Scripting for form validation

Objectives

Upon completion of this session you should be able to:

- define which fields in a fill-in form require validation
- define appropriate rules for the validation of a field
- modify short scripts to validate the content of fields

Background Tasks JavaScript
PHP Summary Quiz

Background

Whilst many of the scripting processes we have looked in the practical sessions so far are interesting (and hopefully instructive), there is little doubt that working with forms is by far the most *useful* application of client-side scripting. In particular, we can use scripts to **validate** the contents of a form before they are sent to be processed. This allows us to ensure that forms have been completed properly, although we cannot know if the information that has been entered is 'correct' (is your name *really* Donald Duck?). "Completed properly" can mean any or all of the following:

- Some elements *must* have a value or selection, so they cannot be '*empty*'
- Some elements should have a *certain* (maximum or minimum) length
- Some elements should have a certain *range* (maximum and minimum values)
- Some elements should only have certain *contents* (nothing but numbers, for example)
- Some elements should have a specific *structure* (such as email addresses, dates and phone numbers)

The aim of **validation** is to check these kinds of requirements. To do this we need to

- identify which elements in the form actually require validation (not all will)
- define the rules for the elements to be validated: 'not empty', 'something selected', 'at least five digits', 'only numbers less than 100', and so on
- define the rules for elements with a specific structure; for example, the rules for a valid email [where x@z.au is the shortest possible legitimate address] might include all or some of these:
  - there must be at least six characters
  - the first character must be alphanumeric: letters or specific punctuation characters, but not digits
  - there must be one - and only one - @ symbol
  - the @ cannot be the first character, or one of the last three
  - there must be at least one period (.) character between the @ symbol and the end
  - there must be at least one character between the @ and the next period
  - the third character from the end must be a period
  - the last two characters must be letters
  - the last two characters must correspond to a legitimate country domain name
- construct a script (one or more functions) that incorporates these rules
- link the script to the form (conventionally using the form's *Submit* event) in such a way that failing any of the validation tests stops the form being sent
- include appropriate messages to the user - usually using *alert* message boxes - telling them what is 'wrong' with what they have filled in, so that they can correct it and re-submit the form

We can approach this process in two ways. We can create a script that processes a specific form, using the specific fields and names. Of course, this means each new form requires that we create a new script (although we might do a great deal of informal 'code reuse'). The function returns a value of 'false' if there are any problems with the form, and 'true' if there are no problems. Appropriate messages are generated by the script to tell the user what is wrong. If the function returns true, the form is immediately sent using the defined *method* and *action* properties.

Here is an example such a 'one-off' validation. In the head we place the script:

```
<script language="JavaScript"> <!--
function validate(testform){
//
// Create a basic array containing the numbers that we can use to check if a field
contains anything but numbers
//
var digits="0123456789"
var temp
//
// Check if the Name field is empty - that is, its value is null
//
if (document.testform.Name.value=="") {
//
// If field string is null, display a (very bad!!) message, and return false (stop
the script and exit here)
//
alert("No Name !")
return false
}
//
// Name field is okay, so check if the age field is too short, displaying
// appropriate messages; again, return false if it is
//
if (document.testform.age.value.length < 1) {
alert("You have not told us your age!")
return false
}
if (document.testform.age.value.length <2 ) {
alert("Nobody under ten can complete this form!")
return false
}
//
// Okay, age field is the correct length (2), so check if it contains anything but
the characters in the variable digits
//
for (var i=0;i<document.testform.age.value.length;i++){
temp=document.testform.age.value.substring(i,i+1)
if (digits.indexOf(temp)==-1){
//
// If we find anything but 0-9, display another (bad) message and return false
//
alert("Invalid Age !")
return false
   }
  }
//
// We made it to here - so the form must have passed all the validation tests
// To make sure it is sent, we must now return true
//
alert("All the fields of the form are checked. Well done!")
return true
}
// -->
</script>
```

The form contains just two fields to be checked (*Name* and *age*) and a *Submit* button; the form is defined like this:

```
<form name="testform" onSubmit="validate(testform)">
Name: <input size="40" name="Name">
Age:  <input size="2" name="age" maxlength="2">
<input type="submit" value="Submit Form">
```

```
</form>
```

