

Validation with JavaScript

By [Nariman K. Melonfire](#)

2003-12-01

Printed from DevShed.com

URL: http://www.devshed.com/Client_Side/JavaScript/Validation_with_JavaScript/

Searching for Answers

So you finally get that site of yours up and running, and sign up about fifty users for your services. You browse through the database entries, only to realize that most of the user information provided is incorrect, with email addresses that read "somewhere#something.com," telephone numbers that read "abgedaf," or birthdates saying that the applicant was born yesterday.

Where did you go wrong? The answer lies between the user submitting his or her registration and the data reaching the database. You neglected the important intermediate step of form validation.

Today, most sites that require user input, be it for a simple mailing list or a more complex online shopping system, use form validation to cut down on instances of erroneous data entering their system. Although there is no concrete way of identifying authentic data, form validation helps identify (to a large extent) the authenticity of the type of data entered, thus cutting down on the amount of "bad" data that gets into the database.

That's where this article comes in. Over the next few pages, I'll be attempting to build a reusable library of functions for client-side form input validation, in an attempt to save you some time the next time an application needs to have its input checked for errors. The end result of this experiment will be a JavaScript object that can be easily included in your scripts, and that exposes basic object methods for data validation. It may not meet *all* your needs; however, the process should be instructive, especially if you're new to object programming in JavaScript, and you'll have very little difficulty customizing it so that it works for you.

Let's get going!

Check Point

In order to illustrate how form validation typically works, consider the following simple example:

```
<html>
<head>
<basefont face="Arial">
<script language="JavaScript">
function validateForm()
{
    // check name
    if (document.forms[0].elements[0].value == "")
    {
        alert ("Please enter a name!");
        return false;
    }

    // check password length
    if (document.forms[0].elements[1].value.length < 6)
    {
        alert ("Please enter a password
                of at least 6 characters!");
        return false;
    }

    // check email address
```

```

        // check age
        if (isNaN(document.forms[0].elements[3].value))
        {
            alert ("Please enter a valid age!");
            return false;
        }

        // check age range
        if (parseInt(document.forms[0].elements[3].value) < 1 ||
parseInt(document.forms[0].elements[3].value) > 99)
        {
            alert ("Please enter a valid age!");
            return false;
        }

        return true;
    }
</script>
</head>

<body>

<form action="someform.cgi" method="post"
    onSubmit="return validateForm();">

Name:
<br>
<input type="text" name="name" id="name">
<p>

Password:
<br>
<input type="password" name="password" id="password">
<p>

Email address:
<br>
<input type="text" name="email" id="email">
<p>

Age:
<br>
<input type="text" name="age" id="age">
<p>

<input type="submit" value="Submit Form">
</form>

</body>
</html>

```

This is a simple form, with fields for the user to enter his or her name, a password, an email address, and an age value. Before the form is submitted, the "onSubmit" event handler calls the validateForm() function, which checks that

- a name is present;

- the length of the password is 6 or more characters;
- the email address is in a valid format;
- the age value is a number, and in the range 1-99.

If any of these tests fail, the `validateForm()` function will print an error message and return `false`, which will prevent the form from being submitted. Only after all the validation tests are successfully passed will the function return `true` and allow the form submission to proceed. In this manner, basic client-side validation can prevent incorrect data from slipping through into your database.

Since this kind of client-side validation is a fairly common task in the Web development world, it's a good idea to encapsulate some of the more common tasks in this context into reusable functions, which can be called as and when needed in your scripts, so as to reduce time and effort spent in this aspect of Web application development. The cleanest way to do this is to build a simple JavaScript form validation object, and add methods to it to validate different types of data. Let's look at that next.

Object Lessons

Before we get into object construction, a quick primer is in order, for all of you who are new to the weird and wonderful world of objects.

In JavaScript, an "object" is simply a set of program statements which perform a specific task; they lay down the basic rules for an object, specifying what it can and cannot do. A typical object constructor contains both variables and functions, and serves as the template from which to spawn specific instances of that object.

Every object constructed from the template has certain characteristics, or "properties," and certain pre-defined functions, or "methods." These properties and methods of the object correspond directly with the variables and functions within the object definition.

Once an object has been defined, JavaScript allows you to spawn as many instances of the object as you like. Each of these instances is a completely independent object, with its own properties and methods, and can thus be manipulated independently of other objects.

