By icarus

Table of Contents

Code To Zero

One of my most common activities during the development cycle for a Web application involves writing code to validate the data entered into online forms.

This might seem like a trivial task, but the reality − as anyone who's ever spent time developing a robust Web application will tell you − is completely different. Data verification is one of the most important safeguards a developer can build into an application that relies on user input, and a failure to build in this basic error checking can snowball into serious problems (and even cause your application to break) if the data entered is corrupt or invalid.

In order to illustrate this, consider a simple example, culled from my own experience: an online loan calculator that allows a user to enter the desired loan amount, finance term and interest rate. Now, let's assume that the application doesn't include any error checks. And let's also suppose that the user decides to enter that magic number, 0, into the term field.

I'm sure you can imagine the result. The application will perform a few internal calculations that will end in it attempting to divide the total amount payable by the specified term − in other words, division by zero. The slew of ugly error messages that follow don't really bear discussion, but it's worth noting that they could have been avoided had the developer had the foresight to include an input−validation routine while designing the application.

The things about data validation, though, is that it's one of those things you can't hide from. Even if you develop and release a Web application without building in any validation routines (either through ignorance or laziness), you can be sure that your customer's going to demand a fix for it in the next release of the software. And since − as I said right at the beginning of this article − it's one of the few things you're likely to do over and over again when building Web applications, it's worthwhile spending a little time to make the process as painless as possible.

That's where this article comes in. Over the next few pages, I'll be attempting to build a reusable library of functions for form input validation, in an attempt to save myself (and, hopefully, 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 PHP class 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 PHP, and you'll have very little difficulty customizing it so that it works for you.

Let's get going!

Back To Class

Before we begin, let's just go over the basics quickly:

In PHP, a "class" is simply a set of program statements which perform a specific task. A typical class definition contains both variables and functions, and serves as the template from which to spawn specific instances of that class.

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

This comes in handy in situations where you need to spawn more than one instance of an object − for example, two simultaneous database links for two simultaneous queries, or two shopping carts. Classes also help you to separate your code into independent modules, and thereby simplify code maintenance and changes.

A class definition typically looks like this:

```
<?php

class ShoppingCart
{
// this is where the properties are defined

var $items;
var $quantities;
var $prices;
...

// this is where the methods are defined

function validate_credit_card()
{
// code goes here
}
...
}

?>
```

Once the class has been defined, an object can be spawned with the "new" keyword and assigned to a PHP variable,

```
<?php
```

```
$myCart = new ShoppingCart;

?>
```

which can then be used to access all object methods and properties.

```
<?php

// accessing properties
$myCart->items = array("eye of newt", "tail of lizard", "wings
of bat");
$myCart->quantities = array(9, 4, 14);

// accessing methods
$myCart->validate_credit_card();

?>
```

The Bare Bones

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 class 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 result code.

2. A PHP structure (here, an associative 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. It's important to note, however, that these methods will have nothing to do with the visual presentation of either the form or the form's result page; rather, they provide a simple API to perform validation of the data entered into the form, and are designed for use by server−side scripts which are more closely connected to the presentation layer.

How Things Work

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

```
<?php

// some form processor
// let's assume that the variables $a and $b
// have been obtained via a form POST operation

// create an object
$fv = new FormValidator();

// run validation methods
// the first argument is the name of the field to validate
// the second is the error to report if validation fails
$fv->isString("a", "Please enter a string for field A");

// same here
$fv->isNumber("b", "Please enter a string for field B");

// check to see error status
echo $fv->isError();

?>
```

As you can see, I would like to call class methods with two parameters: the name of the form variable to validate, and the error message to return in the event the validation test fails. With this in mind, it becomes much easier to design the class appropriately.

Once the basic functionality of the class 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($var, $errormsg) − check whether the specified form variable is empty;

isString($var, $errormsg) − check whether the specified form variable contains string data;

isNumber($var, $errormsg) − check whether the specified form variable contains numeric data;

isWithinRange($var, $errormsg) − check whether the specified form variable contains a value within the specified numeric range;

isEmailAddress($var, $errormsg) − check whether the specified form variable contains a valid email address;

isError() − check whether any errors have occurred in form validation;

getErrorList() − return the list of validation errors;

resetErrorList() − reset the error list.

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

Private Eye

Right. So I now know how the class is supposed to work, plus I have a fairly good idea of what the methods encapsulated within it will look like. This is a good time to start writing some code.