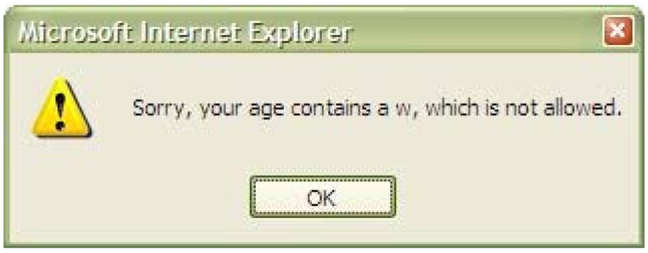
[The absence from this example of the *method* and *action* attributes does not mean that a 'real' form would not need them: it obviously would. But when we are testing the validation script we do not really need them. It is assumed that we will separately test that the form actually 'works': that is, it is sent when the *Submit* button is pressed, and with the correct values.]

Let's look a bit closer at the validation code:

- We are calling the function validate when the onSubmit event occurs (which happens when the user clicks on the Submit button.
- We are passing all the form's contents in a single go with `validate (testform)` where *testform* is the value of the form's *name* attribute.
- We access the individual fields as 'components' of the form, which is itself part of the document object, for which we use the 'dot notation': `document.testform.Name.value` refers to the *value* of the *Name* field in the *form* called testform - which is ultimately part of the *document*. We can safely leave out the document, but not the rest!
- By calling the function in this way - `return validate(testform)` - we are telling the browser to send the form *only* if the function 'returns true'.
- Inside the code we create a variable and 'populate' it with the ten numbers `(0123456789)` (the order does not matter), as we need to make sure that the age field contains nothing but numbers. We could do the same for all the letters `(abcdefg ... WXYZ)`, for letters and numbers `(abcd ... XYWX0123456789)`, or for numbers and a period `(0123456789.)`, depending on our validation requirements.
- We care carrying out three tests:

  1. Our design requires that the name field must have something in it, so we need to check that it is not empty; as the field's value is a string (`document.testform.Name.value`) we check that it is not a **null** string by comparing it with the null string "" (no space between the double quotes!).
     We should also note that, as a null string actually has a length of zero, we could have chosen instead to check that the string *length* is not zero (`document.testform.Name.value.length`, as length is a property of the value string. Remember also that a string is not empty (null) if has *any* characters in it (so its length is not zero); this includes spaces, so to deal with the possibility of a string containing just spaces, the code could go on to check if there were any 'real' characters in the string.
  2. We are using the age field's *maxlength* attribute in the form to prevent any number over 99 being entered (apologies for discriminating against centenarians), and we need to check if it is less than two characters long (apologies this time to the under-tens). [Programmers among you will find this a clumsy and inefficient way of doing things, and so it is; you can change the code to create a much neater version. The aim is to explain what is happening, and to introduce the length property first before we use it in the next section of code.]
     The messages are much better, but still not good enough.
  3. The final section of code actually has nothing new, in that it uses language constructs (the *for loop*, the *if* statement, and comparison operators) that we have seen in previous sessions, but it may not be immediately clear what is happening here. Essentially, the for loop steps us through the string we are checking (`age.value`) one character at a time. Inside the loop we 'extract' each character in turn - using the built-in **substring** function - into the variable temp, and we search to see if that character is found anywhere in our *digits* array, using the indexOf function. The indexOf function (notice the mix of upper and lower case) returns a value that tells us where in the digits array the first occurrence of the character in temp is. Here's a simple example to help us see what is happening. Say the user had typed 18 in the age field; the first time through the loop temp would set to 1 and indexOf would return the value 1 (the first element in *digits* (0) has an index of 0, and the second character (1, our match) has an index of 1). For the 8, indexOf would return a 7; we have reached the end of the string and found no illegal characters, so we move on to the next check. On the other hand, if the user had entered an illegal character in the string we would not find a match anywhere in the digits array and the indexOf function would return -1, so we would 'drop put' of the loop immediately and tell the user what has 'gone wrong'. Note also that the value of temp when we exit the loop is still the illegal character we encountered, so we could display it in our message

     ```
         alert("Sorry, your age contains a " + temp +", which is not allowed.")
     ```



Sorry, your age contains a w, which is not allowed.

