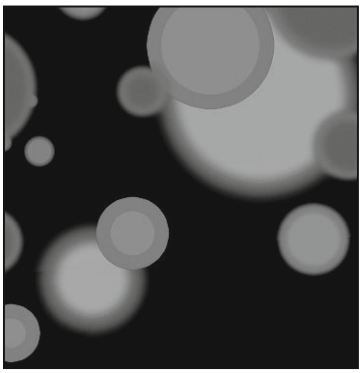


Chapter 5

Bringing Forms to Life

Forms lie at the very heart of working with PHP. You use forms for logging in to restricted pages, registering new users, placing orders with online stores, entering and updating information in a database, sending feedback . . . The list goes on. The same principles lie behind all these uses, so the knowledge you gain from this chapter will have practical value in most PHP applications. To demonstrate how to process information from a form, I'm going to show you how to gather feedback from visitors to your site and send it to your mailbox.

Unfortunately, user input can expose your site to malicious attacks. It's important to always check data submitted from a form before accepting it. Although HTML5 form elements validate user input in the most recent browsers, you still need to check the data on the server. HTML5 validation helps legitimate users avoid submitting a form with errors, but malicious users can easily sidestep checks performed in the browser. Server-side validation is not optional, but essential. The PHP solutions in this chapter show you how to filter out or block anything suspicious or dangerous. It doesn't take a lot of effort to keep marauders at bay. It's also a good idea to preserve user input and redisplay it if the form is incomplete or errors are discovered.

These solutions build a complete mail processing script that can be reused in different forms, so it's important to read them in sequence.

In this chapter, you'll learn about the following:

- Understanding how user input is transmitted from an online form
- Displaying errors without losing user input
- Validating user input and preventing spam with a CAPTCHA
- Sending user input by email

How PHP gathers information from a form

Although HTML contains all the necessary tags to construct a form, it doesn't provide any means to process the form when submitted. For that, you need a server-side solution, such as PHP.

The Japan Journey website contains a simple feedback form (see Figure 5-1). Other elements—such as radio buttons, check boxes, and drop-down menus—will be added later.

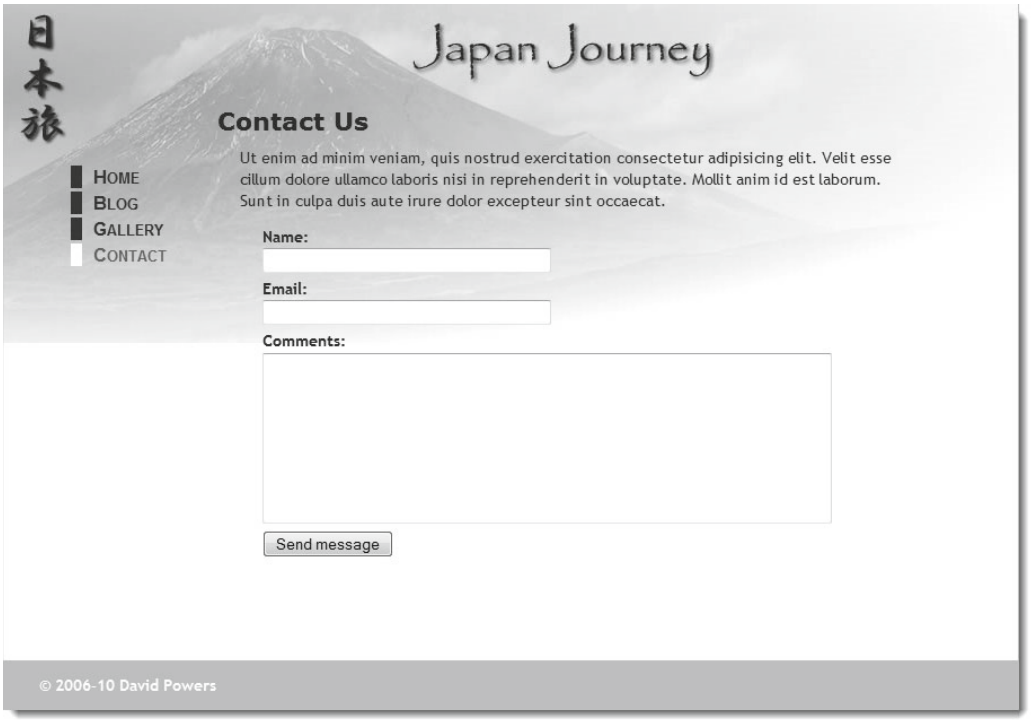


Figure 5-1. Processing a feedback form is one of the most popular uses of PHP.

First, let's take a look at the HTML code for the form (it's in `contact_01.php` in the `ch05` folder):

```
<form id="feedback" method="post" action="">
  <p>
    <label for="name">Name:</label>
    <input name="name" id="name" type="text" class="formbox">
  </p>
  <p>
    <label for="email">Email:</label>
    <input name="email" id="email" type="text" class="formbox">
  </p>
  <p>
    <label for="comments">Comments:</label>
    <textarea name="comments" id="comments" cols="60" rows="8"></textarea>
  </p>
  <p>
     <input name="send" id="send" type="submit" value="Send message">
  </p>
</form>
```

The first thing to notice about this code is that the `<input>` and `<textarea>` tags contain both `name` and `id` attributes set to the same value. The reason for this duplication is that HTML, CSS, and JavaScript all refer to the `id` attribute. Form processing scripts, however, rely on the `name` attribute. So, although the `id` attribute is optional, you *must* use the `name` attribute for each element that you want to be processed.

Two other things to notice are the `method` and `action` attributes inside the opening `<form>` tag. The `method` attribute determines how the form sends data. It can be set to either `post` or `get`. The `action` attribute tells the browser where to send the data for processing when the submit button is clicked. If the value is left empty, as here, the page attempts to process the form itself.

> *I have deliberately avoided using any of the new HTML5 form features, such as `type="email"` and the `required` attribute. This makes it easier to test the PHP server-side validation scripts. After testing, update your forms to use the HTML5 validation features.*

Understanding the difference between post and get

The best way to demonstrate the difference between the `post` and `get` methods is with a real form. If you completed the previous chapter, you can continue working with the same files.

Otherwise, the `ch05` folder contains a complete set of files for the Japan Journey site with all the code from the last chapter incorporated in them. Make sure that the `includes` folder contains `title.inc.php`, `footer.inc.php` and `menu.inc.php`. Copy `contact_01.php` to the site root, and rename it `contact.php`.

1. Locate the opening `<form>` tag in `contact.php`, and change the value of the `method` attribute from `post` to `get` like this:

   ```
   <form id="feedback" method="get" action="">
   ```

2. Save `contact.php`, and load the page in a browser. Type your name, email address, and a short message into the form, and click **Send message**.

3. Look in the browser address bar. You should see the contents of the form attached to the end of the URL like this:

If you break up the URL, it looks like this:

```
http://localhost/phpsols/contact.php
?name=David+Powers
&email=david%40example.com
&comments=I+hope+you+get+this.+%3B-%29
&send=Send+message
```

Each line after the basic URL begins with the `name` attribute of one of the form elements, followed by an equal sign and the contents of the input fields. URLs cannot contain spaces or certain characters (such as my smiley), so the browser encodes them as hexadecimal values, a process known as **URL encoding** (for a full list of values, see `www.w3schools.com/tags/ref_urlencode.asp`).

The first `name` attribute is preceded by a question mark (?) and the others by an ampersand (&). You'll see this type of URL when using search engines, which helps explain why everything after the question mark is known as a **query string**.