You might find it helpful to download the complete class code at this point, so that you can refer to it over the next few pages.

fv.zip

Let's begin by setting up the class definition:

```
// FormValidator.class.inc
// class to perform form validation

class 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.

```
<?php

class FormValidator
{
//
// private variables
//

var $_errorList;

// snip
}
```

Note the underscore (_) prefixed to this variable; I've done this in order to provide a convenient way to visually distinguish between private and public class variables and functions.

You'll remember, from my schematic of how the class will be used on the previous page, that each validation method will be passed the name of the form variable to be tested as a string. Now, I need a method which will take this string, and use it to obtain the value of the corresponding form variable.

```
<?php
class FormValidator
{
// snip

//
// methods (private)
//

// function to get the value of a variable (field)
function _getValue($field)
{
global ${$field};
return ${$field};
}

// snip
}
?>
```

In this case, the _getValue() method will be passed a string representing a particular form variable − for example, "a" − and it will return the value of the corresponding variable in the global scope − for example, $a. Notice how I've used PHP's powerful variable interpolation features to dynamically construct a variable name, and thereby access the corresponding value (more information on this is available at http://www.php.net/manual/en/language.variables.variable.php).

Again, this is a private method − it will be used internally by the class, and will never be exposed to the calling script. As you will see, this function will be used extensively by the public validation routines to get the current value of the named form variable, and test it for errors.

This probably seems like a convoluted way of doing things − after all, you might be wondering, why not just pass the value to be validated directly to the class method? There's a good reason for this, which I'll be explaining a little further down. For the moment, though, just trust that there is method to my madness.

Note also that the _getValue() method can be improved by modifying it to return the value of the form variable from the special $HTTP_POST_VARS array, rather than from the global scope. I haven't done this here because I'm not sure whether my forms will be using GET or POST; however, if you're sure that you're going to be using POST, this is probably an improvement you will want to make. And in case this didn't make any sense to you, take a look at http://www.php.net/manual/en/language.variables.predefined.php which provides more information on the special $HTTP_POST_VARS array.

Running On Empty

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.

```
<?php
class FormValidator
{
// snip

//
// methods (public)
//

// check whether input is empty
function isEmpty($field, $msg)
{
$value = $this->_getValue($field);
if (trim($value) == "")
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
else
{
return true;
}
}

// snip
}
?>
```

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

The isEmpty() public method is called with two arguments: the name of the form variable to be tested, and the error message to be displayed if it is found to be empty. This method internally calls the _getValue() method to obtain the value of the named form variable; it then trim()s this value and checks to see if it is empty.

If the variable is empty, a new element is added to the private class variable $_errorList (an array) to represent the error. This new element is itself a three−element associative array, with the keys "field", "value" and "msg", representing the form variable name, the current value of that variable, and the corresponding error

message respectively.

If the variable is not empty, it will pass the test and the method will simply return true; this is handy in case you want to wrap the method call in a conditional test in your form processing script.

The reason for passing the form variable's name to the class method, rather than using its value directly, should also now be clear. When I pass the variable name to the method as a string, it can be used to access the corresponding form variable via the _getValue() method, and also becomes available within the class scope. This allows me to populate the $_errorList array with both the form variable name and its value; this additional information can come in handy when debugging long and complex forms.

If, instead, I had used the form variable's value directly, I would have certainly been able to record the incorrect form value in the $_errorList array, but would have no way to record the corresponding form variable name.

This might sound overly complicated, but it's actually fairly simple. In order to understand it better, try altering the class methods to accept form data directly, and you will see that it becomes much harder to record the name of the offending form variable(s) in the $_errorList array.

Finally, in case you're wondering, the $this prefix provides a convenient way to access variables and functions which are "local" to the class.

Let's move on to a couple of other, similar functions:

```
<?php
class FormValidator
{
// snip

// check whether input is a string
function isString($field, $msg)
{
$value = $this->_getValue($field);
if(!is_string($value))
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
else
{
return true;
}
}

// check whether input is a number
function isNumber($field, $msg)
{
```

```
$value = $this->_getValue($field);
if(!is_numeric($value))
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
else
{
return true;
}
}
}

?>
```

These are very similar, simply using PHP's built−in variable functions to determine whether the data passed to them is numeric or character data.

Floating Like A Butterfly

Two useful corollaries of the isNumber() method are the isInteger() and isFloat() methods, which provide for more precise data validation capabilities when dealing with numeric data.

