Chapter 6. Making Web Forms

Form processing is an essential component of almost any web application. *Forms* are how users communicate with your server: signing up for a new account, searching a forum for all the posts about a particular subject, retrieving a lost password, finding a nearby restaurant or shoemaker, or buying a book.

Using a form in a PHP program is a two-step activity. Step one is to display the form. This involves constructing HTML that has tags for the appropriate user-interface elements in it, such as text boxes, checkboxes, and buttons. If you're not familiar with the HTML required to create forms, the "Forms" chapter in *HTML & XHTML: The Definitive Guide*, by Chuck Musciano and Bill Kennedy (O'Reilly) is a good place to start.

When a user sees a page with a form in it, she inputs the information into the form and then clicks a button or hits Enter to send the form information back to your server. Processing that submitted form information is step two of the operation.

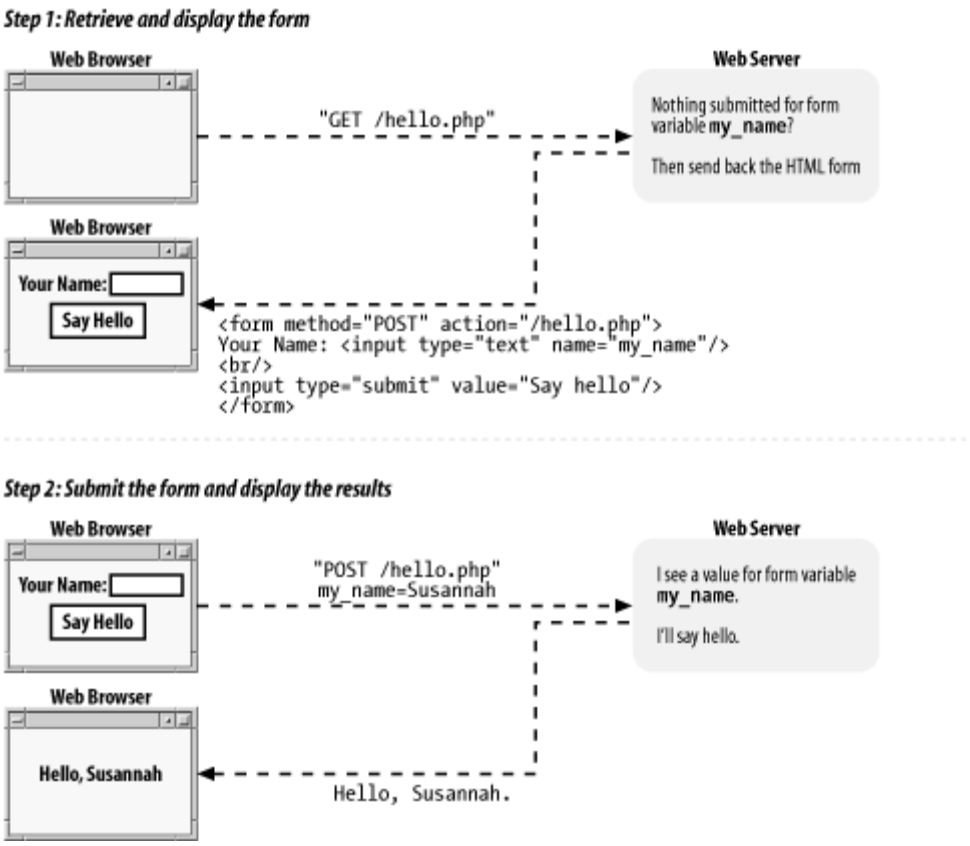
Example 6-1 is a page that says "Hello" to a user. If a name is submitted, then the page displays a greeting. If a name is not submitted, then the page displays a form with which a user can submit her name.

Example 6-1. Saying "Hello"

```
if (array_key_exists('my_name',$_POST)) {
    print "Hello, ". $_POST['my_name'];
} else {
    print<<<_HTML_
<form method="post" action="$_SERVER[PHP_SELF]">
 Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
</form>
_HTML_;
}
```

Remember the client and server communication picture from Chapter 1? Figure 6-1 shows the client and server communication necessary to display and process the form in Example 6-1. The first request and response pair causes the browser to display the form. In the second request and response pair, the server processes the submitted form data and the browser displays the results.

Figure 6-1. Displaying and processing a simple form



The response to the first request is some HTML for a form. Figure 6-2 shows what the browser displays when it receives that response.

Figure 6-2. A simple form

The response to the second request is the result of processing the submitted form data. Figure 6-3 shows the output when the form is submitted with `Susannah` typed in the text box.

Figure 6-3. The form, submitted

The pattern in Example 6-1 of "if form data has been submitted, process it; otherwise, print out a form" is common in PHP programs. When you're building a basic form, putting the code to display the form and the code to process the form in the same page makes it easier to keep the form and its associated logic in sync.

The form submission is sent back to the same URL that was used to request the form in the first place. This is because of the special variable that is the value of the `action` attribute in the `<form>` tag: `$_SERVER[PHP_SELF]`. The `$_SERVER` auto-global array holds a variety of information about your server and the current request the PHP interpreter is processing. The `PHP_SELF` element of `$_SERVER` holds the pathname part of the current request's URL. For example, if a PHP script is accessed at *http://www.example.com/store/catalog.php*, `$_SERVER['PHP_SELF']` is `/store/catalog.php`[1] in that page.

[1] As discussed in Example 4-18, the array element `$_SERVER['PHP_SELF']` goes in the here document without quotes around the key for its value to be interpolated properly.

The `$_POST` array is an auto-global variable that holds submitted form data. The keys in `$_POST` are the form element names, and the corresponding values in `$_POST` are the values of the form elements. Typing your name into the text box in Example 6-1 and clicking the submit button makes the value of `$_POST['my_name']` whatever you typed into the text box because the `name` attribute of the text box is `my_name`.

So, testing whether there is a key called `my_name` in the `$_POST` array tests to see whether a form parameter called `my_name` has been submitted. Even if the `my_name` text box has been left blank, `array_key_exists( )` returns `true` and the greeting is printed.

The structure of Example 6-1 is the kernel of the form processing material in this chapter. However, it has a flaw: printing unmodified external input—as `print "Hello, ". $_POST['my_name'];` does with the value of the `my_name` form parameter—is dangerous. Data that comes from outside of your program, such as a submitted form parameter, can contain embedded HTML or JavaScript. Section 6.4.6, later in this chapter, explains how to make your program safer by cleaning up external input.

The rest of this chapter provides details about the various aspects of form handling. Section 6.2 dives into the specifics of handling different kinds of form input, such as form parameters that can submit multiple values. Section 6.3 lays out a flexible, function-based structure for working with forms that simplifies some form maintenance tasks. This function-based structure also lets you check the submitted form data to make sure it doesn't contain anything unexpected. Section 6.4 explains the different ways you can check submitted form data. Section 6.5 demonstrates how to supply default values for form elements and preserve user-entered values when you redisplay a form. Finally, Section 6.6 shows a complete form that incorporates everything in the chapter: function-based organization, validation and display of error messages, defaults and preserving user input, and processing submitted data.