This means that we 'drop out' as soon as we detect the first illegal character; if there are more, this simple code will not detect them and they may still be there when the user re-submits the form. We could certainly modify the code to detect all illegal characters in one 'pass'; it's a fairly simple challenge for those who want to extend their programming skills.

This discussion leads conveniently to a consideration of one of the most annoying features of many form validation systems: the so-called 'error' messages. This is a misnomer; users do not make errors - designers of web forms do a poor job. If you ask well-worded questions, give clear instructions, and provide sensible defaults, the only 'error' users make is to type the wrong information into a box, or make a wrong (incorrect) selection. Part of the process of good form design - as we saw in the last session - is to know what form elements to use in a specific circumstances (when to use check boxes or list box, for example), and how to use defaults properly. Part of the process of devising good validation is to give **helpful feedback** to users about what they need to do to successfully complete the form (so that it will pass the various validation tests).

In this light a message that says something like **Invalid input** is meaningless and embarassing (as a developer); it doesn't even tell the user which field has the problem, never mind what the problem is, or how to correct it. **There is a problem with the postcode** is better, but does not go far enough; exactly what is the problem? A message that says **Postcodes can only have numbers, and you have entered 'wg56'** is the correct style of message: it identifies the field, tells the user the kind of error (what we might think of - but users most certainly do not - as unacceptable non-numeric input) in a way that they can understand - and even highlights what they need to change (the 'wg' characters). Most importantly it does not 'blame' the user; one might even change the message to **Sorry, but Postcodes can only have numbers, and you have entered 'wg56'** to emphasize this (see the example above for an indication on how to do this).

Remember: it is the keyword return in the call to the **validate( )** function associated with the onSubmit event that tells the browser to send the form *only* if the 'value' returned by the validate( ) function is *true*. Without this, the form would be sent regardless of the value returned from the function.

Tasks

Simple validation

1. Create a **new web document** for this first exercise. Include the example script from the previous paragraphs in the *head* section and the the simple age-and-name form in the *body* section. You can change the form layout, of course, but if you change the 'details' of the form (especially the names of the fields, and the name of the form itself) you will also need to make appropriate changes to the script, and to how the *validate()* script is called using the *onSubmit* event.
2. Check that the script works.
3. Change the alert messages to be more meaningful for the user.
4. Are there any other 'rules' that you might want to apply to these two fields?
5. How would you modify what the user sees on the screen to indicate how a field is to be filled in, or that a field *must* be filled in?

On the other hand, we can recognise that many of these validation processes are used repeatedly, so that every time we want to check that a field is not empty we can use the same function. Given such a 'not empty' function, we just pass it a field's value (which is usually a string) as a parameter; the function returns 'true' (the string is not null (not empty)) or 'false' (the string is *null* (empty)). If we have several such fields to check, we just call the function once for each field, passing a different form element each time; if it returns false for any of them, we give the user a message and return false from the main validation function.

Here is perhaps the simplest possible example of such a routine:

```
function IsNull(str) {
  var isValid = false;
   if (str+"" == "null")
    isValid = true;
  return isValid;
 }
```

This function is passed a string (*val*) and checks if it is empty ('null'). The function returns a value of false (the default) or true (if the string is empty).

For this to be general approach to be effective we need a library of such routines (here it is as a zip file), each of which carries out a single check (IsNull, IsAlpha, ValidEmail ...); we then select the relevant routines and call them from a simple 'main' function that is different for each form we are processing. We can either copy just the required routines as needed, or place them all in a file and link that file into our pages by adding a tag like this in the head section:

```
<script src="datavalidation.js"></script>
```

As with external style sheets, such links to external files of JavaScript functions generally go into the *head* section, and certainly *above* any scripts that may use any of the functions.