Now, you're probably wondering whether this is a little redundant, since JavaScript also allows you to create your own functions and use them wherever required in your code. And you're correct, to some extent; if you're only planning to spawn a single object, a function will work just as well.

But there are situations where you need to spawn more than one instance of an object - for example, multiple menu trees, multiple image swaps, or multiple ticker tapes. In such a situation, objects are preferred, since each instance comes with its own variables and functions, and thus can be manipulated without affecting other variables within the program.

Objects also help you keep your code modular - you can define an object constructor in a separate file, and include that file only in the pages where you plan to use the object - and simplify code changes, since you only need to edit a single file to add new functionality to all your spawned objects.

Rock On

You've probably already used JavaScript objects before. For example, the code...

```
<script language="JavaScript">
a = new Image();
</script>
```

... creates a new instance of the `Image` object, while...

```
<script language="JavaScript">
x = new Date();
</script>
```

... creates a new instance of the Date object.

JavaScript comes with numerous built-in objects, each with pre-defined methods and properties. But what if you want to roll your own?

Well, it isn't very difficult. As a matter of fact, it's almost identical to writing a JavaScript function.

```
<script language="JavaScript">

// object constructor
function Band()
{
    ...
}

</script>
```

You can create an instance of the Band object like this:

```
<script language="JavaScript">

// object instance
obj = new Band();

</script>
```

Note the all-important "new" keyword - this is how JavaScript knows that you're trying to create an instance of an object, rather than merely running a function.

The code above creates an instance named "obj" of the object Band. You can verify that it is indeed an object by popping up an alert().

```
<script language="JavaScript">

obj = new Band();
alert(obj);

</script>
```

Hammer Time

Next, how about modifying the Band() object to accept some arguments.

```
<script language="JavaScript">
```

```
// object constructor
function Band (n, g)
{
    // object properties
    this.name = n;
    this.genre = g;
}

</script>
```

The object Band now has two properties, "name" and "genre." With this, it is possible to create an instance of the object Band, and pass it two parameters ("n" and "g"), which are then stored as object properties. Note my use of the "this" keyword, which provides a convenient way to access variables (and functions) which are "local" to the object.

Just as you can define object properties, it's also possible to define object methods - essentially, simple JavaScript functions. A little more evolution, and the Band object now sports a whoRules() method, which uses the value stored in the object property "name."

```
<script language="JavaScript">

// object constructor
function Band(n, g)
{
    // object properties
    this.name = n;
    this.genre = g;

    // object methods
    this.whoRules = whoRules;
}

// object method whoRules()
function whoRules()
{
    alert (this.name + " rules!");
}

</script>
```

A couple of interesting things here: first, the object method whoRules() is actually defined outside the object constructor block, though it references object properties using the "this" keyword. Second, just as object properties are defined using "this," object methods need to be defined in the same manner. Witness my addition of...

```
function Band(n, g)
{
    ...
    // object methods
    this.whoRules = whoRules;
    ...
}
```

... to the object constructor block. Wanna see how it works? Take a look at the code...

```
<script language="JavaScript">

obj = new Band("SledgeHammer", "Classic Rock");
obj.whoRules();

</script>
```

... and the output.

SledgeHammer rules!

How Things Work

So that's the theory. Let's now spend a few minutes discussing the rationale behind the formValidator object I plan to build.

Stripped down to its bare bones, my formValidator object consists of two components:

1. A series of methods that accept the data to be validated as method arguments, test this data to see whether or not it is valid (however "valid" may be defined within the scope of the method), and return an appropriate signal.
2. A JavaScript structure (here, an array) that holds a list of all the errors encountered during the validation process, and a series of methods to manipulate this structure.

As you will see, these two basic components make it possible to build a very simple (and yet very useful) formValidator object, one that exposes a number of generic methods.

Now, before proceeding further, I need to decide how this class is going to work. Here's how I plan to use it:

```
// instantiate object
fv = new formValidator();

// perform checks
// check for empty field
if (fv.isEmpty(document.forms[0].elements[0].value))
{
    fv.raiseError("Please enter a value");
}

// check for field range
if (!fv.isWithinRange(document.forms[0].elements[1].value, 1, 99))
{
    fv.raiseError("Please enter a value in the range 1-99");
}

// if errors, display, else proceed
if (fv.numErrors() > 0)
{
    fv.displayErrors();
}
```

As you can see, once the object is instantiated, various object methods are called (with the appropriate form input value as parameter) to test whether the input is "good." If it isn't, an error is raised and stored in the object's internal error stack; these errors can be displayed at a later time, once all the validation is complete.