6.1 Useful Server Variables

Aside from `PHP_SELF`, the `$_SERVER` auto-global array contains a number of useful elements that provide information on the web server and the current request. Table Table 6-1 lists some of them.

Table 6-1. Entries in $_SERVER

Element	Example	Description

Table 6-1. Entries in $_SERVER

Element	Example	Description
`QUERY_STRING`	`category=kitchen&price=5`	The part of the URL after the question mark whe live. The example query string shown is for the U *http://www.example.com/catalog/store.php?categ*
`PATH_INFO`	`/browse`	Extra path information tacked onto the end of the This is a way to pass information to a script witho string. The example `PATH_INFO` shown is for the *http://www.example.com/catalog/store.php/brows*
`SERVER_NAME`	*www.example.com*	The name of the web site on which the PHP inter web server hosts many different virtual domains, particular virtual domain that is being accessed.
`DOCUMENT_ROOT`	*/usr/local/htdocs*	The directory on the web server computer that ho available on the web site. If the document root is the web site *http://www.example.com*, then a requ *http://www.example.com/catalog/store.php* corres */usr/local/htdocs/catalog/store.php*.
`REMOTE_ADDR`	*175.56.28.3*	The IP address of the user making the request to
`REMOTE_HOST`	*pool0560.cvx.dialup.verizon.net*	If your web server is configured to translate user hostnames, this is the hostname of the user makin web server. Because this address-to-name transla expensive (in terms of computational time), most it.
`HTTP_REFERER`[2]	http://directory.google.com/Top/Shopping/Clothing/	If someone clicked on a link to reach the current contains the URL of the page that contained the l faked, so don't use it as your sole criteria for givi web pages. It can, however, be useful for finding you.
`HTTP_USER_AGENT`	`Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)`	The web browser that retrieved the page. The exa signature of Internet Explorer 6.0 running on Wi `HTTP_REFERER`, this value can be faked, but is us

[2] The correct spelling is `HTTP_REFERRER`. But it was misspelled in an early Internet specification document, so you frequently see the three-R version when web programming.

6.2 Accessing Form Parameters

At the beginning of every request, the PHP interpreter sets up some auto-global arrays that contain the values of any parameters submitted in a form or passed in the URL. URL and form parameters from GET method forms are put into `$_GET`. Form parameters from POST method forms are put into `$_POST`.

The URL *http://www.example.com/catalog.php?product_id=21&category=fryingpan* puts two values into `$_GET`: `$_GET['product_id']` is set to `21` and `$_GET['category']` is set to `fryingpan`. Submitting the form in Example 6-2 causes the same values to be put into `$_POST`, assuming `21` is entered in the text box and `Frying Pan` is selected from the menu.

Example 6-2. A two-element form

```
<form method="POST" action="catalog.php">
<input type="text" name="product_id">
<select name="category">
<option value="ovenmitt">Pot Holder</option>
<option value="fryingpan">Frying Pan</option>
<option value="torch">Kitchen Torch</option>
</select>
<input type="submit" name="submit">
</form>
```

Example 6-3 incorporates the form in Example 6-2 into a complete PHP program that prints the appropriate values from `$_POST` after displaying the form. Because the `action` attribute of the `<form>` tag in Example 6-3 is `catalog.php`, you need to save the program in a file called `catalog.php` on your web server. If you save it in a file with a different name, adjust the `action` attribute accordingly.

Example 6-3. Printing submitted form parameters

```
<form method="POST" action="catalog.php">
<input type="text" name="product_id">
<select name="category">
<option value="ovenmitt">Pot Holder</option>
<option value="fryingpan">Frying Pan</option>
<option value="torch">Kitchen Torch</option>
</select>
<input type="submit" name="submit">
</form>
Here are the submitted values:

product_id: <?php print $_POST['product_id']; ?>
<br/>
category: <?php print $_POST['category']; ?>
```

A form element that can have multiple values needs to have a name that ends in `[ ]`. This tells the PHP interpreter to treat the multiple values as array elements. The `<select>` menu in Example 6-4 has its submitted values put into `$_POST['lunch']`.

Example 6-4. Multiple-valued form elements

```
<form method="POST" action="eat.php">
<select name="lunch[  ]" multiple>
```

```
<option value="pork">BBQ Pork Bun</option>
<option value="chicken">Chicken Bun</option>
<option value="lotus">Lotus Seed Bun</option>
<option value="bean">Bean Paste Bun</option>
<option value="nest">Bird-Nest Bun</option>
</select>
<input type="submit" name="submit">
</form>
```

If the form in Example 6-4 is submitted with `Chicken Bun` and `Bird-Nest Bun` selected, then `$_POST['lunch']` becomes a two-element array, with element values `chicken` and `nest`. Access these values using the regular multidimensional array syntax. Example 6-5 incorporates the form from Example 6-4 into a complete program that prints out each value selected in the menu. (The same rule applies here to the filename and the `action` attribute. Save the code in Example 6-5 in a file called *eat.php* or adjust the `action` attribute of `the <form>` tag to the correct filename.)

Example 6-5. Accessing multiple submitted values

```
<form method="POST" action="eat.php">
<select name="lunch[  ]" multiple>
<option value="pork">BBQ Pork Bun</option>
<option value="chicken">Chicken Bun</option>
<option value="lotus">Lotus Seed Bun</option>
<option value="bean">Bean Paste Bun</option>
<option value="nest">Bird-Nest Bun</option>
</select>
<input type="submit" name="submit">
</form>
Selected buns:
<br/>
<?php
foreach ($_POST['lunch'] as $choice) {
    print "You want a $choice bun. <br/>";
}
?>
```

With `Chicken Bun` and `Bird-Nest Bun` selected in the menu, Example 6-5 prints (after the form):

```
Selected buns:
You want a chicken bun.
You want a nest bun.
```

You can think of a form element named `lunch[ ]` as translating into the following PHP code when the form is submitted (assuming the submitted values for the form element are `chicken` and `nest`):

```
$_POST['lunch'][  ] = 'chicken';
$_POST['lunch'][  ] = 'nest';
```

As you saw in Example 4-5, this syntax adds an element to the end of an array.

6.3 Form Processing with Functions

The basic form in Example 6-1 can be made more flexible by putting the display code and the processing code in separate functions. Example 6-6 is a version of Example 6-1 with functions.

Example 6-6. Saying "Hello" with functions

```
// Logic to do the right thing based on
// the submitted form parameters
if (array_key_exists('my_name',$_POST) {
    process_form( );
} else {
    show_form( );
}

// Do something when the form is submitted
function process_form( ) {
    print "Hello, ". $_POST['my_name'];
}

// Display the form
function show_form( ) {
    print<<<_HTML_
<form method="POST" action="$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
</form>
_HTML_;
}
```

To change the form or what happens when it's submitted, change the body of `process_form( )` or `show_form( )`. These functions make the code a little cleaner, but the logic at the top still depends on some form-specific information: the `my_name` parameter. We can solve that problem by using a hidden parameter in the form as the test for submission. If the hidden parameter is in `$_POST`, then we process the form. Otherwise, we display it. In Example 6-7, this strategy is shown using a hidden parameter named `_submit_check`.