4. Go back into the code of `contact.php`, and change `method` back to **post**, like this:

```
<form id="feedback" method="post" action="">
```

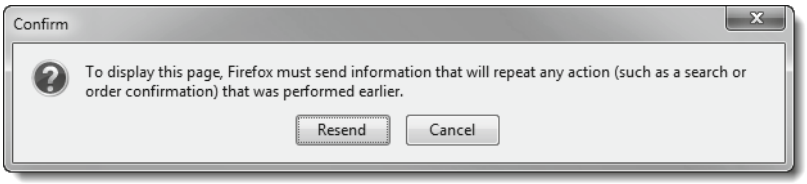
5. Save `contact.php`, and reload the page in your browser. Type another message, and click **Send message**. Your message should disappear, but nothing else happens. So where has it gone? It hasn't been lost, but you haven't done anything to process it yet.

6. In `contact.php`, add the following code immediately below the closing `</form>` tag:

```
<pre>
<?php if ($_POST) { print_r($_POST); } ?>
</pre>
```

This displays the contents of the `$_POST` superglobal array if any `post` data has been sent. As explained in Chapter 3, the `print_r()` function allows you to inspect the contents of arrays; the `<pre>` tags simply make the output easier to read.

7. Save the page, and click the **Refresh** button in your browser. You'll probably see a warning similar to the following. This tells you that the data will be resent, which is exactly what you want. Click **OK** or **Send** depending on your browser.



The code from step 6 should now display the contents of your message below the form as shown in Figure 5-2. Everything has been stored in one of PHP's superglobal arrays, `$_POST`, which contains data sent using the `post` method. The `name` attribute of each form element is used as the array key, making it easy to retrieve the content.

Figure 5-2. The `$_POST` array contains form data with each element identified by its `name` attribute.

As you have just seen, the `get` method sends your data in a very exposed way, making it vulnerable to alteration. Also, Internet Explorer limits the maximum length of a URL to 2,048 characters, so the `get` method can be used only for small amounts of data. The `post` method is more secure and can be used for much larger amounts of data. By default, PHP permits up to 8MB of `post` data, although hosting companies may set a smaller limit.

Consequently, you should normally use the `post` method with forms. The `get` method is used mainly in conjunction with database searches and has the advantage that you can bookmark a search result because all the data is in the URL. We'll return to the `get` method later in the book. This chapter concentrates on the `post` method and its associated superglobal array, `$_POST`.

Although the `post` method is more secure than `get`, you shouldn't assume that it's 100% safe. For secure transmission, you need to use encryption or the Secure Sockets Layer (SSL) with a URL that begins with `https://`.

Keeping safe with PHP superglobals

While I'm on the subject of security, it's worth explaining the background to the PHP superglobal arrays, which include `$_POST` and `$_GET`. The `$_POST` array contains data sent using the `post` method. So it should come as no surprise that data sent by the `get` method is in the `$_GET` array.

In the early days of PHP, you didn't need to use special arrays to access data submitted from a form. If the `name` of the form element was `email`, all that was necessary was to stick a dollar sign on the front, like this: `$email`. Bingo, you had instant access to the data. It was incredibly convenient. Unfortunately, it

also left a gaping security hole. All that an attacker needed to do was view the source of your web page and pass values to your script through a query string.

Occasionally, you'll still see "advice" to turn on `register_globals` in `php.ini` to restore the old way of gathering form data. Turning on `register_globals` is foolish for the following reasons:

- It's totally insecure.
- Most hosting companies now disable `register_globals`. There is no way to override the setting for individual scripts, so any scripts that rely on it won't work.
- The `register_globals` setting will be removed completely from the next major version of PHP. Scripts that rely on `register_globals` won't work with that version, period.

It's very easy to write scripts that don't rely on `register_globals`. It just requires putting the `name` attribute of the form element in quotes between square brackets after `$_POST` or `$_GET`, depending on the form's `method` attribute. So `email` becomes `$_POST['email']` if sent by the `post` method, and `$_GET['email']` if sent by the `get` method. That's all there is to it.

You may come across scripts that use `$_REQUEST`, which avoids the need to distinguish between `$_POST` or `$_GET`. It's less secure. Always use `$_POST` or `$_GET` instead.

Old scripts may use `$HTTP_POST_VARS` or `$HTTP_GET_VARS`, which have the same meaning as `$_POST` and `$_GET`. The old versions don't work on most servers.

Always use `$_POST` and `$_GET` when processing user input from a form.

Removing unwanted backslashes from form input

Some PHP servers automatically insert backslashes in front of quotes when a form is submitted. You should follow the instructions in Chapter 2 to check the value of `magic_quotes_gpc` on your remote server. If it's on, and you can't use `php.ini` or an `.htaccess` file to turn it off, you need to remove these backslashes with the script in `nuke_magic_quotes.php`.

You can ignore PHP Solution 5-1 entirely if `magic_quotes_gpc` is off on your remote server.

PHP Solution 5-1: Using a script to eliminate magic quotes

This PHP solution is the least efficient way of dealing with magic quotes and should be used *only* if you cannot turn off `magic_quotes_gpc` on your remote server by any other means. To reproduce the same conditions as on your remote server, edit your local version of `php.ini` to turn on `magic_quotes_gpc` (Chapter 2 describes how to edit configuration directives in `php.ini`).

Continue working with the file from the previous exercise. Alternatively, use `contact_02.php` from the `ch05` folder. Copy it to the site root and rename it `contact.php`.

1. Load `contact.php` into a browser. Enter some text that contains an apostrophe or some double quotes. Click **Send message**.

2. Check the contents of the `$_POST` array at the bottom of the screen. If magic quotes are on, you will see something like Figure 5-3. A backslash has been inserted in front of all single and double quotes (apostrophes are treated the same as single quotes). If magic quotes are off, you will see no change from your original text.

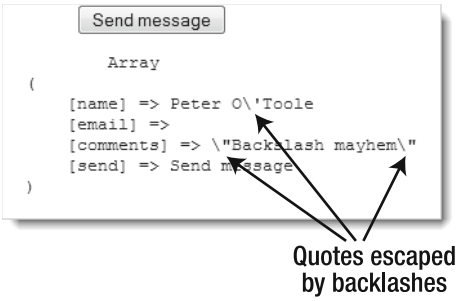


Figure 5-3. PHP magic quotes automatically insert a backslash in front of quotes when a form is submitted.

3. If your remote server uses magic quotes, add the following code shown in bold at the end of the code block at the top of `contact.php`:

```
<?php
include('./includes/title.inc.php');
if ($_POST) {
  include('./includes/nuke_magic_quotes.php');
}
?>
```

4. The conditional statement checks if the `$_POST` array contains any values. If it does, it includes the file `nuke_magic_quotes.php`, which contains a script from the PHP manual at `http://docs.php.net/manual/en/security.magicquotes.disabling.php`. The script removes backslashes from form data and cookies. You should always include this script at the beginning of any page that processes form data.

   In this case, I have wrapped the include command in a conditional statement that checks only the `$_POST` array. Obviously, if the form is submitted using the `get` method, you should check the `$_GET` array. If you're expecting data from multiple sources, you can omit the conditional statement, but it's slightly more efficient to use one, because it avoids running the script in `nuke_magic_quotes.php` unnecessarily.

   The script in `nuke_magic_quotes.php` automatically checks whether `magic_quotes_gpc` is on. If it's off, the form data is not touched, so your pages will continue to work correctly even if your hosting company changes the setting.