Once the basic functionality of the object is clear, it's a good idea to spend some time listing the important methods, together with their purpose. Here's my initial cut:

- **isEmpty()** - check whether the specified form variable is empty;
- **isNumber()** - check whether the specified form variable is a number;
- **isAlphabetic()** - check whether the specified form variable contains alphabetic data;
- **isAlphaNumeric()** - check whether the specified form variable contains alphanumeric data;
- **isWithinRange()** - check whether the specified form variable contains a value within the specified numeric range;
- **isEmailAddress()** - check whether the specified form variable contains a valid email address;
- **raiseError()** - add an error message to the error stack;
- **displayErrors()** - display the list of error messages as alert boxes;
- **numErrors()** - return the number of error messages generated so far.

These are the essential methods; there may be more, which I will add as development progresses.

A Little Space

Let's begin by setting up the class definition:

```
// create object
function formValidator()
{
    // snip
}
```

Now, since I plan to build some basic error-handling routines into this class, I need to add a variable to hold this information.

```
// create object
function formValidator()
{
    // set up array to hold error messages
    this.errorList = new Array;

    // snip
}
```

Let's now begin constructing the validation routines themselves.

Here's the first one, which performs a very basic test; it checks to see whether the corresponding variable contains a value or not.

```
// check to see if input is whitespace only or empty
function isEmpty(val)
{
    if (val.match(/^\s+$/)) || val == "")
    {
        return true;
    }
}
```

```

    }
    else
    {
        return false;
    }
}

```

This is fairly simple, but worth examining in detail, since all subsequent methods will follow a similar structure.

The `isEmpty()` method is called with a single argument: the value to be tested. It then uses the `match()` method of the JavaScript String object to check this value against a regular expression. In this specific example, the regular expression is designed to match a string consisting of spaces only; such a string can reasonably be considered "empty" of any actual data. If such a string is found, or if the value is found to contain nothing, the function returns true; if not (that is, if the value actually contains some data), the function returns false.

As you will see, this type of structure for the object method makes it simple to wrap the method call in a conditional test in your form processing script.

Let's move on to another interesting function:

```

// check to see if input is number
function isNumber(val)
{
    if (isNaN(val))
    {
        return false;
    }
    else
    {
        return true;
    }
}

```

In this case, the JavaScript `isNaN()` function is used to check whether the value supplied to it is a number or not.

Another useful method, especially when dealing with numbers, is the `isWithinRange()` method, which provides an easy way to check whether the input is within a certain numeric range.

```

// check to see if value is within min and max
function isWithinRange(val, min, max)
{
    if (val >= min && val <= max)
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

Finally, you can include a small method to verify if a particular checkbox is checked or not, via the `isChecked()` method:

```
// check to see if form value is checked
function isChecked(obj)
{
    if (obj.checked)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Note the difference between this method, and the ones that preceded it. Previous object methods have been written to accept a form value as argument; this one accepts an object, representing the checkbox element, as argument.

Expressing Yourself

It's also possible to create more complex validation routines using JavaScript's built-in support for regular expressions. Consider the next method, `isAlphabetic()`, which uses a regular expression to test whether all the characters in the input string are alphabets...

```
// check to see if input is alphabetic
function isAlphabetic(val)
{
    if (val.match(/^[a-zA-Z]+$/))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

... or whether the input string contains a combination of letters and numbers.

```
// check to see if input is alphanumeric
function isAlphaNumeric(val)
{
    if (val.match(/^[a-zA-Z0-9]+$/))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

This ability to use regular expressions to perform data validation comes in particularly handy when checking user-supplied email addresses for validity, as the very cool (and very useful) method `isEmailAddress()` demonstrates:

```
// check to see if input is a valid email address
function isEmailAddress(val)
{
    if (val.match(/^([a-zA-Z0-9]+
        ([\a-zA-Z0-9_-])*)
        @([a-zA-Z0-9_-])+(\a-zA-Z0-9_
        -]+)+/))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Under Construction

So that takes care of the basic validation routines. If these aren't enough for you, it's fairly easy to add new ones to the class, as per your specific requirements; just follow the template I've used above. All that's left now is to add some methods to take care of error messages in case a value fails its validation test.

The first of these is a simple little method named `raiseError()`, which allows you to add a human-readable message to the error stack. If you look at the very first schematic of how I designed this object to work, you'll see that this method is typically called if a validation test returns false. The method accepts an error message, which gets added to the error stack; this error stack itself is implemented as a regular JavaScript array. Look at the object constructor described previously to see its initialization.

