !checkRadioArray(form.container) ) {
        errors[errors.length] = "You must choose a cup or cone.";
    }

    if (errors.length > 0) {
        reportErrors(errors);
        return false;
    }

    return true;
}

function checkRadioArray(radioButtons){
    for (var i=0; i < radioButtons.length; i++) {
        if (radioButtons[i].checked) {
            return true;
        }
    }
    return false;
}

function reportErrors(errors){
    var msg = "There were some problems...\n";
    var numError;
    for (var i = 0; i<errors.length; i++) {
        numError = i + 1;
        msg += "\n" + numError + ". " + errors[i];
    }
    alert(msg);
}
</script>
</head>
<body>
<h1>Ice Cream Form</h1>
<form method="post" action="Process.html"
            onsubmit="return validate(this);">
    <strong>Cup or Cone?</strong>
    <input type="radio" name="container" value="cup"> Cup
    <input type="radio" name="container" value="plaincone"> Plain cone
    <input type="radio" name="container" value="sugarcone"> Sugar cone
    <input type="radio" name="container" value="wafflecone"> Waffle cone
    <br><br>
    <input type="submit" value="Place Order">
</form>

</body>
</html>
```

The `checkRadioArray()` function takes a radio button array as an argument, loops through each radio button in the array, and returns `true` as soon as it finds one that is checked. Since it is only possible for one option to be checked, there is no reason to continue looking once a checked button has been found. If none of the buttons are checked, the function returns `false`.

We'll go over this code in more details in the upcoming presentation

---

Validating Checkboxes

Like radio buttons, checkboxes have the `checked` property, which is `true` if the button is checked and `false` if it is not. However, unlike radio buttons, checkboxes are not stored as arrays. The example below shows a simple function for checking to make sure a checkbox is checked.

Code Sample:

FormValidation/Demos/CheckBoxes.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Checkboxes</title>
<script type="text/javascript">
function validate(form){
    var errors = [];

    if ( !checkCheckBox(form.terms) ) {
        errors[errors.length] = "You must agree to the terms.";
    }

    if (errors.length > 0) {
        reportErrors(errors);
        return false;
    }

    return true;
}

function checkCheckBox(cb){
    return cb.checked;
}

function reportErrors(errors){
    var msg = "There were some problems...\n";
    var numError;
```

```
        for (var i = 0; i<errors.length; i++) {
            numError = i + 1;
            msg += "\n" + numError + ". " + errors[i];
        }
        alert(msg);
    }
    </script>
    </head>
    <body>
    <h1>Ice Cream Form</h1>
    <form method="post" action="Process.html" onsubmit="return validate(this);">
        <input type="checkbox" name="terms">
        I understand that I'm really not going to get any ice cream.
        <br><br>
        <input type="submit" value="Place Order">
    </form>

    </body>
    </html>
```

We'll go over this code in more details in the upcoming presentation

---

Validating Select Menus

Select menus contain an array of options. The `selectedIndex` property of a select menu contains the index of the option that is selected. Often the first option of a select menu is something meaningless like "Please choose an option..." The `checkSelect()`function in the example below makes sure that the first option is not selected.

Code Sample:

FormValidation/Demos/SelectMenus.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
```

```
<title>Select Menus</title>
<script type="text/javascript">
function validate(form){
    var errors = [];

    if ( !checkSelect(form.flavor) ) {
        errors[errors.length] = "You must choose a flavor.";
    }

    if (errors.length > 0) {
        reportErrors(errors);
        return false;
    }