5. Save `contact.php`, and click the **Reload** button in your browser. Confirm that you want to resend the `post` data.

6. The `$_POST` array should now be clear of backslashes, as shown in Figure 5-4. You can check your code with `contact_03.php` in the `ch05` folder.

Figure 5-4. The backslashes have been cleaned up from the `$_POST` array.

Since magic quotes are rapidly being phased out of PHP, the remaining PHP solutions and download files assume `magic_quotes_gpc` is off.

Processing and validating user input

The ultimate aim of this chapter is to send the input from the form in `contact.php` by email to your inbox. Using the PHP `mail()` function is relatively simple. It takes a minimum of three arguments: the address(es) the email is being sent to, a string containing the subject line, and a string containing the body of the message. You build the body of the message by concatenating (joining) the contents of the input fields into a single string.

Security measures implemented by most Internet service providers (ISPs) make it difficult—if not impossible—to test the `mail()` function in a local testing environment. Instead of jumping straight into the use of `mail()`, PHP Solutions 5-2 through 5-5 concentrate on validating user input to make sure required fields are filled in and displaying error messages. Implementing these measures makes your online forms more user-friendly and secure.

For many years, web designers have used JavaScript to check user input when the submit button is clicked. That role is being gradually taken over by browsers that support HTML5. This is called **client-side validation** because it happens on the user's computer (or client). It's useful because it's almost instantaneous and can alert the user to a problem without making an unnecessary round trip to the server. However, you should never rely on client-side validation alone because it's too easy to sidestep. All a malicious user has to do is turn off JavaScript in the browser, or submit data from a custom script, and your checks are rendered useless. It's vital to check user input on the server side with PHP, too.

Creating a reusable script

Email processing scripts are usually stored in a separate file that contains generic code capable of handling any form input. Information specific to the form, such as the destination address and subject line, must be added directly to the script or sent to it using hidden form fields. The location of the processing script is stored in the `action` attribute of the `<form>` tag, so the browser knows where to send the input data when the user clicks the submit button.

The ability to reuse the same script—perhaps with only a few edits—for multiple websites is a great timesaver. However, sending the input data to a separate file for processing makes it difficult to alert users to errors without losing their input. To get around this problem, the approach taken in this chapter is to use what's known as a **self-processing form**.

Instead of sending the data to a separate file, the page containing the form is reloaded, and the processing script is wrapped in a PHP conditional statement above the `DOCTYPE` declaration that checks if the form has been submitted. The advantage is that the form can be redisplayed with error messages and preserving the user's input if errors are detected by the server-side validation.

Parts of the script that are specific to the form will be embedded in the PHP code block above the `DOCTYPE` declaration. The generic, reusable parts of the script will be in a separate file that can be included in any page that requires an email processing script.

PHP Solution 5-2: Making sure required fields aren't blank

When required fields are left blank, you don't get the information you need, and the user may never get a reply, particularly if contact details have been omitted.

Continue using the same files. Alternatively, use `contact_02.php` from the `ch05` folder. If your remote server has magic quotes turned on, use `contact_03.php` instead.

1. The processing script uses two arrays called `$errors` and `$missing` to store details of errors and required fields that haven't been filled in. These arrays will be used to control the display of error messages alongside the form labels. There won't be any errors when the page first loads, so initialize `$errors` and `$missing` as empty arrays in the PHP code block at the top of `contact.php` like this:

```
<?php
include('./includes/title.inc.php');
$errors = array();
$missing = array();
?>
```

2. The email processing script should be executed only if the form has been submitted. As Figures 5-2 through 5-4 show, the `$_POST` array contains a name/value pair for the submit button, which is called `send` in `contact.php`. You can test whether the form has been submitted by creating a conditional statement and passing `$_POST['send']` to `isset()`. If `$_POST['send']` has been defined (set), the form has been submitted. Add the code highlighted in bold to the PHP block at the top of the page.

```
<?php
include('./includes/title.inc.php');
$errors = array();
$missing = array();
// check if the form has been submitted
if (isset($_POST['send'])) {
  // email processing script
}
?>
```

Note that `send` is the value of the `name` attribute of the submit button in this form. If you give your submit button a different name, you need to use that name.

If your remote server has `magic_quotes_gpc` turned on, this is where you should include `nuke_magic_quotes.php`:

```
if (isset($_POST['send'])) {
  // email processing script
  include('./includes/nuke_magic_quotes.php');
}
```

You don't need to include `nuke_magic_quotes.php` if your remote server has turned off `magic_quotes_gpc`.

3. Although you won't be sending the email just yet, define two variables to store the destination address and subject line of the email. The following code goes inside the conditional statement that you created in the previous step:

```
if (isset($_POST['send'])) {
  // email processing script
  $to = 'david@example.com'; // use your own email address
  $subject = 'Feedback from Japan Journey';
}
```

4. Next, create two arrays: one listing the `name` attribute of each field in the form and the other listing all *required* fields. For the sake of this demonstration, make the `email` field optional, so that only the `name` and `comments` fields are required. Add the following code inside the conditional block immediately after the code that defines the subject line:

```
  $subject = 'Feedback from Japan Journey';
  // list expected fields
  $expected = array('name', 'email', 'comments');
  // set required fields
  $required = array('name', 'comments');
}
```

Why is the `$expected` array necessary? It's to prevent an attacker from injecting other variables in the `$_POST` array in an attempt to overwrite your default values. By processing only those variables that you expect, your form is much more secure. Any spurious values are ignored.

5. The next section of code is not specific to this form, so it should go in an external file that can be included in any email processing script. Create a new PHP file called `processmail.inc.php` in the `includes` folder. Then include it in `contact.php` immediately after the code you entered in the previous step like this:

```
  $required = array('name', 'comments');
```

```
  require('./includes/processmail.inc.php');
}
```

6. The code in `processmail.inc.php` begins by checking the `$_POST` variables for required fields that have been left blank. Strip any default code inserted by your editor, and add the following to `processmail.inc.php`:

```
<?php
foreach ($_POST as $key => $value) {
  // assign to temporary variable and strip whitespace if not an array
  $temp = is_array($value) ? $value : trim($value);
  // if empty and required, add to $missing array
  if (empty($temp) && in_array($key, $required)) {
    $missing[] = $key;
  } elseif (in_array($key, $expected)) {
    // otherwise, assign to a variable of the same name as $key
    ${$key} = $temp;
  }
}
```

   In simple terms, this `foreach` loop goes through the `$_POST` array, strips out any whitespace from text fields, and assigns its contents to a variable with the same name (so `$_POST['email']` becomes `$email`, and so on). If a required field is left blank, its `name` attribute is added to the `$missing` array.

7. Save `processmail.inc.php`. You'll add more code to it later, but let's turn now to the main body of `contact.php`. You need to display a warning if anything is missing. Add a conditional statement at the top of the page content between the `<h2>` heading and first paragraph like this:

```
<h2>Contact us</h2>
<?php if ($missing || $errors) { ?>
  <p class="warning">Please fix the item(s) indicated.</p>
<?php } ?>
<p>Ut enim ad minim veniam . . . </p>
```

   This checks `$missing` and `$errors`, which you initialized as empty arrays in step 1. PHP treats an empty array as `false`, so the paragraph inside the conditional statement isn't displayed when the page first loads. However, if a required field hasn't been filled in when the form is submitted, its name is added to the `$missing` array. An array with at least one element is treated as `true`. The `||` means "or," so this warning paragraph will be displayed if a required field is left blank or if an error is discovered. (The `$errors` array comes into play in PHP Solution 5-4.)