Methods and events

One way of thinking of the scripting process is to see it is as an example of *event-driven* programming. Events occur when the user takes an action (such as moving the mouse over an image, or clicking on a link); scripts act as *event-handlers*, specifying what action is to be taken when a particular event occurs for a given object. In reality, only a very few (if any) events associated with the many objects in a document will actually need such event-handling code. However, one critical event is the one that we are concerned with in this session; when the user clicks on a form's *Submit* button a **Submit** event occurs. It is this event that we respond to with appropriate code (our script) to validate the form's contents *before* the form is delivered by the browser.

Objects and properties

You may have wondered about one aspect of the code used in some of the scripts, an example being *document.testform.age.value*. This is an indicator of the simple *object hierarchy* of Javascript. The 'top' level object is the **window** object; this contains objects like **frame**, **history**, **screen**, and **document**. In turn, the **document** object contains other objects, including the **form** object. Using the common 'dot' notation, this gives us the full object definition **window.document.form**, but the window part is often omitted. We use the name of a specific form ('testform'), and each form has a number of (named) elements, such a text box called 'age'; this gives us **document.testform.age**. Finally, each form element has a a number of **properties**, one of which is its *value* (usually a string, sometimes a number or a boolean), giving us **document.testform.age.value**.

Validating a 'real' form

We will use the form that you worked on in a previous practical to look at the validation process. I have created a basic version of that form which you can look at; you might want to check your efforts against it (they might be better than mine), or even use it in this exercise (here it is as a zip file).

1. Start with a simple script that checks that one of the text fields (such as the postcode) has not been left blank. Link the script to the *onSubmit* event, and make sure that the return value of the function is used to prevent the form being sent if the field has been left blank. The script should also tell the user (using intelligible alert messages) why the form is not being sent so that they can correct the error. Saying "Invalid postcode!" is unhelpful and rather annoying; saying "Your postcode can't have less than four numbers" is neither.
   You could

- base the script on the example script given earlier (the most obvious starting point)
- search for a script to use from one of the many online script 'repositories'
- look for online tutorials on form validation
- use the library of routines describe earlier

2. Extend the script code to check that the length of the entered postcode is *exactly* four characters.
3. Extend the script code further to check that all the characters in the entered postcode are numbers [0-9].
4. How would you validate a checkbox, radio button or selection (list or combo box)? Do you use their *value* property, as with input (text) boxes? Do they even *have* a value property? If not, what other 'properties' would you use?
   Hint One: Radio buttons and checkboxes are selected by default if we set them as *checked*; is this a 'property' that we could use?.
   Hint Two: We can think of option boxes (list and combo) as being similar to certain Visual Basic controls; perhaps they also have some kind of *index* property that tells us which item has been selected. If so, might we not want to think of them as some form of *array*?
5. We usually validate a form by passing all its values at once, then validating those fields that we are interested in. How many ways can you find to do this?
6. In your own time if necessary, continue to develop the validation script until all the appropriate fields are properly checked. The only fields that do not need validation are those which are *optional* (not required to be completed), or those that have a pre-set default that cannot be 'removed' (as opposed to changed) by the user - namely radio buttons. All other fields - including those with defaults that the user can remove, like selection lists, text fields and checkboxes - should be considered for validation. Depending on the circumstances, this may mean all fields, only some fields, or no fields at all. **Merely telling the user on the form that certain fields are required does not absolve the form designer from the responsibility of providing adequate validation.** Effective validation complements good processing; this must include correct data storage and manipulation, and meaningful feedback to the user. The latter might include
   - acknowledgement of receipt of the form
   - thanks for completing the form (if voluntary)
   - another form to be completed, including where necessary data from the previous form as *hidden* fields
   - information about the 'consequences' of completing the form (information about when goods that have been ordered will be delivered, for example)
   - if there is a problem with the contents of the form (beyond what was detected by validation), the form should be 'returned' with the entered data still included, the problem bring clearly indicated; no user wants - *or needs* - to enter it all again!

JavaScript

Given that JavaScript is the 'star' of this week's whole session, it seems a bit strange to still have a JavaScript section, but I don't want to change the buttons so ... let's do something that uses forms and validation but is just for fun.