```
// add an error to error list
function raiseError(msg)
{
    this.errorList[this.errorList.length] = msg;
}
```

The `numErrors()` method returns the size of the "errorList" array; if the size of the array is greater than 1, it implies that one or more errors have occurred while validating the form data. Take a look:

```
// return number of errors in error array
function numErrors()
{
    return this.errorList.length;
}
```

Now, `numErrors()` will only tell you how many errors have occurred; it won't show you the actual error messages. For that, you need the `displayErrors()` method, which returns the contents of the errorList array through a series of `alert()` boxes.

```
// display all errors
```

```
// iterate through error array and print each item
function displayErrors()
{
    for (x=0; x
```

With the validation functions all created, the final step is to add them to the object constructor so that they can be accessed as methods of the formValidator object. Here's the revised object constructor...

```
// create object
function formValidator()
{
    // set up array to hold error messages
    this.errorList = new Array;

    // set up object methods
    this.isEmpty = isEmpty;
    this.isNumber = isNumber;
    this.isAlphabetic = isAlphabetic;
    this.isAlphaNumeric = isAlphaNumeric;
    this.isWithinRange = isWithinRange;
    this.isEmailAddress = isEmailAddress;
    this.isChecked = isChecked;

    this.raiseError = raiseError;
    this.numErrors = numErrors;
    this.displayErrors = displayErrors;
}
```

... and here's what the final object code looks like:

```
// create object
function formValidator()
{
    // set up array to hold error messages
    this.errorList = new Array;

    // set up object methods
    this.isEmpty = isEmpty;
    this.isNumber = isNumber;
    this.isAlphabetic = isAlphabetic;
    this.isAlphaNumeric = isAlphaNumeric;
    this.isWithinRange = isWithinRange;
    this.isEmailAddress = isEmailAddress;
    this.isChecked = isChecked;

    this.raiseError = raiseError;
    this.numErrors = numErrors;
```

```

        this.displayErrors = displayErrors;
    }

    // check to see if input is whitespace only or empty
    function isEmpty(val)
    {
        if (val.match(/^\s+$/)) || val == "")
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    // check to see if input is number
    function isNumber(val)
    {
        if (isNaN(val))
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    // check to see if input is alphabetic
    function isAlphabetic(val)
    {
        if (val.match(/^[a-zA-Z]+$/))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    // check to see if input is alphanumeric
    function isAlphaNumeric(val)
    {
        if (val.match(/^[a-zA-Z0-9]+$/))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    // check to see if value is within range
    function isWithinRange(val, min, max)

```

```

{
    if (val >= min && val <= max)
    {
        return true;
    }
    else
    {
        return false;
    }
}

// check to see if input is a valid email address
function isEmailAddress(val)
{
    if (val.match(/^([a-zA-Z0-9])+
        ([\a-zA-Z0-9_-])*@
        ([a-zA-Z0-9_-])+
        (\.[a-zA-Z0-9_-]+)+/))
    {
        return true;
    }
    else
    {
        return false;
    }
}

// check to see if form value is checked
function isChecked(obj)
{
    if (obj.checked)
    {
        return true;
    }
    else
    {
        return false;
    }
}

// display all errors
// iterate through error array and print each item
function displayErrors()
{
    for (x=0; x<this.errorList.length; x++)
    {
        alert("Error: " + this.errorList[x]);
    }
}

// add an error to error list
function raiseError(msg)
{
    this.errorList[this.errorList.length] = msg;
}

```

```
// return number of errors in error array
function numErrors()
{
    return this.errorList.length;
}

// end object
```

A Quick Snack

At this stage, I think I have enough building blocks to actually begin using this class to perform form input validation. Let's build a simple form, and also add the JavaScript to validate the data entered into it. Note that the form validation object created on the previous page is stored in the file "formValidator.js," which is read in near the top of the page.