8. To make sure it works so far, save `contact.php`, and load it normally in a browser (don't click the Refresh button). The warning message is not displayed. Click **Send message** without filling in any of the fields. You should now see the message about missing items, as shown in the following screenshot.

9. To display a suitable message alongside each missing required field, add a PHP code block to display a warning as a `<span>` inside the `<label>` tag like this:

```
<label for="name">Name:
<?php if ($missing && in_array('name', $missing)) { ?>
  <span class="warning">Please enter your name</span>
<?php } ?>
</label>
```

   The first condition checks the `$missing` array. If it's empty, the conditional statement fails, and the `<span>` is never displayed. But if `$missing` contains any values, the `in_array()` function checks if the `$missing` array contains the value `name`. If it does, the `<span>` is displayed as shown in Figure 5-5.

10. Insert similar warnings for the `email` and `comments` fields like this:

```
  <label for="email">Email:
  <?php if ($missing && in_array('email', $missing)) { ?>
    <span class="warning">Please enter your email address</span>
  <?php } ?>
  </label>
  <input name="email" id="email" type="text" class="formbox">
</p>
<p>
  <label for="comments">Comments:
  <?php if ($missing && in_array('comments', $missing)) { ?>
    <span class="warning">Please enter your comments</span>
  <?php } ?>
  </label>
```

   The PHP code is the same except for the value you are looking for in the `$missing` array. It's the same as the `name` attribute for the form element.

11. Save `contact.php`, and test the page again, first by entering nothing into any of the fields. The page should look like Figure 5-5.

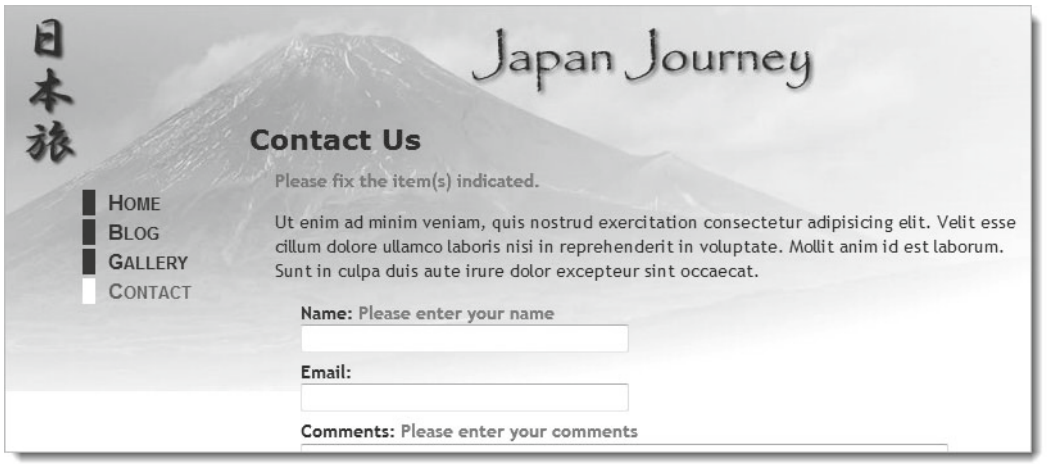


Figure 5-5. By validating user input, you can display warnings about required fields.

Although you added a warning to the `<label>` for the `email` field, it's not displayed, because `email` hasn't been added to the `$required` array. As a result, it's not added to the `$missing` array by the code in `processmail.inc.php`.

12. Add `email` to the `$required` array in the code block at the top of `comments.php` like this:

    ```
    $required = array('name', 'comments', 'email');
    ```

13. Click **Send message** again without filling in any fields. This time, you'll see a warning message alongside each label.

14. Type your name in the **Name** field. In the **Email** and **Comments** fields, just press the spacebar several times. Then click **Send message**. The warning message alongside the **Name** field disappears, but the other two warning messages remain. The code in `processmail.inc.php` strips whitespace from text fields, so it rejects attempts to bypass required fields by entering a series of spaces.

    If you have any problems, compare your code with `contact_04.php` and `processmail.inc_01.php` in the `ch05` folder.

All you need to do to change the required fields is change the names in the `$required` array and add a suitable alert inside the `<label>` tag of the appropriate input element inside the form. It's easy to do, because you always use the `name` attribute of the form input element.

Preserving user input when a form is incomplete

Imagine you have spent ten minutes filling in a form. You click the submit button, and back comes the response that a required field is missing. It's infuriating if you have to fill in every field all over again. Since the content of each field is in the `$_POST` array, it's easy to redisplay it when an error occurs.

PHP Solution 5-3: Creating sticky form fields

This PHP solution shows how to use a conditional statement to extract the user's input from the `$_POST` array and redisplay it in text input fields and text areas.

Continue working with the same files as before. Alternatively, use `contact_04.php` and `processmail.inc_01.php` from the `ch05` folder.

1. When the page first loads, you don't want anything to appear in the input fields. But you do want to redisplay the content if a required field is missing or there's an error. So that's the key: if the `$missing` or `$errors` arrays contain any values, you want the content of each field to be redisplayed. You set default text for a text input field with the value attribute of the `<input>` tag, so amend the `<input>` tag for name like this:

```
<input name="name" id="name" type="text" class="formbox"
<?php if ($missing || $errors) {
  echo 'value="' . htmlentities($name, ENT_COMPAT, 'UTF-8') . '"';
} ?>>
```

   The line inside the curly braces contains a combination of quotes and periods that might confuse you. The first thing to realize is that there's only one semicolon—right at the end—so the `echo` command applies to the whole line. As explained in Chapter 3, a period is called the concatenation operator, which joins strings and variables. You can break down the rest of the line into three sections, as follows:

   - `'value="' .`
   - `htmlentities($name, ENT_COMPAT, 'UTF-8')`
   - `. '"'`

   The first section outputs `value="` as text and uses the concatenation operator to join it to the next section, which passes `$name` to a function called `htmlentities()`. I'll explain what the function does in a moment, but the third section uses the concatenation operator again to join the next section, which consists solely of a double quote. So, if `$missing` or `$errors` contain any values, and `$_POST['name']` contains `Joe`, you'll end up with this inside the `<input>` tag:

```
<input name="name" id="name" type="text" class="formbox" value="Joe">
```

   The `$name` variable contains the original user input, which was transmitted through the `$_POST` array. The `foreach` loop that you created in `processmail.inc.php` in PHP Solution 5-2 processes the `$_POST` array and assigns each element to a variable with the same name. This allows you to access `$_POST['name']` simply as `$name`.

   So, what's the `htmlentities()` function for? As the function name suggests, it converts certain characters to their equivalent HTML entity. The one you're concerned with here is the double quote. Let's say Elvis really is still alive and decides to send feedback through the form. If you use `$name` on its own, Figure 5-6 shows what happens when a required field is omitted and you don't use `htmlentities()`.

