SIENA*college* Computer Science

Computer Science 180
Web Design
Siena College
Fall 2011

Topic Notes: JavaScript Form Validation

Our next group of examples involve doing error checking on form elements. For example we may wish to require that certain fields have values (like a name or address), that certain fields use only a limited set of characters (like a ZIP code or phone number), or that `<select>` menus or radio button groups have a valid option selected. This process is known as *validation*.

We can accomplish this in many ways – we will consider some techniques that use JavaScript.

Required Fields

We start simple: a form that requires one to enter a first and last name.

See Example: entername.html

The form itself is nothing new – we have two text inputs and a button that calls a JavaScript function to process it.

The difference here is that we check the contents of the text inputs before using them and hiding the form. If we find an error, an alert is displayed, then the function returns before proceeding to the end, where the result message is posted and the form is hidden.

A second option would be to replace the alert popups with error messages on the form.

See Example: entername2.html

Here, when an empty text input is found, an error message is displayed next to that field (in a third column in the table that was not evident from the initial display). We also have a local variable `errorFound` that remembers if any error has been encountered. It starts as `false` (after all, we haven't found any errors before looking). Then, if any empty text input is found, the variable is changed to `true`.

Then after all fields have been checked, if an error was found, we return before removing the form and displaying the result.

This approach has the advantage of avoiding what might be an annoying popup alert, and it informs the form's user of all errors at once, with messages right next to the place where the error occurred.

Let's do a minor improvement on this, informing the user of how many errors have been detected.

See Example: entername3.html

Here, rather than a variable that holds `true` or `false`, we have a number that counts the errors as they occur. Then if any errors were found, a new `<div>` is given a message.

Another possibility is to disable the form submission button until all elements have been given valid values.

See Example: entername4.html

In this case, the `pressed` function becomes much simpler, because it can only be called if the button is not disabled, which is the case only when all form elements are valid.

The validation work ends up being done in the `fieldChanged` function, which gets called every time someone presses a key inside one of the text inputs. This is accomplished by setting the `onkeypress` attribute.

The function looks at the `value` attribute (the contents) of each text input (not just the one being modified – why?). If it finds one that is empty, it disables the button and returns. If the function proceeds beyond all of those checks, it must be the case that no errors were found, so the button is enabled.

Numeric Fields and Regular Expressions

Next, we consider the validation of a numeric field. There is no form control that allows only numbers to be entered, so we need to do some work to check for this.

We will consider a special case of this – a 5-digit ZIP code. A ZIP code consists only of numbers, of which there must be 5.

See Example: zipcode.html

We can enforce the upper limit of 5 digits by creating the text input with a `maxlength` of 5. This way, no one can possibly type more than 5 characters. But they can type fewer than 5, and they can type things other than numbers.

The work here is in the `pressed` function, which reads the `value` from the field and makes sure it's a valid ZIP code. If it fails any of the tests, an appropriate error message is displayed. If all tests are passed, it displays the result.

The first check is to see if the ZIP code is missing completely. This is the same as we saw in the "enter your name" examples for a required field.

If something is in the field, we now need to make sure it's a valid ZIP code. First, we check that its length is 5. We can do this by checking the `.length` property of the value we retrieved from the text input. If it's not, we display an appropriate error.

The next check is to see if the 5-character value is a valid number. JavaScript provides some built-in functions that help here. The first is `parseInt`. `parseInt` takes a string as its parameter (such as the one we retrieved from the text input) and returns a numeric representation. Often, we'd use this to obtain a number on which we might be able to do math, for example. But here, we just want to see if it's possible to convert the value to a number. If not, it returns a special value `NaN`, which stands for "not a number". We can then see if that is the value that was returned using JavaScript's builtin `isNaN` function. If `isNaN` returns `true` when passed the number returned by `parseInt`, then the text input didn't contain a valid number. So, we display an appropriate message. Whew.

But we're not done. The string `-1234` is 5 characters long and is a valid number, but not a valid

ZIP code. So we then check to see if the number returned by `parseInt` is less than 0. If so, we display an error message.

Unfortunately, this does not work – it accepts any 5-character value that *starts with* a number. This is because the `parseInt` function will look for any number at the start of the string it is passed, and will ignore any additional characters. So...not good enough.