```
<?php
class FormValidator
{
// snip

// check whether input is an integer
function isInteger($field, $msg)
{
$value = $this->_getValue($field);
if(!is_integer($value))
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
else
{
return true;
}
}

// check whether input is a float
function isFloat($field, $msg)
{
$value = $this->_getValue($field);
if(!is_float($value))
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
else
{
return true;
}
}

}

?>
```

Another very useful method, and one which I find very useful when building forms which ask for the user's age, is the isWithinRange() method − it provides an easy way to check whether the input is within a certain numeric range.

```
<?php
class FormValidator
{
// snip

// check whether input is within a valid numeric range
function isWithinRange($field, $msg, $min, $max)
{
$value = $this->_getValue($field);
if(!is_numeric($value) || $value < $min || $value >
$max)
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
else
{
return true;
}
}
}

?>
```

Mail Dot Com

It's also possible to create more complex validation routines using PHP's built−in support for regular expressions. This next method, isAlpha(), uses a regular expression to test whether all the characters in the input string are alphabetic.

```
<?php
class FormValidator
{
// snip

// check whether input is alphabetic
function isAlpha($field, $msg)
{
$value = $this−>_getValue($field);
$pattern = "/^[a−zA−Z]+$/";
if(preg_match($pattern, $value))
{
return true;
}
else
{
$this−>_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
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) class method isEmailAddress() demonstrates:

```
<?php
class FormValidator
{
// snip

// check whether input is a valid email address
function isEmailAddress($field, $msg)
{
$value = $this−>_getValue($field);
```

```
$pattern =
"/^([a-zA-Z0-9])+([\.a-zA-Z0-9_-])*@([a-zA-Z0-9_-])+(\.[a-zA-Z0-9_-]+)+/
";
if(preg_match($pattern, $value))
{
return true;
}
else
{
$this->_errorList[] = array("field" => $field,
"value" => $value, "msg" => $msg);
return false;
}
}

}

?>
```

Under Construction

So that takes care of the basic validation routines. As you can see, it's fairly easy to add new ones to the class, as per your specific requirements. All that's left now are a couple of methods to access the error list.

The first of these is a simple little method named isError(), which lets you know whether or not any errors have occurred yet. Internally, all this method does is check 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:

```
<?php
class FormValidator
{
// snip

// check whether any errors have occurred in validation
// returns Boolean
function isError()
{
if (sizeof($this->_errorList) > 0)
{
return true;
}
else
{
return false;
}
}

}

?>
```

Of course, all isError() does is tell you whether or not an error occurred; it won't let you view the list of errors. For that, you need the getErrorList() method, which returns the current $_errorList array.

```
<?php
class FormValidator
{
// snip

// return the current list of errors
function getErrorList()
{
return $this->_errorList;
```

```
}

}

?>
```

Short and sweet!

Finally, how about resetting the error list? Well, that's why I have the resetErrorList() method, which clears the $_errorList array of all data.

```
<?php
class FormValidator
{
// snip

// reset the error list
function resetErrorList()
{
$this->_errorList = array();
}

}

?>
```

It's a good idea to run this resetErrorList() method whenever the class is first instantiated − which is why I've added it to the class constructor.

```
<?php
class FormValidator
{
// snip


// constructor
// reset error list
function FormValidator()
{
$this->resetErrorList();
}

}

?>
```

In case you didn't already know, PHP makes it possible to automatically execute a specific function when a new instance of a class is spawned. This function is referred to as a "constructor" and must have the same name as the class. Which, if you look at the code above, is exactly what I've done.

A Quick Snack

At this stage, I think I have enough building blocks to actually begin using this class to perform form input validation. Keep in mind, though, that I've been wrong before, and so my initial feeling of satisfaction may soon disappear.