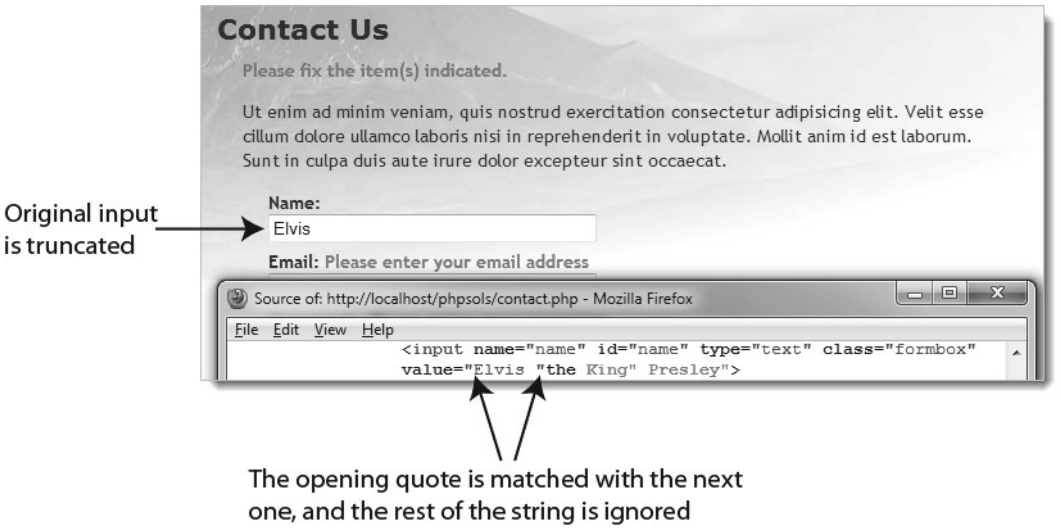


Figure 5-6. Quotes need special treatment before form fields can be redisplayed.

Passing the content of the `$_POST` array element to the `htmlentities()`, however, converts the double quotes in the middle of the string to `&quot;`. And, as Figure 5-7 shows, the content is no longer truncated. What's cool about this is that the HTML entity `&quot;` is converted back to double quotes when the form is resubmitted. As a result, there's no need for any further conversion before the email can be sent.

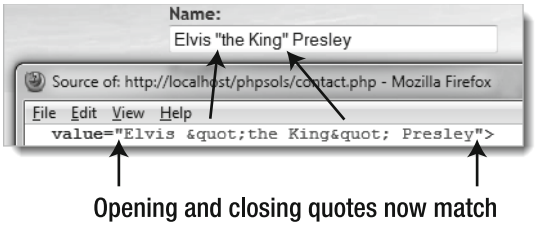


Figure 5-7. The problem is solved by passing the value to `htmlentities()` before it's displayed.

By default, `htmlentities()` uses the Latin1 (ISO-8859-1) character set, which doesn't support accented characters. To support Unicode (UTF-8) encoding, you need to pass three arguments to `htmlentities()`:

- The string you want to convert
- A PHP constant indicating how to handle single quotes (`ENT_COMPAT` leaves them untouched; `ENT_QUOTES` converts them to `&#039;`, the numeric entity for a single straight quote)
- A string containing one of the permitted character sets (encodings) listed at `http://docs.php.net/manual/en/function.htmlentities.php`

2. Edit the `email` field the same way, using `$email` instead of `$name`.

3. The `comments` text area needs to be handled slightly differently because `<textarea>` tags don't have a `value` attribute. You place the PHP block between the opening and closing tags of the text area like this (new code is shown in bold):

```
<textarea name="comments" id="comments" cols="60" rows="8">
  if ($missing || $errors) {
    echo htmlentities($comments, ENT_COMPAT, 'UTF-8');
  } ?></textarea>
```

   It's important to position the opening and closing PHP tags right up against the `<textarea>` tags. If you don't, you'll get unwanted whitespace inside the text area.

4. Save `contact.php`, and test the page in a browser. If any required fields are omitted, the form displays the original content along with any error messages.

   You can check your code with `contact_05.php` in the `ch05` folder.

Using this technique prevents a form reset button from clearing any fields that have been changed by the PHP script. This is a minor inconvenience in comparison with the greater usability offered by preserving existing content when an incomplete form is submitted.

Filtering out potential attacks

A particularly nasty exploit known as **email header injection** seeks to turn online forms into spam relays. A simple way of preventing this is to look for the strings "Content-Type:", "Cc:", and "Bcc:", as these are email headers that the attacker injects into your script to trick it into sending HTML email with copies to many people. If you detect any of these strings in user input, it's a pretty safe bet that you're the target of an attack, so you should block the message. An innocent message may also be blocked, but the advantages of stopping an attack outweigh that small risk.

PHP Solution 5-4: Blocking emails that contain specific phrases

This PHP solution checks the user input for suspect phrases. If one is detected, a Boolean variable is set to `true`. This will be used later to prevent the email from being sent.

Continue working with the same page as before. Alternatively, use `contact_05.php` and `processmail.inc_01.php` from the `ch05` folder.

1. PHP conditional statements rely on a `true/false` test to determine whether to execute a section of code. So the way to filter out suspect phrases is to create a Boolean variable that is switched to `true` as soon as one of those phrases is detected. The detection is done using a search pattern or **regular expression**. Add the following code at the top of `processmail.inc.php` before the existing `foreach` loop:

```
// assume nothing is suspect
$suspect = false;
// create a pattern to locate suspect phrases
$pattern = '/Content-Type:|Bcc:|Cc:/i';

foreach ($_POST as $key => $value) {
```

The string assigned to `$pattern` will be used to perform a case-insensitive search for any of the following: "Content-Type:", "Bcc:", or "Cc:". It's written in a format called Perl-compatible regular expression (PCRE). The search pattern is enclosed in a pair of forward slashes, and the `i` after the final slash makes the pattern case-insensitive.

> *For a basic introduction to regular expressions (regex), see my tutorial in the Adobe Developer Connection at `www.adobe.com/devnet/dreamweaver/articles/regular_expressions_pt1.html`. For a more in-depth treatment,* Regular Expressions Cookbook *by Jan Goyvaerts and Steven Levithan (O'Reilly, 2009, ISBN: 978-0-596-52068-7) is excellent.*

2. You can now use the PCRE stored in `$pattern` to filter out any suspect user input from the `$_POST` array. At the moment, each element of the `$_POST` array contains only a string. However, multiple-choice form elements, such as check box groups, return an array of results. So you need to tunnel down any subarrays and check the content of each element separately. That's precisely what the following custom-built function `isSuspect()` does. Insert it immediately after the `$pattern` variable from step 1.

```
$pattern = '/Content-Type:|Bcc:|Cc:/i';

// function to check for suspect phrases
function isSuspect($val, $pattern, &$suspect) {
  // if the variable is an array, loop through each element
  // and pass it recursively back to the same function
  if (is_array($val)) {
    foreach ($val as $item) {
      isSuspect($item, $pattern, $suspect);
    }
  } else {
    // if one of the suspect phrases is found, set Boolean to true
    if (preg_match($pattern, $val)) {
      $suspect = true;
    }
  }
}

foreach ($_POST as $key => $value) {
```

The `isSuspect()` function is a piece of code that you may want to just copy and paste without delving too deeply into how it works. The important thing to notice is that the third argument has an ampersand (&) in front of it (`&$suspect`). This means that any changes made to the variable passed as the third argument to `isSuspect()` will affect the value of that variable elsewhere in the script.

This technique is known as **passing by reference**. As explained in "Passing values to functions" in Chapter 3, changes to a variable passed as an argument to a function normally have no effect on the variable's value outside the function unless you explicitly return the

value and reassign it to the original variable. They're limited in scope. Prefixing an argument with an ampersand in the function definition overrides this limited scope. When you pass a value by reference, the changes are automatically reflected outside the function. There's no need to return the value and reassign it to the same variable. This technique isn't used very often, but it can be useful in some cases. The ampersand is used only when defining the function. When using the function, you pass arguments in the normal way.

The other feature of this function is that it's what's known as a **recursive function**. It keeps on calling itself until it finds a value that it can compare against the regex.