Instead, we need to examine the characters individually and make sure that each is numeric. While this is more complex, it does mean we can replace two of our tests ("must be numeric!" and "cannot be negative!") with a single test.

See Example: zipcodefixed.html

The key here is to use a *regular expression* to determine whether the string from the field contains only the digits 0 through 9.

Essentially what we are doing is pattern matching. In this case, we define a pattern using a special notation, then call a function `test` to see if that pattern matches our given input.

The patterns are defined in a special regular expression notation. This notation is not specific to JavaScript – you can find it in other programming languages and other contexts.

Our pattern is:

```
/[0-9]{5}/g
```

What is that supposed to be? Let's break it down:

- The two `/` characters are the delimiters of the regular expression. Everything in between them defines the pattern.
- `[0-9]` defines a *range* of matching characters – in this case all characters from 0 through 9 – the numbers. If we wanted all lowercase letters, we could use the range `[a-z]`, while `[A-Z]` would match all uppercase letters. `[0-9a-zA-Z]` would match all numbers and both uppercase and lowercase letters. `[aeiou]` would match only vowels. It is a flexible construct.
- `{5}` defines how many copies of that character are required in the pattern. Here, we want 5 digits in a row. Anything else would not match the pattern.
- The `g` at the end indicates that this is to be a *global* match – for now, we'll include it on all of our patterns.

So this is a pattern consisting of exactly 5 characters, all of which are in the range from 0 to 9.

The pattern match occurs with the expression:

```
pattern.test(zipcode)
```

`test` is a method (like a function) that can be applied to a regular expression. We send it a string value and it returns `true` if the string matches the regular expression, `false` otherwise.

Since we want to trigger our error in the case when the method returns `false` and we only would execute the statements inside the the `if` block when the condition is `true`, we need to take the opposite, which can be done with the `!` in front of the method call.

What would the `test` method return for each of these inputs and why?

- `pattern.test("39xe2")`
- `pattern.test("392")`
- `pattern.test("-2344")`
- `pattern.test("10.02")`
- `pattern.test("12010")`

There is a comment in the `pressed` function in the example that shows an alternate regular expression that would work here:

```
var pattern=/[0-9][0-9][0-9][0-9][0-9]/g;
```

Here, we require that there be a character in the 0-9 range followed by another and another and another and another.

We will see more regular expressions as we validate some other types of input.

You may be wondering why we still need the check for the length of the ZIP code being 5. Won't the pattern match take care of it? The answer is yes, it would, and we could remove that check and still accept only valid ZIP codes. It remains in the example only because it gives a slightly more specific error message in the case when the length does not match.

For a slightly more interesting case, consider a field that requires a ZIP+4, which consists of a regular 5-digit ZIP code followed by 4 more digits.

See Example: zipplus4.html

We can accomplish this easily with a more complex regular expression:

```
/[0-9]{5}-[0-9]{4}/g
```

Here, the requirement for a pattern match is that there are 5 characters in the 0-9 range, followed by a `-`, followed by 4 more characters in the 0-9 range.

Other changes in this document from the previous are just to change some labelling and to allow 10 characters of input in our text field.

Select Drop-Down Menus

When using a `<select>` we know that the `<option>` selected must be one of the `<option>`s in the list, so there is usually not much work to do. An exception is when there is no “preselected” valid default option. You have certainly seen some forms where the drop down starts up showing the option “Please select”.

This has the advantage that someone will need to make a conscious decision about their selection rather than taking a default value that may not be correct. But...we do not want to allow submission of the form unless one of the real options has been chosen.

See Example: favoritecolor.html

Here, we simply check for the `value` of the “Please select...” option and display the error message and return if it is the current selection.

Radio Buttons

We have already seen what we need to validate a set of radio buttons – we simply need to ensure that some selection has been made.

See Example: radiovalidate.html

The approach is to set the variable `year` to the value `"none"` initially, then check each radio button (as we did before), updating `year` if we find one of the inputs to be checked. If after all radio buttons have been tested and the `year` variable remains `"none"`, we display an error and return.

Matching Fields

You have probably encountered web forms where some information must be entered twice, often an email address or a password. You are not allowed to continue until you have entered the same value in both fields.

See Example: passwordmatch.html

This is a straightforward thing to check. We retrieve both password values from the form. If they are not equal, we set the error message as in previous examples.