Example 6-7. Using a hidden parameter to indicate form submission

```
// Logic to do the right thing based on
// the hidden _submit_check parameter
if ($_POST['_submit_check']) {
    process_form( );
} else {
    show_form( );
}

// Do something when the form is submitted
function process_form( ) {
    print "Hello, ". $_POST['my_name'];
}

// Display the form
```

```
function show_form( ) {
    print<<<_HTML_
<form method="POST" action="$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
<input type="hidden" name="_submit_check" value="1">
</form>
_HTML_;
}
```

It's OK in this example to take a shortcut and not use `array_key_exists( )` in the `if( )` statement at the top of the code. The `_submit_check` form parameter can only have one value: 1. You don't have to worry about it being present in `$_POST` but having a value that evaluates to `false`.

In addition to making the main logic of the page independent of any changing form elements, using a hidden parameter as a submission test also ensures that the form is processed when a user clicks "Enter" in their browser to submit it instead of clicking the submit button. When a form is submitted with "Enter," some browsers don't send the name and value of the submit button as part of the submitted form data. A hidden parameter, however, is always included.

Breaking up the form processing and display into functions also makes it easy to add a data validation stage. Data validation, covered in detail in Section 6.4, is an essential part of any web application that accepts input from a form. Data should be validated after a form is submitted, but before it is processed. Example 6-8 adds a validation function to Example 6-7.

Example 6-8. Validating form data

```
// Logic to do the right thing based on
// the hidden _submit_check parameter
if ($_POST['_submit_check']) {
    if (validate_form( )) {
        process_form( );
    } else {
        show_form( );
    }
} else {
    show_form( );
}

// Do something when the form is submitted
function process_form( ) {
    print "Hello, ". $_POST['my_name'];
}

// Display the form
function show_form( ) {
    print<<<_HTML_
<form method="POST" action="$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
<input type="hidden" name="_submit_check" value="1">
</form>
```

```
_HTML_;
}

// Check the form data
function validate_form( ) {
    // Is my_name at least 3 characters long?
    if (strlen($_POST['my_name']) < 3) {
        return false;
    } else {
        return true;
    }
}
```

The `validate_form( )` function in Example 6-8 returns `false` if `$_POST['my_name']` is less than three characters long, and returns `true` otherwise. At the top of the page, `validate_form( )` is called when the form is submitted. If it returns `true`, then `process_form( )` is called. Otherwise, `show_form( )` is called. This means that if you submit the form with a name that's at least three characters long, such as `Bob` or `Bartholomew`, the same thing happens as in previous examples: a `Hello, Bob` (or `Hello, Bartholomew`) message is displayed. If you submit a short name such as `BJ` or leave the text box blank, then `validate_form( )` returns `false` and `process_form( )` is never called. Instead `show_form( )` is called and the form is redisplayed.

Example 6-8 doesn't tell you what's wrong if you enter a name that doesn't pass the test in `validate_form( )`. Ideally, when someone submits data that fails a validation test, you should explain the error when you redisplay the form and, if appropriate, redisplay the value he entered inside the appropriate form element. Section 6.4 shows you how to display error messages, and Section 6.5 explains how to safely redisplay user-entered values.

6.4 Validating Data

Some of the validation strategies discussed in this section use *regular expressions*, which are powerful text-matching patterns, written in a language all their own. If you're not familiar with regular expressions, Appendix B provides a quick introduction.

Data validation is one of the most important parts of a web application. Weird, wrong, and damaging data shows up where you least expect it. Users are careless, users are malicious, and users are fabulously more creative (often accidentally) than you may ever imagine when you are designing your application. Without a *Clockwork Orange*-style forced viewing of a filmstrip on the dangers of unvalidated data, I can't over-emphasize how crucial it is that you stringently validate any piece of data coming into your application from an external source. Some of these external sources are obvious: most of the input to your application is probably coming from a web form. But there are lots of other ways data can flow into your programs as well: databases that you share with other people or applications, web services and remote servers, even URLs and their parameters.

As mentioned earlier, Example 6-8 doesn't indicate what's wrong with the form if the check in `validate_form( )` fails. Example 6-9 alters `validate_form( )` and `show_form( )` to manipulate and print an array of possible error messages.

Example 6-9. Displaying error messages with the form

```
// Logic to do the right thing based on
// the hidden _submit_check parameter
if ($_POST['_submit_check']) {
    // If validate_form( ) returns errors, pass them to show_form( )
    if ($form_errors = validate_form( )) {
        show_form($form_errors);
    } else {
        process_form( );
    }
} else {
    show_form( );
}

// Do something when the form is submitted
function process_form( ) {
    print "Hello, ". $_POST['my_name'];
}

// Display the form
function show_form($errors = '') {
    // If some errors were passed in, print them out
    if ($errors) {
        print 'Please correct these errors: <ul><li>';
        print implode('</li><li>', $errors);
        print '</li></ul>';
    }

    print<<<_HTML_
<form method="POST" action="$_SERVER[PHP_SELF]">
Your name: <input type="text" name="my_name">
<br/>
<input type="submit" value="Say Hello">
<input type="hidden" name="_submit_check" value="1">
</form>
_HTML_;
}

// Check the form data
function validate_form( ) {
    // Start with an empty array of error messages
    $errors = array( );

    // Add an error message if the name is too short
    if (strlen($_POST['my_name']) < 3) {
        $errors[  ] = 'Your name must be at least 3 letters long.';
    }

    // Return the (possibly empty) array of error messages
    return $errors;
}
```

The code in Example 6-9 takes advantage of the fact that an empty array evaluates to `false`. The line `if ($form_errors = validate_form( ))` decides whether to call `show_form( )` again and pass it the error array or to call `process_form(`

). The array that validate_form( ) returns is assigned to $form_errors. The truth value of the if( ) test expression is the result of that assignment, which, as you saw in Chapter 3 in Section 3.1, is the value being assigned. So, the if( ) test expression is true if $form_errors has some elements in it, and false if $form_errors is empty. If validate_form( ) encounters no errors, then the array it returns is empty.

It is a good idea to do validation checks on all of the form elements in one pass, instead of redisplaying the form immediately when you find a single element that isn't valid. A user should find out all of his errors when he submits a form instead of having to submit a form over and over again, with a new error message revealed on each submission. The validate_form( ) function in Example 6-9 does this by adding an element to $errors for each problem with a form element. Then, show_form( ) prints out a list of the error messages.

The validation methods shown here all go inside the validate_form( ) function. If a form element doesn't pass the test, then a message is added to the $errors array.

6.4.1 Required Elements

To make sure something has been entered into a required element, check the element's length with strlen( ), as in Example 6-10.

Example 6-10. Verifying a required element

```
if (strlen($_POST['email']) =  = 0) {
   $errors[  ] = "You must enter an email address.";
}
```

It is important to use strlen( ) when checking a required element instead of testing the value itself in an if( ) statement. A test such as if (! $_POST['quantity']) treats a value that evaluates to false as an error. Using strlen( ) lets users enter a value such as 0 into a required element.

6.4.2 Numeric or String Elements

To ensure that a submitted value is an integer or floating-point number, use the conversion functions intval( ) and floatval( ). They give you the number (integer or floating point) inside a string, discarding any extraneous text or alternative number formats.