Lottery Number Generator

Create a new web document and put the following code into the body section:

```
<div align="center">
<h3>Lottery Numbers</h3>
<form name="lotto">
<p>Pick  <select name="numbercount">
<option value="1">1</option>
<option value="2">2</option>
<option value="3">3</option>
<option value="4">4</option>
<option value="5">5</option>
<option value="6" selected>6</option>
<option value="7">7</option>
<option value="8">8</option>
<option value="9">9</option>
<option value="10">10</option>
</select>
numbers from 1 to <input type="text" name="maxnum" value="49" size="2"
```

```
maxlength="2"><br> <br>
<input type="button" value="Pick my numbers!" onClick="numbers()">
<br> <br><textarea name="results" rows="11" cols="15"></textarea></p>
</form>
</div>
```

and put the following script code into the head section:

```
<script language="JavaScript">
function numbers() {
var nummenu = document.lotto.numbercount;
var numbercount = nummenu.options[nummenu.selectedIndex].value*1;
var maxnumbers = document.lotto.maxnum.value*1;
if (numbercount > maxnumbers) {
alert("The highest value must be more than "+numbercount+"  ");
}
else {
var ok = 1;
r = new Array (numbercount);
for (var i = 1; i <= numbercount; i++) {
r[i] = Math.round(Math.random() * (maxnumbers-1))+1;
}
for (var i = numbercount; i >= 1; i--) {
for (var j = numbercount; j >= 1; j--) {
if ((i != j) && (r[i] == r[j])) ok = 0;
}
}
if (ok) {
var output = "";
for (var k = 1; k <= numbercount; k++) {
output += "Number " + k + " = " + r[k] + "\n";
}
document.lotto.results.value = output;
}
else numbers();
}
}
</script>
```

When you run the page it will be clear that we are using a form to give us a way of deciding our range of numbers (1 to 40, say) and how many we want to draw out. The text area is being used to display the choices, and if we enter a value in the text box that is less than the number in the drop-down list, we have put in some simple validation to tell us so. However, there is no validation on what the user types into that text box, and the impact on the output if they enter something besides is going to be unpredictable or unpleasant; you should be able to add that validation to the early part of the script, using what you have learnt earlier in this session.
The rest of the script uses JavaScript's **Math** object and **random** method to generate the required number of random integer values, within the requested range, and write them into the textarea field (`document.lotto.results.value = output;`). Programmers can pore over the mechanics of this, but everybody should ignore that but take a close look at the last half dozen lines of code, as they will reveal a couple of interesting programming 'tricks' that might be useful in the future:

- The results are written into the string *output* inside the loop by using the `+=` (add) operator, which adds together (concatenates) strings to produce a single string (it also works with numbers, but they are added together arithmetically, of course).
- Even though *output* is a single string, we want to display the different parts on separate lines in the textarea field, so we are writing a `\n` (newline character) at the end of each section. You might have though about writing an HTML tag such as <br>, but you will find that it will not achieve the desired effect; tags are not rendered inside form elements (including the textarea).

PHP

A simple email system

This example creates a simple email system (and it can only be tested properly if you are online). Create a new web document and copy the following code (it's basically just another form) into the body section. Make sure that you save it onto (or upload it to) your home area on the Deakin server, but it doesn't matter if it is with a .php or .html extension, as it contains no PHP code. Change the default value in the **from** field to your own email address; note that it is disabled to prevent the user editing it. [You could change this of course; indeed, you could enhance the whole interface in many ways: by using other forms elements to allow the user to pick from a list of people to emails to, and so on. It is described as a *simple* email system, after all!]

```
<h2 style="color:#ff0000">Send Email</h2>
<form method="post" action="sendmail.php" name="email">
<p><strong>To:</strong>  <input type="text" name="to" value="" size="30"></p>
<p><strong>From:</strong>  <input type="text" name="from"
value="me@deakin.edu.au"></p> <p><strong>Subject:</strong>  <input type="text"
name="subject" value="" size="30"></p>
<p><strong>Message:</strong>  <input type="text" name="message" value=""
size="80"></p>
<p> </p>
<p>   <input type="submit" value="Send"></p>
</form>
```