3. To call the function, pass it the `$_POST` array, the pattern, and the `$suspect` Boolean variable. Insert the following code immediately after the function definition:

```
// check the $_POST array and any subarrays for suspect content
isSuspect($_POST, $pattern, $suspect);
```

Note that you don't put an ampersand in front of `$suspect` this time. The ampersand is required only when you define the function in step 2, not when you call it.

4. If suspect phrases are detected, the value of `$suspect` changes to `true`. There's also no point in processing the `$_POST` array any further. Wrap the code that processes the `$_POST` variables in a conditional statement like this:

```
if (!$suspect) {
  foreach ($_POST as $key => $value) {
    // assign to temporary variable and strip whitespace if not an array
    $temp = is_array($value) ? $value : trim($value);
    // if empty and required, add to $missing array
    if (empty($temp) && in_array($key, $required)) {
      $missing[] = $key;
    } elseif (in_array($key, $expected)) {
      // otherwise, assign to a variable of the same name as $key
      ${$key} = $temp;
    }
  }
}
```

This processes the variables in the `$_POST` array only if `$suspect` is not `true`.

Don't forget the extra curly brace to close the conditional statement.

5. Add a new warning message at the top of page in `contact.php` like this:

```
<?php if ($_POST && $suspect) { ?>
  <p class="warning">Sorry, your mail could not be sent. Please try later.</p>
<?php } elseif ($missing || $errors) { ?>
  <p class="warning">Please fix the item(s) indicated.</p>
<?php } ?>
```

This sets a new condition that takes priority over the original warning message by being considered first. It checks if the `$_POST` array contains any elements—in other words, the form has been submitted—and if `$suspect` is `true`. The warning is deliberately neutral in tone. There's no point in provoking attackers. More important, it avoids offending anyone who may have innocently used a suspect phrase.

6. Save `contact.php`, and test the form by typing one of the suspect phrases in one of the fields. You should see the second warning message, but your input won't be preserved.

   You can check your code against `contact_06.php` and `processmail.inc_02.php` in the `ch05` folder.

Sending email

Before proceeding any further, it's necessary to explain how the PHP `mail()` function works, because it will help you understand the rest of the processing script.

The PHP `mail()` function takes up to five arguments, all of them strings, as follows:

- The address(es) of the recipient(s)
- The subject line
- The message body
- A list of other email headers (optional)
- Additional parameters (optional)

Email addresses in the first argument can be in either of the following formats:

```
'user@example.com'
'Some Guy <user2@example.com>'
```

To send to more than one address, use a comma-separated string like this:

```
'user@example.com, another@example.com, Some Guy <user2@example.com>'
```

The message body must be presented as a single string. This means that you need to extract the input data from the `$_POST` array and format the message, adding labels to identify each field. By default, the `mail()` function supports only plain text. New lines must use both a carriage return and newline character. It's also recommended to restrict the length of lines to no more than 78 characters. Although it sounds complicated, you can build the message body automatically with about 20 lines of PHP code, as you'll see in PHP Solution 5-6.

Adding other email headers is covered in detail in the next section.

Many hosting companies now make the fifth argument a requirement. It ensures that the email is sent by a trusted user, and it normally consists of your own email address prefixed by `-f` (without a space in between), all enclosed in quotes. Check your hosting company's instructions to see whether this is required and the exact format it should take.

Using additional email headers safely

You can find a full list of email headers at `www.faqs.org/rfcs/rfc2076`, but some of the most well-known and useful ones enable you to send copies of an email to other addresses (Cc and Bcc), or to change the encoding. Each new header, except the final one, must be on a separate line terminated by a carriage return and new line character. This means using the `\r` and `\n` escape sequences in double-quoted strings (see Table 3-4 in Chapter 3).

By default, `mail()` uses Latin1 (ISO-8859-1) encoding, which doesn't support accented characters. Web page editors these days frequently use Unicode (UTF-8), which supports most written languages, including the accents commonly used in European languages, as well as nonalphabetic scripts, such as Chinese and Japanese. To ensure that email messages aren't garbled, use the `Content-Type` header to set the encoding to UTF-8 like this:

```
$headers = "Content-Type: text/plain; charset=utf-8\r\n";
```

You also need to add UTF-8 as the `charset` attribute in a `<meta>` tag in the `<head>` of your web pages like this in HTML5:

```
<meta charset=utf-8">
```

In HTML 4.01, use this:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Let's say you also want to send copies of messages to other departments, plus a copy to another address that you don't want the others to see. Email sent by `mail()` is often identified as coming from `nobody@yourdomain` (or whatever username is assigned to the web server), so it's a good idea to add a more user-friendly "From" address. This is how you build those additional headers, using the combined concatenation operator (`.=`) to add each one to the existing variable:

```
$headers .= "From: Japan Journey<feedback@example.com>\r\n";
$headers .= "Cc: sales@example.com, finance@example.com\r\n";
$headers .= 'Bcc: secretplanning@example.com';
```

After building the set of headers you want to use, you pass the variable containing them as the fourth argument to `mail()` like this (assuming that the destination address, subject, and message body have already been stored in variables):

```
$mailSent = mail($to, $subject, $message, $headers);
```

Hard-coded additional headers like this present no security risk, but anything that comes from user input must be filtered before it's used. The biggest danger comes from a text field that asks for the user's email address. A widely used technique is to incorporate the user's email address into a `Reply-To` header, which enables you to reply directly to incoming messages by clicking the **Reply** button in your email program. It's very convenient, but attackers frequently try to pack an email input field with a large number of spurious headers.

Although email fields are the prime target for attackers, the destination address and subject line are both vulnerable if you let users change the value. User input should always be regarded as suspect. PHP Solution 5-4 performs only a basic test for suspect phrases. Before using external input directly in a header you need to apply a more rigorous test.

PHP Solution 5-5: Adding headers and automating the reply address

This PHP solution adds three headers to the email: `From`, `Content-Type` (to set the encoding to UTF-8), and `Reply-To`. Before adding the user's email address to the final header, it uses one of the filter functions introduced in PHP 5.2 to verify that the submitted value conforms to the format of a valid email address.

Continue working with the same page as before. Alternatively, use `contact_06.php` and `processmail.inc_02.php` from the `ch05` folder.

1. Headers are often specific to a particular website or page, so the `From` and `Content-Type` headers will be added to the script in `contact.php`. Add the following code to the PHP block at the top of the page just before `processmail.inc.php` is included:

```
$required = array('name', 'comments', 'email');
// create additional headers
$headers = "From: Japan Journey<feedback@example.com>\r\n";
$headers .= 'Content-Type: text/plain; charset=utf-8';
require('./includes/processmail.inc.php');
```

   The `\r\n` at the end of the `From` header is an escape sequence that inserts a carriage return and newline character, so the string must be in double quotes. At the moment, `Content-Type` is the final header, so it isn't followed by a carriage return and newline character, and the string is in single quotes.

2. The purpose of validating the email address is to make sure it's in a valid format, but the field might be empty because you decide not to make it required or because the user simply ignored it. If the field is required but empty, it will be added to the `$missing` array, and the warning you added in PHP Solution 5-2 will be displayed. If the field isn't empty, but the input is invalid, you need to display a different message.

   Switch to `processmail.inc.php`, and add this code at the bottom of the script:

```
// validate the user's email
if (!$suspect && !empty($email)) {
  $validemail = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
  if ($validemail) {
    $headers .= "\r\nReply-To: $validemail";
  } else {
    $errors['email'] = true;
  }
}
```

   This begins by checking that no suspect phrases have been found and that the `email` field isn't empty. Both conditions are preceded by the logical Not operator (`!`), so they return `true` if `$suspect` and `empty($email)` are both `false`. The `foreach` loop you added in PHP Solution 5-2 assigns all expected elements in the `$_POST` array to simpler variables, so `$email` contains the same value as `$_POST['email']`.