To use these functions for form validation, compare a submitted form value with what you get when you pass the submitted form value through intval( ) or floatval( ) and then through strval( ). The strval( ) function converts the cleaned-up number back into a string so that the comparison with the element of $_POST works properly. If the submitted string and the cleaned-up string don't match, then there is some funny business in the submitted value and you should reject it. Example 6-11 shows how to check whether a submitted form element is an integer.

Example 6-11. Checking for an integer

```
if ($_POST['age'] != strval(intval($_POST['age'])) {
    $errors[  ] = 'Please enter a valid age.';
}
```

If `$_POST['age']` is an integer such as 59, 0, or -32, then `intval($_POST['age'])` returns, respectively, 59, 0, or -32. The two values match and nothing is added to `$errors`. But if `$_POST['age']` is `52-pickup`, then `intval($_POST['age'])` is 52. These two values aren't equal, so the `if( )` test expression succeeds and a message is added to `$errors`. If `$_POST['age']` contains no numerals at all, then `intval($_POST['age'])` returns 0. For example, if `old` is submitted for `$_POST['age']`, then `intval($_POST['age'])` returns 0.

Similarly, Example 6-12 shows how to use `floatval( )` and `strval( )` to check that a submitted value is a floating-point or decimal number.

Example 6-12. Checking for a floating-point number

```
if ($_POST['price'] != strval(floatval($_POST['price']))) {
    $errors[  ] = 'Please enter a valid price.';
}
```

The `floatval( )` function works like `intval( )`, but it understands a decimal point. In Example 6-12, if `$_POST['price']` contains a valid floating-point number or integer (such as 59.2, 12, or -23.2), then `floatval($_POST['price'])` is equal to `$_POST['price']`, and nothing is added to `$errors`. But letters and other junk in `$_POST['price']` trigger an error message.

When validating elements (particularly string elements), it is often helpful to remove leading and trailing whitespace with the `trim( )` function. You can combine this with the `strlen( )` test for required elements to disallow an entry of just space characters. The combination of `trim( )` and `strlen( )` is shown in Example 6-13.

Example 6-13. Combining trim( ) and strlen( )

```
if (strlen(trim($_POST['name'])) =  = 0) {
    $errors[  ] = "Your name is required.";
}
```

If you want to use the whitespace-trimmed value subsequently in your program, alter the value in `$_POST` and the test the altered value, as in Example 6-14.

Example 6-14. Changing a value in $_POST

```
$_POST['name'] = trim($_POST['name']);

if (strlen($_POST['name']) =  = 0) {
    $errors[  ] = "Your name is required.";
}
```

Because `$_POST` is auto-global, a change to one of its elements inside the `validate_form( )` function persists to other uses of `$_POST` after the change in another function, such as `process_form( )`.

6.4.3 Number Ranges

To check whether a number falls within a certain range, first make sure the input is a number. Then, use an `if( )` statement to test the value of the input, as shown in Example 6-15.

Example 6-15. Checking for a number range

```
if ($_POST['age'] != strval(intval($_POST['age']))) {
    $errors[  ] = "Your age must be a number.";
} elseif (($_POST['age'] < 18) || ($_POST['age'] > 65)) {
    $errors[  ] = "Your age must be at least 18 and no more than 65.";
}
```

To test a date range, convert the submitted date value into an epoch timestamp and then check that the timestamp is appropriate. (For more information on epoch timestamps and the `strtotime( )` function used in Example 6-16, see Chapter 9.) Because epoch timestamps are integers, you don't have to do anything special when using a range that spans a month or year boundary. Example 6-16 checks to see whether a supplied date is less than six months old.

Example 6-16. Checking a date range

```
// Get the epoch timestamp for 6 months ago
$range_start = strtotime('6 months ago');
// Get the epoch timestamp for right now
$range_end   = time( );

// 4-digit year is in $_POST['yr']
// 2-digit month is in $_POST['mo']
// 2-digit day is is $_POST['dy']
$submitted_date = strtotime($_POST['yr'] . '-' .
                            $_POST['mo'] . '-' .
                            $_POST['dy']);

if (($range_start > $submitted_date) || ($range_end < $submitted_date)) {
    $errors[  ] = 'Please choose a date less than six months old.';
}
```

6.4.4 Email Addresses

Checking an email address is arguably the most common form validation task. There is, however, no perfect one-step way to make sure an email address is valid, since "valid" could mean different things depending on your goal. If you truly want to make sure that someone providing you an email address is giving you a working address, and that the person providing it controls that address, you need to do two things. First, when the email address is submitted, send a message containing a random string to that address. In the message, tell the user to submit the random string in a form on your site. Or, include a URL in the message that the user can just click on, which has the code embedded into it. If the code is submitted (or the URL is clicked on), then you know that the person who received the message and controls the email address submitted it to your site (or at least is aware of and approves of the submission).

If you don't want to go to all the trouble of verifying the email address with a separate message, there are still some syntax checks you can do in your form validation code to weed out mistyped addresses. The regular expression `^[^@\s]+@([-a-`

`z0-9]+\.)+[a-z]{2,}$` matches most common email addresses and fails to match common mistypings of addresses. Use it with `preg_match( )` as shown in Example 6-17.

Example 6-17. Checking the syntax of an email address

```
if (! preg_match('/^[^@\s]+@([-a-z0-9]+\.)+[a-z]{2,}$/i',
                 $_POST['email'])) {
    $errors[  ] = 'Please enter a valid e-mail address';
}
```

The one danger with this regular expression is that it doesn't allow any whitespace in the username part of the email address (before the `@`). An address such as "`Marles Pickens`"`@sludge.example.com` is valid according to the standard that defines Internet email addresses, but it won't pass this test because of the space character in it. Fortunately, addresses with embedded whitespace are rare enough that you shouldn't run into any problems with it.

6.4.5 <select> Menus

When you use a `<select>` menu in a form, you need to ensure that the submitted value for the menu element is one of the permitted choices in the menu. Although a user can't submit an off-menu value using a mainstream, well-behaved browser such as Mozilla or Internet Explorer, an attacker can construct a request containing any arbitrary value without using a browser.

To simplify display and validation of `<select>` menus, put the menu choices in an array. Then, iterate through that array to display the `<select>` menu inside the `show_form( )` function. Use the same array in `validate_form( )` to check the submitted value. Example 6-18 shows how to display a `<select>` menu with this technique.