```
<html>
<head>
<basefont face="Arial">
<script language="JavaScript" src="formValidator.js">
</script>
<script language="JavaScript">

// check form values
function checkForm()
{

    // instantiate object
    fv = new formValidator();

    // perform checks
    // check for empty name field
    if (fv.isEmpty(document.forms[0].elements[0].value))
    {
        fv.raiseError("Please enter a name");
    }

    // check for empty age field
    if (fv.isEmpty(document.forms[0].elements[1].value))
    {
        fv.raiseError("Please enter an age");
    }

    // check for valid age range
    if (!fv.isWithinRange(document.forms[0].elements[1].value,
        1, 99))
    {
        fv.raiseError("Please enter an age in the
        range 1-99");
    }

    // check for empty email address
    if (fv.isEmpty(document.forms[0].elements[5].value))
    {
```

```

        fv.raiseError("Please enter an email address");
    }

    // check for valid email address format
    if (!fv.isEmpty(document.forms[0].elements[5].value) &&
        !fv.isEmailAddress(document.forms[0].elements[5].value))
    {
        fv.raiseError("Please enter a valid email address");
    }

    // check for checkbox
    if (!fv.isChecked(document.forms[0].elements[6]))
    {
        fv.raiseError("Please indicate your agreement with
                        this site's terms and conditions");
    }

    // all done

    // if errors, display, else proceed
    if (fv.numErrors() > 0)
    {
        fv.displayErrors();
        return false;
    }
    else
    {
        return true;
    }
}

```

</script>

</head>

<body>

<form action="action.cgi" method="POST"
 onSubmit="return checkForm()">

Name:

<input type="text" name="name" size="15">

<p>

Age:

<input type="text" name="age" size="2" maxlength="2">

<p>

Sex:

<input type="Radio" name="sex" value="m" checked>Male

<input type="Radio" name="sex" value="f">Female

```

<p>
<b>Favourite sandwich type:
<br>
<select name="stype">
<option value="1">Thin crust
<option value="2">Thick crust
<option value="3">Toasted
</select>

<p>

<b>Email address:
<br>
<input type="text" name="email" size="25">

<p>

<input type="Checkbox" name="agree">
I agree to the terms and conditions of this site

<p>

<input type="Submit" name="submit" value="Save">
</form>

</body>
</html>

```

In this case, when the form is submitted, the "onSubmit" event handler invokes the checkForm() function, which instantiates a new object of the formValidator class. Once this object has been instantiated, all the validation routines defined earlier become available to the script as object methods, and can be used to test the various input values provided in the form.

Let's see if it works. Here's what happens when I try submitting the form with no fields filled in:

```

* Error: Please enter a name
* Error: Please enter an age
* Error: Please enter an age in the range 1-99
* Error: Please enter an email address
* Error: Please indicate your agreement with this
site's terms and conditions

```

And here's what happens when I submit the form with incorrect data in the age field:

```

* Error: Please enter an age in the range 1-99

```

Only when all the errors have been corrected, and the user's input passes all the validation tests, will the function checkForm() return true and the form get submitted to the form processing script. In this way, input validation can prevent erroneous data from corrupting your database.

Of course, you can perform input validation without using an object (as demonstrated in the very first example in this article).

However, developers typically perform pretty standard tasks when validating form input; instead of writing (and rewriting) the same code over and over for each of your projects, it makes more sense to create a simple, reusable library of functions, encapsulate them in an object, and plug that object into your forms as and when needed.

It's important to note that the object above is still fairly primitive. If you plan to use it in a live environment, you should modify it to include more stringent error checking. (I've omitted this in the example above to make the code a little easier to read and explain.) You might also want to add more validation methods to it as and when you need them, so to build a library that you can reuse over and over again, for different applications and requirements.

And that's about all for the moment. In this article, you expanded your knowledge of JavaScript's OOP capabilities by actually using all that theory to build something useful - a form validation widget which can be used to verify the data entered into an online form, and provide a way for you to take action in case the data entered by the user does not meet your expectations.

If you're a stressed-out Web developer working on a Web site, a Web application or an embedded system, you might find this object a handy tool in your next development effort. If you're a novice programmer struggling to understand how OOP can make your life easier, I hope this article offered some pointers, as well as some illustration of how object-oriented programming works. And if you don't fit into either of those categories - well, I hope you found it interesting and informative, anyway.

See you soon!

Note: All examples in this article have been tested in Microsoft Internet Explorer 6. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article.