Password Requirements

While discussing passwords, we should consider some other typical requirements for passwords. These may include minimum length requirements or requirements that different combinations of letters, numbers, and punctuation be used in a password.

See Example: passwordset.html

Here, we require that the same password be entered twice, as before, but also require that the password has at least 6 characters including at least one letter and at least one number.

Checking the length is done by making sure that the `.length` property of the `passwd` string is at least 6.

Checking that the required combination of at least one letter and one number has been satisfied requires a bit more work. To help with this, two regular expressions are defined: `/[a-zA-Z]/g` represents all of the alphabetic characters and `/[0-9]/g` represents all of the numeric characters. Each `test` method call with these checks that the `passwd` contains, first, a letter, and second, a number.

Required Checkboxes

Another common requirement on forms is that a checkbox be selected to agree to some terms of use, or something similar.

See Example: agree.html

Here, we check to make sure that the checkbox is `checked` – if not, we display an error. If so, we continue.

General Numeric Input

We considered a special case of a numeric field – the ZIP code. But what about other numeric fields? Typical restrictions on numeric fields might include that the number has to be within a given range ("I am thinking of a number between 1 and 100") or cannot be negative, or cannot include a decimal point.

In this example, we are requiring an "invoice number" that we know should be numbered between 0 and 999,999.

See Example: invoicenumber.html

The text input is created with a `maxlength` of 6, which will prevent entry of any numbers that are too large. So in the JavaScript validation function, we need to check the following:

- The field cannot be empty – this is accomplished in the same way as in previous examples.
- That the entry is a valid integer. To do this, we call `parseInt(invoice)` to convert the input into an integer, if possible. But since this could return a valid number for inputs we wish to disallow (like "45b3se" which would be converted to "45" by `parseInt`), we compare the value it returns to our original input. If they match, it means the entire string was successfully converted to an integer. Otherwise, all or part of the string was not numeric, so an error is reported.
- Finally, we need to make sure the number is not negative, as `parseInt` is perfectly happy to process negative numbers.

Special Fields

We have already seen one type of "special" field: the ZIP code must be numeric and must contain exactly 5 digits. As a general rule, it is good to do the most precise checking of input fields like this as we can, and to provide the most specific error messages we can.

Telephone Numbers

A common example is the processing of a telephone number. We will assume we are using numbers in the multi-country "North American Numbering Plan" where a telephone number consists of a 3-digit area code, a 3-digit exchange, and a 4-digit subscriber number. There are additional restrictions, we we will ignore them for the moment.

Any of the following are perfectly reasonable representations of my office phone number:

```
5187834171
518-783-4171
518 783 4171
(518)783-4171
(518) 783-4171
518.783.4171
```

And this is before we consider the fact that most numbers on a telephone keypad also correspond to letters of the alphabet!

We have all been annoyed by poorly-designed web forms (laziness, I call it) where you enter a phone number and are told "you must enter your phone number in the following format: (xxx) xxx-xxxx" and if you don't get the format perfect, an error will be generated. Worse yet, you are not told what format to use until you have already used a different (still perfectly reasonable) one and have been presented with an error message.

There are two reasonable approaches to eliminate this source of annoyance and frustration among those who will be filling out our form:

1. Accept as many reasonable formats as possible using a "free form" input field, or
2. Present the fields in a way such that the user can only fill out the fields in the format you are expecting.

In either case, the input should be converted to a canonical format convenient for later storage or display. This might mean that each phone number is converted into three separate numeric strings: one for the area code, one for the exchange, and one for the subscriber number. Or, the number could be converted to a single string of 10 digits with no spaces of punctuation.

The following example illustrates both input techniques, and converts to the second canonical format described (one 10-character string).

See Example: phonenumbers.html

In the form itself, we see a single text input with a `maxlength` of 15 (to allow for some spaces, dashes, etc., in addition to the 10 required digits) for the home phone number, and three separate text inputs, with appropriate `maxlength` values (3, 3, 4), and parentheses around the area code field and a dash between the exchange and the subscriber number to help indicate the expected format.

See the comments in the example's JavaScript event handler function for details of how these fields are validated, converted into the canonical format, and displayed in a nice format after the form is accepted.