Example 6-18. Displaying a <select> menu

```
$sweets = array('Sesame Seed Puff','Coconut Milk Gelatin Square',
                'Brown Sugar Cake','Sweet Rice and Meat');

// Display the form
function show_form( ) {
    print<<<_HTML_
<form method="post" action="$_SERVER[PHP_SELF]">
Your Order: <select name="order">

_HTML_;
foreach ($GLOBALS['sweets'] as $choice) {
    print "<option>$choice</option>\n";
}
print<<<_HTML_
</select>
<br/>
<input type="submit" value="Order">
<input type="hidden" name="_submit_check" value="1">
</form>
_HTML_;
}
```

The HTML that `show_form( )` in Example 6-18 prints is:

```
<form method="post" action="order.php">
Your Order: <select name="order">
<option>Sesame Seed Puff</option>
<option>Coconut Milk Gelatin Square</option>
<option>Brown Sugar Cake</option>
<option>Sweet Rice and Meat</option>
</select>
<br/>
<input type="submit" value="Order">
<input type="hidden" name="_submit_check" value="1">
</form>
```

Inside `validate_form( )`, use the array of `<select>` menu options like this:

```
if (! in_array($_POST['order'], $GLOBALS['sweets'])) {
    $errors[  ] = 'Please choose a valid order.';
}
```

If you want a `<select>` menu with different displayed choices and option values, you need to use a more complicated array. Each array element key is a value attribute for one option. The corresponding array element value is the displayed choice for that option. In Example 6-19, the option values are `puff`, `square`, `cake`, and `ricemeat`. The displayed choices are `Sesame Seed Puff`, `Coconut Milk Gelatin Square`, `Brown Sugar Cake`, and `Sweet Rice and Meat`.

Example 6-19. A <select> menu with different choices and values

```
$sweets = array('puff' => 'Sesame Seed Puff',
                'square' => 'Coconut Milk Gelatin Square',
                'cake' => 'Brown Sugar Cake',
                'ricemeat' => 'Sweet Rice and Meat');

// Display the form
function show_form( ) {
    print<<<_HTML_
<form method="post" action="$_SERVER[PHP_SELF]">
Your Order: <select name="order">

_HTML_;
// $val is the option value, $choice is what's displayed
foreach ($GLOBALS['sweets'] as $val => $choice) {
    print "<option value=\"$val\">$choice</option>\n";
}
print<<<_HTML_
</select>
<br/>
<input type="submit" value="Order">
<input type="hidden" name="_submit_check" value="1">
</form>
_HTML_;
}
```

The form displayed by Example 6-19 is as follows:

```
<form method="post" action="order.php">
Your Order: <select name="order">
<option value="puff">Sesame Seed Puff</option>
<option value="square">Coconut Milk Gelatin Square</option>
<option value="cake">Brown Sugar Cake</option>
<option value="ricemeat">Sweet Rice and Meat</option>
</select>
<br/>
<input type="submit" value="Order">
<input type="hidden" name="_submit_check" value="1">
</form>
```

The submitted value for the `<select>` menu in Example 6-19 should be `puff`, `square`, `cake`, or `ricemeat`. Example 6-20 shows how to verify this in `validate_form( )`.

Example 6-20. Checking a <select> menu submission value

```
if (! array_key_exists($_POST['order'], $GLOBALS['sweets'])) {
    $errors[  ] = 'Please choose a valid order.';
}
```

6.4.6 HTML and JavaScript

Submitted form data that contains HTML or JavaScript can cause big problems. Consider a simple "guestbook" application that lets users submit comments on a web page and then displays a list of those comments. If users behave nicely and enter only comments containing plain text, the guestbook remains benign. One user submits `Cool page! I like how you list the different ways to cook fish`. When you come along to browse the guestbook, that's what you see.

The situation is more complicated when the guestbook submissions are not just plain text. If an enthusiastic user submits `This page <b>rules!!!!</b>` as a comment, and it is redisplayed verbatim by the guestbook application, then you see `rules!!!!` in bold when you browse the guestbook. Your web browser can't tell the difference between HTML tags that come from the guestbook application itself (perhaps laying out the comments in a table or a list) and HTML tags that happen to be embedded in the comments that the guestbook is printing.

Although seeing bold text instead of plain text is a minor annoyance, displaying unfiltered user input leaves the guestbook open to giving you a much larger headache. Instead of `<b></b>` tags, one user's submission could contain a malformed or unclosed tag (such as `<a href="` with no ending " or `>`) that prevents your browser from displaying the page properly. Even worse, that submission could contain JavaScript code that, when executed by your web browser as you look at the guestbook, does nasty stuff such as send a copy of your cookies to a stranger's email box or surreptitiously redirect you to another web page.

The guestbook acts as a facilitator, letting a malicious user upload some HTML or JavaScript that is later run by an unwitting user's browser. This kind of problem is called a *cross-site scripting attack* because the poorly written guestbook allows code from one source (the malicious user) to masquerade as coming from another place (the guestbook site.)

To prevent cross-site scripting attacks in your programs, never display unmodified external input. Either remove suspicious parts (such as HTML tags) or encode special characters so that browsers don't act on embedded HTML or JavaScript. PHP gives you two functions that make these tasks simple. The `strip_tags( )` function removes HTML tags from a string, and the `htmlentities( )` function encodes special HTML characters.

Example 6-21 demonstrates `strip_tags( )`.

Example 6-21. Stripping HTML tags from a string

```
// Remove HTML from comments
$comments = strip_tags($_POST['comments']);
// Now it's OK to print $comments
print $comments;
```

If `$_POST['comments']` contains `I <b>love</b> sweet <div class="fancy">rice</div> & tea.`, then Example 6-21 prints:

```
I love sweet rice & tea.
```

All HTML tags and their attributes are removed, but the plain text between the tags is left intact.

Example 6-22 demonstrates `htmlentities( )`.

Example 6-22. Encoding HTML entities in a string

```
$comments = htmlentities($_POST['comments']);
// Now it's OK to print $comments
print $comments;
```

If `$_POST['comments']` contains `I <b>love</b> sweet <div class="fancy">rice</div> & tea.`, then Example 6-22 prints:

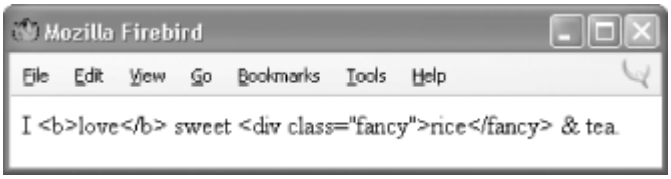
```
I &lt;b&gt;love&lt;/b&gt; sweet &lt;div class=&quot;fancy&quot;&gt;rice&lt;/fancy&gt;
&amp; tea.
```

The characters that have a special meanings in HTML (`<`, `>`, `&`, and ") have been changed into their entity equivalents:

- `<` to `<`
- `>` to `>`
- `&` to `&`
- " to `&quot;`

When a browser sees `<`, it prints out a `<` character instead of thinking "OK, here comes an HTML tag." This is the same idea (but with a different syntax) as escaping a " or `$` character inside a double-quoted string, as you saw earlier in Chapter 2 in Section 2.1. Figure 6-4 shows what the output of Example 6-22 looks like in a web browser.

Figure 6-4. Displaying entity-encoded text



In most applications, you should use `htmlentities( )` to sanitize external input. This function doesn't throw away any content, and it also protects against cross-site scripting attacks. A discussion board where users post messages, for example, about HTML ("What does the `<div>` tag do?") or algebra ("If x<y, is 2x>z?") wouldn't be very useful if those posts were run through `strip_tags( )`. The questions would be printed as "What does the tag do?" and "If xz?".