First, let's build a simple form:

```
<html>
<head>
<basefont face="Arial">
</head>

<body>
<form action="processor.php" method="POST">

<b>Name:</b>
<br>
<input type="text" name="name" size="15">

<p>

<b>Age:</b>
<br>
<input type="text" name="age" size="2" maxlength="2">

<p>

<b>Sex:</b>
<br>
<input type="Radio" name="sex" value="m">Male
<input type="Radio" name="sex" value="f">Female

<p>

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

<p>

<b>Favourite sandwich filling:</b>
```

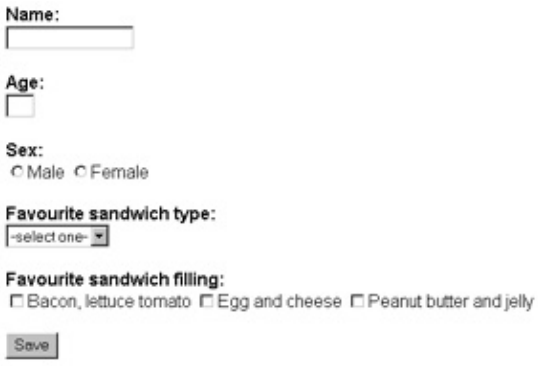
```
<br>
<input type="Checkbox" name="sfill[]" value="BLT">Bacon,
lettuce tomato
<input type="Checkbox" name="sfill[]" value="EC">Egg and
cheese <input
type="Checkbox" name="sfill[]" value="PBJ">Peanut butter and
jelly

<p>

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

</body>
</html>
```

Here's what it looks like:



Now, when this form is submitted, the PHP script "processor.php" will take over. This script will first need to verify that the data entered into the form is acceptable (that's where our newly−minted FormValidator class comes in) and then do something with it (for example, INSERT the data into a MySQL database). Take a look:

```
<?php
// include class
include("FormValidator.class.inc");

// instantiate object
$fv = new FormValidator();

// perform validation
$fv->isEmpty("name", "Please enter a name");
$fv->isNumber("age",
"Please enter a valid age"); $fv->isWithinRange("age", "Please
enter an
```

```
age within the numeric range 1-99", 1, 99);
$fv->isEmpty("sex", "Please
enter your sex"); $fv->isEmpty("stype", "Please select one of
the listed
sandwich types"); $fv->isEmpty("sfill", "Please select one or
more of
the listed sandwich fillings");

if ($fv->isError())
{
$errors = $fv->getErrorList();

echo "<b>The operation could not be performed because one or
more error(s) occurred.</b> <p> Please resubmit the form after
making
the following changes:";
echo "<ul>";
foreach ($errors as $e)
{
echo "<li>" . $e['msg'];
}
echo "</ul>";
}
else
{
// do something useful with the data
echo "Data OK";
}

?>
```

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

```
The operation could not be performed because one or more
error(s)
occurred.
Please resubmit the form after making the following changes:
* Please enter a name
* Please enter a valid age
* Please enter an age within the numeric range 1-99
* Please enter your sex
* Please select one of the listed sandwich types
* Please select one or more of the listed sandwich fillings
```

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

```
The operation could not be performed because one or more
error(s)
occurred.
Please resubmit the form after making the following changes:
* Please enter an age within the numeric range 1-99
```

And here's what happens when I repent, correct all my errors and submit the form with valid data:

```
Data OK
```

Going To The Source

Of course, this is just my first stab at a generic form validation class. It's designed for very simple requirements, and may be way too primitive for your requirements. If this is the case, you have two basic options:

1. File the results of my efforts in the trash can and write your own, much−cooler, does−everything−but−make−toast class;

2. Pick up a free, open−source PHP class which offers a more powerful feature set.

If you picked door one, you don't really need my help any more. You can stop reading right now and get to work. Have fun, and remember to send me a Christmas card if you sell your software for a million bucks.

If, on the other hand, you're lazy and figured that door two was more promising, you'll be happy to hear that the Web has a huge number of powerful form validation classes floating around it, many of them extremely powerful. Here are two that looked particularly interesting:

Manuel Lemos' form class allows you to programmatically construct HTML forms, offering extremely powerful client− and server−side validation features to make your forms as bulletproof as possible. The class is freely available at http://www.phpclasses.org/browse.html/package/1, and comes with excellent documentation and examples.