   The next line uses `filter_input()` to validate the email address. The first argument is a PHP constant, `INPUT_POST`, which specifies that the value must be in the `$_POST` array. The

second argument is the name of the element you want to test. The final argument is another PHP constant that specifies you want to check the element conforms to the valid format for an email.

The `filter_input()` function returns the value being tested if it's valid. Otherwise, it returns `false`. So, if the value submitted by the user looks like a valid email address, `$validemail` contains the address. If the format is invalid, `$validemail` is `false`. The `FILTER_VALIDATE_EMAIL` constant accepts only a single email address, so any attempt to insert multiple email addresses will be rejected.

`FILTER_VALIDATE_EMAIL` checks only the format. It doesn't check that the address is genuine.

If `$validemail` isn't `false`, it's safe to incorporate into a `Reply-To` email header. Since the last value added to `$headers` in step 1 doesn't end with a carriage return and newline character, they're added before `Reply-To`. When building the `$headers` string, it doesn't matter whether you put the `\r\n` at the end of a header or at the beginning of the next one, as long as a carriage return and newline character separates them.

If `$validemail` is `false`, `$errors['email']` is added to the `$errors` array.

3. You now need to amend the `<label>` for the `email` field in `contact.php` like this:

```
<label for="email">Email:
<?php if ($missing && in_array('email', $missing)) { ?>
  <span class="warning">Please enter your email address</span>
<?php } elseif (isset($errors['email'])) { ?>
  <span class="warning">Invalid email address</span>
<?php } ?>
</label>
```

This adds an `elseif` clause to the first conditional statement and displays a different warning if the email address fails validation.

4. Save `contact.php`, and test the form by leaving all fields blank and clicking **Send message**. You'll see the original error message. Test it again by entering a value that isn't an email address in the **Email** field. This time, you'll see the invalid message. The same happens if you enter two email addresses.

You can check your code against `contact_07.php` and `processmail.inc_03.php` in the `ch05` folder.

PHP Solution 5-6: Building the message body and sending the mail

Many PHP tutorials show how to build the message body manually like this:

```
$message = "Name: $name\r\n\r\n";
$message .= "Email: $email\r\n\r\n";
$message .= "Comments: $comments";
```

This adds a label to identify which field the input comes from and inserts two carriage returns and newline characters between each one. This is fine for a small number of fields, but it soon becomes tedious with more fields. As long as you give your form fields meaningful `name` attributes, you can build the message body automatically with a `foreach` loop, which is the approach taken in this PHP solution.

> *The `name` attribute must not contain any spaces. If you want to use multiple words to name your form fields, join them with an underscore or hyphen, for example: `first_name` or `first-name`.*

Continue working with the same files as before. Alternatively, use `contact_07.php` and `processmail.inc_03.php` from the `ch05` folder.

1. Add the following code at the bottom of the script in `processmail.inc.php`:

   ```
   $mailSent = false;
   ```

   This initializes a variable that will be used to redirect the user to a thank you page after the mail has been sent. It needs to be set to `false` until you know the `mail()` function has succeeded.

2. Now add that code that builds the message. It goes immediately after the variable you have just initialized.

   ```
   // go ahead only if not suspect and all required fields OK
   if (!$suspect && !$missing && !$errors) {
     // initialize the $message variable
     $message = '';
     // loop through the $expected array
     foreach($expected as $item) {
       // assign the value of the current item to $val
       if (isset(${$item}) && !empty(${$item})) {
         $val = ${$item};
       } else {
         // if it has no value, assign 'Not selected'
         $val = 'Not selected';
       }
       // if an array, expand as comma-separated string
       if (is_array($val)) {
         $val = implode(', ', $val);
       }
       // replace underscores and hyphens in the label with spaces
       $item = str_replace(array('_', '-'), ' ', $item);
       // add label and value to the message body
       $message .= ucfirst($item).": $val\r\n\r\n";
     }

     // limit line length to 70 characters
     $message = wordwrap($message, 70);

     $mailSent = true;
   }
   ```

This is another complex block of code that you might prefer just to copy and paste. Still, you need to know what it does. In brief, the code checks that `$suspect`, `$missing`, and `$errors` are all `false`. If they are, it builds the message body by looping through the `$expected` array and stores the result in `$message` as a series of label/value pairs. The label is derived from the input field's `name` attribute. Underscores and hyphens in `name` attributes are replaced by spaces, and the first letter is set to uppercase.

If a field that's not specified as required is left empty, its value is set to "Not selected." The code also processes values from multiple-choice elements, such as check box groups and `<select>` lists, which are transmitted as subarrays of the `$_POST` array. The `implode()` function converts the subarrays into comma-separated strings.

After the message body has been combined into a single string, it's passed to the `wordwrap()` function to limit the line length to 70 characters. The code that sends the email still needs to be added, but for testing purposes, `$mailSent` has been set to `true`.

If you're interested in learning how the code in this block works, read the inline comments, which describe each stage of the process. The key to understanding it is in the following conditional statement:

```
if (isset(${$item}) && !empty(${$item})) {
  $val = ${$item};
}
```

The rather odd-looking `${$item}` is what's known as a **variable variable** (the repetition is deliberate, not a misprint). Since the value of `$item` is `name` the first time the loop runs, `${$item}` refers to `$name`. In effect, the conditional statement becomes this:

```
if (isset($name) && !empty($name)) {
  $val = $name;
}
```

On the next pass through the loop, `${$item}` refers to `$email`, and so on.

The vital point about this script is that it builds the message body only from items in the `$expected` array. You must list the names of all form fields in the `$expected` array for it to work.

3. Save `processmail.inc.php`. Locate this code block at the bottom of `contact.php`:

```
<pre>
<?php if ($_POST) {print_r($_POST);} ?>
</pre>
```

4. Change it to this:

```
<pre>
<?php if ($_POST && $mailSent) {
  echo htmlentities($message, ENT_COMPAT, 'UTF-8') . "\n";
  echo 'Headers: '. htmlentities($headers, ENT_COMPAT, 'UTF-8');
} ?>
</pre>
```

This checks that the form has been submitted and the mail is ready to send. It then displays the values in `$message` and `$headers`. Both values are passed to `htmlentities()` to ensure they display correctly in the browser.

5. Save `contact.php`, and test the form by entering your name, email address, and a brief comment. When you click **Send message**, you should see the message body and headers displayed at the bottom of the page, as shown in Figure 5-8.

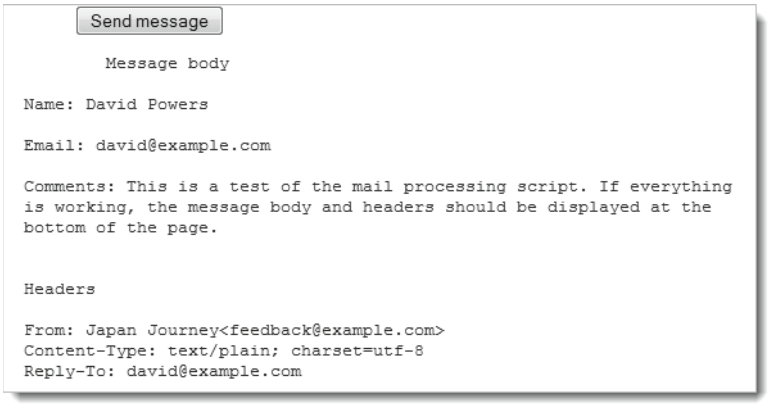


Figure 5-8. Verifying that the message body and headers are correctly formed

Assuming that the message body and headers display correctly at the bottom of the page, you're ready to add the code to send the email. If your code didn't work, check it against `contact_08.php` and `processmail.inc_04.php` in the `ch05` folder.