6.4.7 Beyond Syntax

Most of the validation strategies discussed in this chapter so far check the syntax of a submitted value. They make sure that what's submitted matches a certain format. However, sometimes you want to make sure that a submitted value has not just the correct syntax, but an acceptable meaning as well. The `<select>` menu validation does this. Instead of just assuring that the submitted value is a string, it matches against a specific array of values. The confirmation-message strategy for checking email messages is another example of checking for more than syntax. If you ensure only that a submitted email address has the correct form, a mischievous user can provide an address such as `president@whitehouse.gov` that almost certainly doesn't belong to her. The confirmation message makes sure that the meaning of the address—i.e., "this email address belongs to the user providing it"—is correct.

6.5 Displaying Default Values

Sometimes, you want to display a form with a value already in a text box or with selected checkboxes, radio buttons, or `<select>` menu items. Additionally, when you redisplay a form because of an error, it is helpful to preserve any information that a user has already entered. Example 6-23 shows the code to do this. It belongs at the beginning of `show_form( )` and makes `$defaults` the array of values to use with the form elements.

Example 6-23. Building an array of defaults

```
if ($_POST['_submit_check']) {
    $defaults = $_POST;
} else {
    $defaults = array('delivery'  => 'yes',
                      'size'      => 'medium',
                      'main_dish' => array('taro','tripe'),
                      'sweet'     => 'cake');
}
```

If `$_POST['_submit_check']` is set, that means the form has been submitted. In that case, the defaults should come from whatever the user submitted. If `$_POST['_submit_check']` is not set, then you can set your own defaults. For most form parameters, the default is a string or a number. For form elements that can have more than one value, such as the multivalued `<select>` menu `main_dish`, the default value is an array.

After setting the defaults, provide the appropriate value from `$defaults` when printing out the HTML tag for the form element. Remember to encode the defaults with `htmlentities( )` when necessary in order to prevent cross-site scripting attacks. Because of the structure of the HTML tags, you need to treat text boxes, `<select>` menus, text areas, and checkboxes/radio buttons differently.

For text boxes, set the `value` attribute of the `<input>` tag to the appropriate element of `$defaults`. Example 6-24 shows how to do this.

Example 6-24. Setting a default value in a text box

```
print '<input type="text" name="my_name" value="' .
      htmlentities($defaults['my_name']). '">';
```

For multiline text areas, put the entity-encoded value between the `<textarea>` and `</textarea>` tags, as shown in Example 6-25.

Example 6-25. Setting a default value in a multiline text area

```
print '<textarea name="comments">';
print htmlentities($defaults['comments']);
print '</textarea>';
```

For `<select>` menus, add a check to the loop that prints out the `<option>` tags that prints a `selected="selected"` attribute when appropriate. Example 6-26 contains the code to do this for a single-valued `<select>` menu.

Example 6-26. Setting a default value in a <select> menu

```
$sweets = array('puff' => 'Sesame Seed Puff',
                'square' => 'Coconut Milk Gelatin Square',
                'cake' => 'Brown Sugar Cake',
                'ricemeat' => 'Sweet Rice and Meat');

print '<select name="sweet">';
// $val is the option value, $choice is what's displayed
foreach ($sweets as $option => $label) {
    print '<option value="' .$option .'"';
    if ($option =  = $defaults['sweet']) {
        print ' selected="selected"';
    }
    print "> $label</option>\n";
}
print '</select>';
```

To set defaults for a multivalued `<select>` menu, you need to convert the array of defaults into an associative array in which each key is a choice that should be selected. Then, print the `selected="selected"` attribute for the options found in that associative array. Example 6-27 demonstrates how to do this.

Example 6-27. Setting defaults in a multivalued <select> menu

```
$main_dishes = array('cuke' => 'Braised Sea Cucumber',
```

```
                    'stomach' => "Sauteed Pig's Stomach",
                    'tripe' => 'Sauteed Tripe with Wine Sauce',
                    'taro' => 'Stewed Pork with Taro',
                    'giblets' => 'Baked Giblets with Salt',
                    'abalone' => 'Abalone with Marrow and Duck Feet');

print '<select name="main_dish[  ]" multiple="multiple">';

$selected_options = array( );
foreach ($defaults['main_dish'] as $option) {
    $selected_options[$option] = true;
}

// print out the <option> tags
foreach ($main_dishes as $option => $label) {
    print '<option value="' . htmlentities($option) . '"';
    if ($selected_options[$option]) {
        print ' selected="selected"';
    }
    print '>' . htmlentities($label) . '</option>';

    print "\n";
}
print '</select>';
```

For checkboxes and radio buttons, add a `checked="checked"` attribute to the `<input>` tag. The syntax for checkboxes and radio buttons is identical except for the `type` attribute. Example 6-28 prints a default-aware checkbox named `delivery` and prints three default-aware radio buttons, each named `size` and each with a different value.

Example 6-28. Setting defaults for checkboxes and radio buttons

```
print '<input type="checkbox" name="delivery" value="yes";
if ($defaults['delivery'] =  = 'yes') { print ' checked="checked"'; }
print '> Delivery?';

print '<input type="radio" name="size" value="small";
if ($defaults['size'] =  = 'small') { print ' checked="checked"'; }
print '> Small ';
print '<input type="radio" name="size" value="medium";
if ($defaults['size'] =  = 'medium') { print ' checked="checked"'; }
print '> Medium';
print '<input type="radio" name="size" value="large";
if ($defaults['size'] =  = 'large') { print ' checked="checked"'; }
print '> Large';
```

6.6 Putting It All Together

Turning the humble web form into a feature-packed application with data validation, printing default values, and processing the submitted results might seem like an intimidating task. To ease your burden, this section contains a complete example of a program that does it all:

- Displaying a form, including default values
- Validating the submitted data
- Redisplaying the form with error messages and preserved user input if the submitted data isn't valid
- Processing the submitted data if it is valid

The do-it-all example relies on some helper functions to simplify form element display. These are listed in Example 6-29.

Example 6-29. Form element display helper functions

```
//print a text box
function input_text($element_name, $values) {
    print '<input type="text" name="' . $element_name .'" value="';
    print htmlentities($values[$element_name]) . '">';
}

//print a submit button
function input_submit($element_name, $label) {
    print '<input type="submit" name="' . $element_name .'" value="';
    print htmlentities($label) .'"/>';
}

//print a textarea
function input_textarea($element_name, $values) {
    print '<textarea name="' . $element_name .'">';
    print htmlentities($values[$element_name]) . '</textarea>';
}

//print a radio button or checkbox
function input_radiocheck($type, $element_name, $values, $element_value) {
    print '<input type="' . $type . '" name="' . $element_name .'" value="' . $element_
value . '" ';
    if ($element_value =  = $values[$element_name]) {
        print ' checked="checked"';
    }
    print '/>';
}

//print a <select> menu
function input_select($element_name, $selected, $options, $multiple = false) {
    // print out the <select> tag
    print '<select name="' . $element_name;
    // if multiple choices are permitted, add the multiple attribute
    // and add a [  ] to the end of the tag name
    if ($multiple) { print '[  ]" multiple="multiple'; }
    print '">';

    // set up the list of things to be selected
    $selected_options = array( );
    if ($multiple) {
        foreach ($selected[$element_name] as $val) {
            $selected_options[$val] = true;
        }
    } else {
        $selected_options[ $selected[$element_name] ] = true;
    }
```

```
    // print out the <option> tags
    foreach ($options as $option => $label) {
        print '<option value="' . htmlentities($option) . '"';
        if ($selected_options[$option]) {
            print ' selected="selected"';
        }
        print '>' . htmlentities($label) . '</option>';
    }
    print '</select>';
}
```

Each helper function in Example 6-29 incorporates the appropriate logic discussed in Section 6.5 for a particular kind of form element. Because the form code in Example 6-30 has a number of different elements, it's easier to put the element display code in functions that are called repeatedly than to duplicate the code each time you need to print a particular element.