    return true;
}

function checkSelect(select){
    return (select.selectedIndex > 0);
}

function reportErrors(errors){
    var msg = "There were some problems...\n";
    var numError;
    for (var i = 0; i<errors.length; i++) {
        numError = i + 1;
        msg += "\n" + numError + ". " + errors[i];
    }
    alert(msg);
}
</script>
</head>
<body>
<h1>Ice Cream Form</h1>
<form method="post" action="Process.html"
            onsubmit="return validate(this);">
    <strong>Flavor:</strong>
    <select name="flavor">
        <option value="0" selected></option>
        <option value="choc">Chocolate</option>
        <option value="straw">Strawberry</option>
        <option value="van">Vanilla</option>
    </select>
    <br><br>
    <input type="submit" value="Place Order">
</form>
</body>
</html>
```

Focus, Blur, and Change Events

Focus, *blur* and *change* events can be used to improve the user experience.

Focus and Blur

Focus and blur events are caught with the `onfocus` and `onblur` event handlers. These events have corresponding `focus()` and `blur()` methods. The example below shows

1. how to set focus on a field.
2. how to capture when a user leaves a field.
3. how to prevent focus on a field.

Code Sample:

FormValidation/Demos/FocusAndBlur.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Focus and Blur</title>
<script src="DateUDFs.js" type="text/javascript"></script>
<script type="text/javascript">
function getMonth(){
    var elemMonthNumber = document.DateForm.MonthNumber;
    var monthNumber = elemMonthNumber.value;

    var elemMonthName = document.DateForm.MonthName;
    var month = monthAsString(elemMonthNumber.value);

    elemMonthName.value = (monthNumber > 0 && monthNumber <=12) ? month : "Bad Numbe
}
</script>
</head>
<body onload="document.DateForm.MonthNumber.focus();">
<h1>Month Check</h1>
```

```
<form name="DateForm">
    Month Number:
    <input type="text" name="MonthNumber" size="2" onblur="getMonth();">
    Month Name:
    <input type="text" name="MonthName" size="10" onfocus="this.blur();">
</form>
</body>
</html>
```

Things to notice:

1. When the document is loaded, the `focus()` method of the text field element is used to set focus on the `MonthNumber` element.
2. When focus leaves the `MonthNumber` field, the `onblur` event handler captures the event and calls the `getMonth()` function.
3. The `onfocus` event handler of the `MonthName` element triggers a call to the `blur()`method of `this` (the `MonthName` element itself) to prevent the user from focusing on the `MonthName` element.

Change

Change events are caught when the value of a text element changes or when the selected index of a select element changes. The example below shows how to capture a change event.

Code Sample:

FormValidation/Demos/Change.html

```
---- C O D E   O M I T T E D ----

<script src="DateUDFs.js" type="text/javascript"></script>
<script type="text/javascript">
    function getMonth(){
        var elemMonthNumber = document.DateForm.MonthNumber;
        var i = elemMonthNumber.selectedIndex;
        var monthNumber = elemMonthNumber[i].value;

        var elemMonthName = document.DateForm.MonthName;
        var month = monthAsString(monthNumber);
        elemMonthName.value = (i === 0) ? "" : month;
    }
</script>
</head>
<body onload="document.DateForm.MonthNumber.focus();">
<h1>Month Check</h1>
<form name="DateForm">
    Month Number:
    <select name="MonthNumber" onchange="getMonth();">
        <option>--Choose--</option>
        <option value="1">1</option>
        <option value="2">2</option>

---- C O D E   O M I T T E D ----

        <option value="11">11</option>
        <option value="12">12</option>
    </select><br>
    Month Name: <input type="text" name="MonthName" size="10"
                    onfocus="this.blur();">
</form>
</body>
</html>
```

This is similar to the last example. The only major difference is that `MonthNumber` is a select menu instead of a text field and that the `getMonth()` function is called when a different option is selected.

Validating Textareas

Textareas can be validated the same way that text fields are by using the `checkLength()`function shown earlier. However, because textareas generally allow for many more characters, it's often difficult for the user to know if he's exceeded the limit. It could be helpful to let the user know if there's a problem as soon as focus leaves the textarea. The example below, which contains a more complete form for ordering ice cream, includes a function that alerts the user if there are too many characters in a textarea.

Code Sample:

FormValidation/Demos/IceCreamForm.html

```
---- C O D E   O M I T T E D ----
function checkLength(text, min, max){
    min = min || 1;
    max = max || 10000;
    if (text.length < min || text.length > max) {
        return false;
    }
    return true;
}

function checkTextArea(textArea, max){
    var numChars, chopped, message;
    if (!checkLength(textArea.value, 0, max)) {
        numChars = textArea.value.length;
        chopped = textArea.value.substr(0, max);
        message = 'You typed ' + numChars + ' characters.\n';
        message += 'The limit is ' + max + '.';
        message += 'Your entry will be shortened to:\n\n' + chopped;
        alert(message);
        textArea.value = chopped;
    }
}

---- C O D E   O M I T T E D ----

    <p>
    <strong>Special Requests:</strong><br>
    <textarea name="requests" cols="40" rows="6" wrap="virtual"
        onblur="checkTextArea(this, 100);"></textarea>
    </p>
---- C O D E   O M I T T E D ----
```