Create a second new web document and copy the following code into its body section. Save it into your home area on Deakin (or upload it by FTP); this one *must* have a .php extension, and it **must** have the same filename as appears in the form's *action* in the first file; if you changed that from sendmail.php you have to save the second with the name you changed it to.

```
<?php
ini_set(SMTP,"mail.deakin.edu.au");

$to = $_REQUEST['to'] ;
$from = $_REQUEST['from'] ;
$subject = $_REQUEST['subject'] ;
$message = $_REQUEST['message'] ;

$fromfield = "From: " . $from;
mail ($to, $subject, $message, $fromfield);
echo ("Email sent to ".$to." as requested");
?>
```

When you deliver the first file from the Dealin server (for example, http://www.deakin.edu.au/~jmsith/sit104/prac6/mail.php) you should see the form. Enter an email address into the **To:** field, a subject and a message, then click the Send button. You should get a new page saying that an email message has been sent to whoever was in your To: field, and when you check your email it should be there.

This simple system introduces several important aspects of working with PHP that we will develop over the next few weeks:

- We are sending information from a form in one page to a separate PHP page; that second page contains code that 'processes' it (here it uses it to generates an email, but it could be writing it to a file or database), and also creates and sends a 'reply' message (as a web page).
- The real 'work' happens in the second file, which must be executed off the server so it must be identified as PHP (by its file extension). Several quite important things happen in this code:
    - In the first statement we are telling the system what mail [SMTP] server we are using; we are actually creating a (very) temporary extension to the system's PHP settings (normally accessed by the server in the php.ini file) to tell it what server to contact to deliver our mail message. We would also have to define what *port* the SMTP server uses if it were not the default port number, 25; conveniently Deakin's SMTP server does use the default port so we don't have to set it.
    - The block of four lines that include **$_REQUEST** are very important; we use that function whenever we want to extract the contents of forms to use in our PHP code. As we saw in the session on forms (and in the lectures) the browser bundles togther the form's contents (name+values) and sends them to the processing progam (as defined in the action) as a string. The $_REQUEST function is how we extract those names and values from that string in PHP; it returns the value of the form element that is given to it as a parameter, into a variable. So in the first line of the four we are retrieving the value of the field whose name

was *to* and storing it in the variable called (not very originally) **$to**; in the second we are retrieving the value of the *from* field and storing it in **$from** - and so on. The names of the variables can be whatever you want (starting with a $), but the names of the fields must be in single quotes and must match *exactly* the names of the fields in the originating form (including the case of all characters). The values returned are generally strings, so the variables we are using ($to, $from etc.) will be of string type; any operations we do on those should treat them as strings.

- Once we have retrieved these values we can use them in any way that suits our needs. In this simple example we are using them to format a mail message which is sent with PHP's **mail** function.
  This example has no form validation or error checking, so if the user did not fill in all the fields (especially the **to** field) the mail function would fail, as the SMTP server would reject the message and our script would 'crash'; you can easily add this as form validation to the first page, or as simple error trapping in the second. Think about why it would be better to do it on the first page!
- The final line of code is an echo statement to send back a message to the user; note that we have included some of the information they sent in the message. If we greatly extended this idea, we could create a full page 'echoing' all the information they sent, and neatly formatting it; we will certainly want to do something like this in our processing of order forms later in this unit (and in Assignment Two). Indeed, we will start to look at this in more depth, starting with the next session.

Summary

In this session you have looked at how to

- define which fields in a fill-in form require validation
- define appropriate rules for the validation of a field
- modify short JavaScript scripts to validate the content of fields
- access form fields in PHP

Quiz

This simple quiz should help you to check that you have covered the material of this session, and that you know the key concepts and techniques.

1. Which event do we usually use to validate form data?	onClick ▾
2. A string that has no content is called...	nill ▾
3. To prevent a form being sent a validation function should return a value of ...	true ○ false ○
4. Which JavaScript function can be used to tell us if a characters occurs in string?	indexUp ▾
5. The process of checking that a form has been properly completed before it is sent is called ...	Verification ▾
6. Which PHP function is used to extract form data for processing?	$_SUBMIT ▾
7. Which tag is used to include an external JavaScript file into a web document?	link ▾
8. In Javascript, Math is an example of ...	A Function ▾
9. onSubmit is an example of ...	An Occurrence ▾

Check my answers