The `input_text( )` function takes two arguments: the name of the text element and an array of form element values. It prints out an `<input type="text">` tag—a single-line text box. If there is an entry in the form element values array that matches the text element's name, that entry is used for the value attribute of the `<input type="text">` tag. Any special characters in the value are encoded with `htmlentities( )`.

The `input_submit( )` function prints an `<input type="submit">` tag—a submit button. It takes two arguments: the name of the button and the label that should appear on the button.

The `input_textarea( )` function takes two arguments like `input_text( )`: the element name and an array of form element values. Instead of a single-line text box, however, it prints the `<textarea></textarea>` tag pair for a multiline text box. If there is an entry in the form element values array that matches the element name, that entry is used as the default value of the multiline text box. Special characters in the value are encoded with `htmlentities( )`.

Both radio buttons and checkboxes are handled by `input_radiocheck( )`. The first argument to this function is either `radio` (to display a radio button) or `checkbox` (to display a checkbox). This determines whether the function prints an `<input type="radio">` tag or an `<input type="checkbox">` tag. Then comes the element name, the array of form element values, and the value for this particular element. You need to pass both the entire array of form element values and the value for the specific element so the function can see whether the entry in the array for this element matches the passed-in value. For example, Example 6-30 prints three radio buttons named `size`, each with a different value (`small`, `medium`, and `large`). Only one of those radio buttons can have the `checked="checked"` attribute set: the one whose entry in the form element values array matches the passed-in value.

The `input_select( )` function prints `<select>` menus. It requires three arguments: the name of the element, an array of form element values, and an array of options to display in the menu. You can also pass `true` as a fourth argument to allow multiple values to be selected in the menu. The function uses the logic from Examples Example 6-26 and Example 6-27 to build the `$selected_options` array with one entry for each menu choice that should be marked with the `selected="selected"` attribute. Then, it loops through the `$options` array, printing out one `<option></option>` tag pair for each menu choice.

The code in Example 6-30 relies on the form helper functions and displays a short food-ordering form. When the form is submitted correctly, it shows the results in the browser and emails them to an address defined in `process_form( )`

(presumably to the chef, so he can start preparing your order). Because the code jumps in and out of PHP mode, it includes the `<?php` start tag at the beginning of the example and the `?>` closing tag at the end to make things clearer.

Example 6-30. A complete form: display with defaults, validation, and processing

```
<?php
// don't forget to include the code for the form
// helper functions defined in Example 6-29
//
// setup the arrays of choices in the select menus
// these are needed in display_form( ), validate_form( ),
// and process_form( ), so they are declared in the global scope
$sweets = array('puff' => 'Sesame Seed Puff',
                'square' => 'Coconut Milk Gelatin Square',
                'cake' => 'Brown Sugar Cake',
                'ricemeat' => 'Sweet Rice and Meat');

$main_dishes = array('cuke' => 'Braised Sea Cucumber',
                     'stomach' => "Sauteed Pig's Stomach",
                     'tripe' => 'Sauteed Tripe with Wine Sauce',
                     'taro' => 'Stewed Pork with Taro',
                     'giblets' => 'Baked Giblets with Salt',
                     'abalone' => 'Abalone with Marrow and Duck Feet');

// The main page logic:
// - If the form is submitted, validate and then process or redisplay
// - If it's not submitted, display
if ($_POST['_submit_check']) {
    // If validate_form( ) returns errors, pass them to show_form( )
    if ($form_errors = validate_form( )) {
        show_form($form_errors);
    } else {
        // The submitted data is valid, so process it
        process_form( );
    }
} else {
    // The form wasn't submitted, so display
    show_form( );
}

function show_form($errors = '') {
    // If the form is submitted, get defaults from submitted parameters
    if ($_POST['_submit_check']) {
        $defaults = $_POST;
    } else {
        // Otherwise, set our own defaults: medium size and yes to delivery
        $defaults = array('delivery' => 'yes',
                          'size'     => 'medium');
    }

    // If errors were passed in, put them in $error_text (with HTML markup)
    if ($errors) {
        $error_text = '<tr><td>You need to correct the following errors:';
        $error_text .= '</td><td><ul><li>';
        $error_text .= implode('</li><li>',$errors);
        $error_text .= '</li></ul></td></tr>';
    } else {
```

```
        // No errors? Then $error_text is blank
        $error_text = '';
    }

    // Jump out of PHP mode to make displaying all the HTML tags easier
?>
<form method="POST" action="<?php print $_SERVER['PHP_SELF']; ?>">
<table>
<?php print $error_text ?>

<tr><td>Your Name:</td>
<td><?php input_text('name', $defaults) ?></td></tr>

<tr><td>Size:</td>
<td><?php input_radiocheck('radio','size', $defaults, 'small');  ?> Small <br/>
<?php input_radiocheck('radio','size', $defaults, 'medium'); ?> Medium <br/>
<?php input_radiocheck('radio','size', $defaults, 'large');  ?> Large
</td></tr>

<tr><td>Pick one sweet item:</td>
<td><?php input_select('sweet', $defaults, $GLOBALS['sweets']); ?>
</td></tr>

<tr><td>Pick two main dishes:</td>
<td>
<?php input_select('main_dish', $defaults, $GLOBALS['main_dishes'], true) ?>
</td></tr>

<tr><td>Do you want your order delivered?</td>
<td><?php input_radiocheck('checkbox','delivery', $defaults, 'yes'); ?> Yes
</td></tr>

<tr><td>Enter any special instructions.<br/>
If you want your order delivered, put your address here:</td>
<td><?php input_textarea('comments', $defaults); ?></td></tr>

<tr><td colspan="2" align="center"><?php input_submit('save','Order'); ?>
</td></tr>

</table>
<input type="hidden" name="_submit_check" value="1"/>
</form>
<?php
      } // The end of show_form( )

function validate_form( ) {
    $errors = array( );

    // name is required
    if (! strlen(trim($_POST['name']))) {
        $errors[  ] = 'Please enter your name.';
    }
    // size is required
    if (($_POST['size'] != 'small') && ($_POST['size'] != 'medium') &&
        ($_POST['size'] != 'large')) {
        $errors[  ] = 'Please select a size.';
    }
```

```
    // sweet is required
    if (! array_key_exists($_POST['sweet'], $GLOBALS['sweets'])) {
        $errors[  ] = 'Please select a valid sweet item.';
    }
    // exactly two main dishes required
    if (count($_POST['main_dish']) != 2) {
        $errors[  ] = 'Please select exactly two main dishes.';
    } else {
        // We know there are two main dishes selected, so make sure they are
        // both valid
        if (! (array_key_exists($_POST['main_dish'][0], $GLOBALS['main_dishes']) &&
               array_key_exists($_POST['main_dish'][1], $GLOBALS['main_dishes']))) {
            $errors[  ] = 'Please select exactly two valid main dishes.';
        }
    }
    // if delivery is checked, then comments must contain something
    if (($_POST['delivery'] =  = 'yes') && (! strlen(trim($_POST['comments'])))) {
        $errors[  ] = 'Please enter your address for delivery.';
    }

    return $errors;
}

function process_form( ) {
    // look up the full names of the sweet and the main dishes in
    // the $GLOBALS['sweets'] and $GLOBALS['main_dishes'] arrays
    $sweet = $GLOBALS['sweets'][ $_POST['sweet'] ];
    $main_dish_1 = $GLOBALS['main_dishes'][ $_POST['main_dish'][0] ];
    $main_dish_2 = $GLOBALS['main_dishes'][ $_POST['main_dish'][1] ];
    if ($_POST['delivery'] =  = 'yes') {
        $delivery = 'do';
    } else {
        $delivery = 'do not';
    }
    // build up the text of the order message
    $message=<<<_ORDER_
Thank you for your order, $_POST[name].
You requested the $_POST[size] size of $sweet, $main_dish_1, and $main_dish_2.
You $delivery want delivery.
_ORDER_;
    if (strlen(trim($_POST['comments']))) {
        $message .= 'Your comments: '.$_POST['comments'];
    }

    // send the message to the chef
    mail('chef@restaurant.example.com', 'New Order', $message);
    // print the message, but encode any HTML entities
    // and turn newlines into <br/> tags
    print nl2br(htmlentities($message));
}
?>
```