Here's an example of a simple online form created with this class:

```
<html>
<head><basefont face="Arial"></head>
<body>
<?php

// function to output form
// called by class Output() method
function displayForm($str)
{
echo $str;
}

// include class
include("forms.php");

// instantiate object
$contact = new form_class;

// set some form properties
$contact->NAME="contact";
$contact->METHOD="POST";
$contact->ACTION=$PHP_SELF;
```

```
// start building the form fields

// add a name field
$contact->AddInput(array(
"TYPE"=>"text",
"NAME"=>"name",
"MAXLENGTH"=>50,
"ValidateAsNotEmpty"=>1,
"ValidateAsNotEmptyErrorMessage"=>"You must
enter your name" ));

// note that you can specify how to validate the data while
adding a
field // a number of different validation routines are
available // look
at the documentation for details

// add an email address field
$contact->AddInput(array(
"TYPE"=>"text",
"NAME"=>"email",
"MAXLENGTH"=>150,
"ValidateAsEmail"=>1,
"ValidateAsEmailErrorMessage"=>"You must enter a valid email
address",
));

// add a textarea box for message
$contact->AddInput(array(
"TYPE"=>"textarea",
"NAME"=>"message",
"ROWS"=>6,
"COLS"=>40,
"ValidateAsNotEmpty"=>1,
"ValidateAsNotEmptyErrorMessage"=>"You must
enter a message", ));

// and so on
// you can also add radio buttons, list boxes and check boxes

// add a submit button
$contact->AddInput(array(
"TYPE"=>"submit",
"NAME"=>"submit",
"VALUE"=>"Send Message"
));

// check to see if the form has been submitted
```

```
// use the form variable "submit" as the key
if ($contact->WasSubmitted("submit"))
{
// if form has been submitted
// read in all the form data
$contact->LoadInputValues(TRUE);

// validate the data on the server-side again
// invalid fields are stored in the array $invalid[]
// the first error message is stored in $error
$error = $contact->Validate(&$invalid);

if ($error != "")
{
// if error, report it
echo "The following error occurred: $error";
}
else
{
// else mail the form data to the webmaster

// and report result
echo "Your message was successfully sent.";
}
}
else
{
// form not yet submitted
// so display it to the user

// begin constructing the form
// using the field definitions above
// note how AddDataPart() can be used to intersperse
// HTML code between the form fields
$contact->AddDataPart("<br> Name: <br>");
$contact->AddInputPart("name");
$contact->AddDataPart("<br> Email address: <br>");
$contact->AddInputPart("email");
$contact->AddDataPart("<br> Message: <br>");
$contact->AddInputPart("message");
$contact->AddDataPart("<br>");
$contact->AddInputPart("submit");
}

// all done
// now output the form/form result
// this function dumps the HTML form (or its result)
// together with all required JavaScript code
$error = $contact->Output(array(
```

```
"Function"=>"displayForm",
"EndOfLine"=>"\n"
));

// end
?>
</body>
</html>
```

As you can see, Lemos' class exposes a number of methods to dynamically construct form elements, and to apply validation rulesets to them. These form elements can then be printed to the browser, and the data entered into the form can be validated using the built−in client−side and server−side validation functions.

If you're looking for something a little simpler, consider David Tufts' validation class, which provides a an easy−to−use, albeit primitive, form validation API that would probably be best suited to smaller applications. This class is available online at http://dave.imarc.net/php.php

Here's an example of a simple online form created with this class:

```
<html>
<head>
<basefont face=Arial>
</head>

<body>

<?php
if (!$submit)
{
?>
<form action="<?=$PHP_SELF?>" method="POST">
Name:
<br>
<input type="text" name="name" size="30">
<p>
Email address:
<br>
<input type="text" name="email" size="30">
<p>
Message:
<br>
<textarea name="message" cols="45" rows="6"></textarea>
<p>
<input type="submit" name="submit" value="Send">
</form>
<?php
}
```

```
else
{
// include class
include("form_validator.class");

// instantiate object
$fv = new form_validator();

// specify fields to be checked
// these fields are only checked to ensure that they contain
values
// advanced data type validation is not possible

if ($fv->validate_fields("name, email, message"))
{
// if no errors
// send out mail

// report status
echo "Your message was successfully sent.";
}
else
{
// or list errors
echo "The form could not be processed because the
following fields contained invalid data:";
echo "<ul>";
foreach ($fv->error_array as $e)
{
echo "<li>$e";
}
echo "</ul>";
echo "Click <a href=javascript:history.back()>here</a>
to go back and correct the errors";
}
}
?>
</body>
</html>
```

Short and very sweet − and perfect for applications that don't require advanced data validation!

Closing Time

And that's about all for the moment. In this article, you expanded your knowledge of PHP'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 met 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 on Linux/i586 with Apache 1.3.20 and PHP 4.1.1. 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. YMMV!