There are four parts to the code in Example 6-30: the code in the global scope at the top of the example, the `show_form( )` function, the `validate_form( )` function, and the `process_form( )` function.

The global scope code does two things. The first is it sets up two arrays that describe the choices in the form's two `<select>` menus. Because these arrays are used by each of the `show_form( )`, `validate_form( )`, and `process_form( )` functions, they need to be defined in the global scope. The global code's other task is to process the `if( )` statement that decides what to do: display, validate, or process the form.

Displaying the form is accomplished by `show_form( )`. First, the function makes `$defaults` an array of default values. If the form has been submitted and is being redisplayed, then the default values come from `$_POST`. Otherwise, they are explicitly set to `yes` for the `delivery` checkbox and `medium` for the `size` radio button. Then, `show_form( )` prints out a list of errors, if any were passed to it. The HTML list of errors is constructed from the `$errors` array using `implode( )` in a similar technique to the one shown in Example 4-21. Next, `show_form( )` jumps out of PHP mode to print the form. Amid the HTML table-formatting tags, it returns to PHP mode repeatedly to call the helper functions that print out the appropriate tags for each form element. The hidden `_submit_check` element at the end of the form is printed without using a helper function.

The `validate_form( )` function builds an array of error messages if the submitted form data doesn't meet appropriate criteria. Note that the checks for `size`, `sweet`, and `main_dish` don't just look to see whether something was submitted for those parameters, but also check whether what was submitted is a valid value for the particular parameter. For `size`, this means that the submitted value must be `small`, `medium`, or `large`. For `sweet` and `main_dish`, this means that the submitted values must be keys in the global `$sweets` or `$main_dishes` arrays. Even though the form contains default values, it's still a good idea to validate the input. Someone trying to break into your web site could bypass a regular web browser and construct a request with an arbitrary value that isn't a legitimate choice for the `<select>` menu or radio button.

Last, `process_form( )` takes action when the form is submitted with valid data. It builds a string, `$message`, that contains a description of the submitted order. Then it emails `$message` to `chef@restaurant.example.com` and prints it. The built-in `mail( )` function sends the email message. (See Section 13.5 for more details on `mail( )`.) Before printing `$message`, `process_form( )` passes it through two functions. The first is `htmlentities( )`, which, as you've already seen, encodes any special characters as HTML entities. The second is `nl2br( )`, which turns any newlines in `$message` into HTML `<br>` tags. Turning newlines into `<br>` tags makes the line breaks in the message display properly in a web browser.

6.7 Chapter Summary

Chapter 6 covers:

- Understanding the conversation between the web browser and web server that displays a form, processes the submitted form parameters, and then displays a result.
- Making the connection between the `<form>` tag's `action` attribute and the URL to which form parameters are submitted.
- Using values from the `$_SERVER` auto-global array.
- Accessing submitted form parameters in the `$_GET` and `$_POST` auto-global arrays.
- Accessing multivalued submitted form parameters.
- Using `show_form( )`, `validate_form( )`, and `process_form( )` functions to modularize form handling.
- Using a hidden form element to check whether a form has been submitted.
- Displaying error messages with a form.
- Validating form elements: required elements, integers, floating-point numbers, strings, date ranges, email addresses, and `<select>` menus.

- Defanging or removing submitted HTML and JavaScript before displaying it.
- Displaying default values for form elements.
- Using helper functions to display form elements.

6.8 Exercises

1. What does `$_POST` look like when the following form is submitted with the third option in the `Braised Noodles` menu selected, the first and last options in the `Sweet` menu selected, and 4 entered into the text box?

```
<form method="POST" action="order.php">
Braised Noodles with: <select name="noodle">
<option>crab meat</option>
<option>mushroom</option>
<option>barbecued pork</option>
<option>shredded ginger and green onion</option>
</select>
<br/>
Sweet: <select name="sweet[  ]" multiple>
<option value="puff"> Sesame Seed Puff
<option value="square"> Coconut Milk Gelatin Square
<option value="cake"> Brown Sugar Cake
<option value="ricemeat"> Sweet Rice and Meat
</select>
<br/>
Sweet Quantity: <input type="text" name="sweet_q">
<br/>
<input type="submit" name="submit" value="Order">
</form>
```

2. Write a `process_form( )` function that prints out all submitted form parameters and their values. You can assume that form parameters have only scalar values.

4. Write a program that does basic arithmetic. Display a form with text box inputs for two operands and a `<select>` menu to choose an operation: addition, subtraction, multiplication, or division. Validate the inputs to make sure that they are numeric and appropriate for the chosen operation. The processing function should display the operands, operator, and the result. For example, if the operands are 4 and 2 and the operation is multiplication, the processing function should display something like "4 * 2 = 8".
5. Write a program that displays, validates, and processes a form for entering information about a package to be shipped. The form should contain inputs for the from and to addresses for the package, dimensions of the package, and weight of the package. The validation should check (at least) that the package weighs no more than 150 pounds and that no dimension of the package is more than 36 inches. You can assume that the addresses entered on the form are both U.S. addresses, but you should check that a valid state and a ZIP Code with valid syntax are entered. The processing function in your program should print out the information about the package in an organized, formatted report.
6. (Optional) Modify your `process_form( )` function from Exercise 6.2 so that it correctly handles submitted form parameters that have array values. Remember, those array values could themselves contain arrays.