6. In `processmail.inc.php`, add the code to send the mail. Locate the following line:

```
$mailSent = true;
```

Change it to this:

```
$mailSent = mail($to, $subject, $message, $headers);
if (!$mailSent) {
  $errors['mailfail'] = true;
}
```

This passes the destination address, subject line, message body, and headers to the `mail()` function, which returns `true` if it succeeds in handing the email to the web server's mail transport agent (MTA). If it fails—perhaps because the mail server is down—`$mailSent` is set to `false`, and the conditional statement adds an element to the `$errors` array, allowing you to preserve the user's input when the form is redisplayed.

7. In the PHP block at the top of `contact.php`, add the following conditional statement immediately after the command that includes `processmail.inc.php`:

```
require('./includes/processmail.inc.php');
if ($mailSent) {
```

```
        header('Location: http://www.example.com/thank_you.php');
        exit;
    }
}
?>
```

Replace `www.example.com` with your own domain name. This checks if `$mailSent` is `true`. If it is, the `header()` function redirects the user to `thank_you.php`, a page acknowledging that the message has been sent. The `exit` command on the following line ensures that the script is terminated after the page has been redirected.

There's a copy of `thank_you.php` in the `ch05` folder.

8. If `$mailSent` is `false`, `contact.php` is redisplayed, and you need to warn the user that the message couldn't be sent. Edit the conditional statement just after the `<h2>` heading like this:

```
<h2>Contact Us </h2>
<?php if ((($_POST && $suspect) || ($_POST && isset($errors['mailfail'])))) { ?>
  <p class="warning">Sorry, your mail could not be sent. Please try later.</p>
```

The original and new conditions have been wrapped in parentheses, so each pair is considered as a single entity. The warning about the message not being sent is displayed if the form has been submitted and suspect phrases have been found, *or* if the form has been submitted and `$errors['mailfail']` has been set.

9. Delete the code block (including the `<pre>` tags) that displays the message body and headers at the bottom of `contact.php`.

10. Testing this locally is likely to result in the thank you page being shown, but the email never arriving. This is because most testing environments don't have an MTA. Even if you set one up, most mail servers reject mail from unrecognized sources. Upload `contact.php` and all related files, including `processmail.inc.php` and `thank_you.php` to your remote server, and test the contact form there.

    You can check your code with `contact_09.php` and `processmail.inc_05.php` in the `ch05` folder.

Troubleshooting mail()

It's important to understand that `mail()` isn't an email program. PHP's responsibility ends as soon as it passes the address, subject, message, and headers to the MTA. It has no way of knowing if the email is delivered to its intended destination. Normally, email arrives instantaneously, but network logjams can delay it by hours or even a couple of days.

If you're redirected to the thank you page after sending a message from `contact.php`, but nothing arrives in your inbox, check the following:

- Has the message been caught by a spam filter?
- Have you checked the destination address stored in `$to`? Try an alternative email address to see if it makes a difference.

- Have you used a genuine address in the `From` header? Using a fake or invalid address is likely to cause the mail to be rejected. Use a valid address that belongs to the same domain as your web server.
- Check with your hosting company to see if the fifth argument to `mail()` is required. If so, it should normally be a string composed of `-f` followed by your email address. For example, `david@example.com` becomes `'-fdavid@example.com'`.

If you still don't receive messages from `contact.php`, create a file with this simple script:

```
<?php
ini_set('display_errors', '1');
$mailSent = mail('you@example.com', 'PHP mail test', 'This is a test email');
if ($mailSent) {
  echo 'Mail sent';
} else {
  echo 'Failed';
}
```

Replace `you@example.com` with your own email address. Upload the file to your website, and load the page into a browser.

If you see an error message about there being no `From` header, add one as a fourth argument to the `mail()` function like this:

```
$mailSent = mail('you@example.com', 'PHP mail test', 'This is a test email', ➥
  'From: me@example.com');
```

It's usually a good idea to use a different address from the destination address in the first argument.

If your hosting company requires the fifth argument, adjust the `mail()` function like this:

```
$mailSent = mail('you@example.com', 'PHP mail test', 'This is a test email', null, ➥
  '-fme@example.com');
```

Using the fifth argument normally replaces the need to supply a `From` header, so using `null` (without quotes) as the fourth argument indicates that it has no value.

If you see **Mail sent** and no mail arrives, or you see **Failed** after trying all five arguments, consult your hosting company for advice.

If you receive the test email from this script but not from `contact.php`, it means you have made a mistake in the code, or that you have forgotten to upload `processmail.inc.php`.

Keeping spam at bay

Validating user input on the server is an important weapon in the fight against spam. Unfortunately, spam merchants are resourceful and often find ways of circumventing measures designed to stop them. Opinions differ about the effectiveness of anti-spam techniques, but one that's worth considering is reCAPTCHA (`www.google.com/recaptcha/captcha`).

CAPTCHA stands for Completely Automated Public Turing Test to Tell Computers and Humans Apart. In its most common form, the user is presented with an image of random characters that need to be typed correctly into a text field. The images are designed to be unreadable by optical character recognition

(OCR) software, but humans often have equal difficulty in reading them. The downside of CAPTCHA tests is that they also present a barrier to the blind and people with poor eyesight.

What makes reCAPTCHA (see Figure 5-9) stand out among similar anti-spam measures is that it automatically provides an option to refresh the image if the user can't read it. Perhaps more important, it offers an audio alternative for people with visual difficulties.

Figure 5-9. Adding a reCAPTCHA widget to a form is an effective anti-spam measure.

Using reCAPTCHA actually has a double benefit. The images used by the reCAPTCHA service come from books and newspapers that have been digitized but which OCR software has difficulty in deciphering. The user is asked to type two words, one of which has been successfully deciphered by OCR. Success or failure is determined by the response to the known word, which could be on either the left or the right. The service collates responses to the unknown word, and uses them to improve the accuracy of OCR technology.

To use reCAPTCHA, you need to set up a Google account, which is free, and obtain a pair of software keys (random strings designed to prevent spammers from circumventing the test). Once you have set up an account, incorporating a reCAPTCHA widget into your contact form is easy.

PHP Solution 5-7: Incorporating a reCAPTCHA widget into your form

This PHP solution describes how to obtain a pair of software keys and add a reCAPTCHA widget to `contact.php`. Continue working with the same files as before. Alternatively, use `contact_09.php` and `processmail.inc_05.php` from the `ch05` folder.

1. Go to `www.google.com/recaptcha/whyrecaptcha`, and click the **Sign up Now!** button. If you have a Gmail account, log in with your email address and password. If you don't have a Google account, you'll be prompted to create one.

2. To create the software keys, enter your website's domain name, select the check box if you want to enable them on all domains, and click **Create Key**. The public and private keys are random strings of characters. Copy and save them in a text file on your local computer.

3. You also need `recaptchalib.php`, which contains the PHP code to generate the reCAPTCHA widget. There's a copy in the `includes` folder. To get the most up-to-date version go to `http://code.google.com/apis/recaptcha/docs/php.html`, and click the link for the **reCAPTCHA PHP library**.

4. Include `recaptchalib.php` in `contact.php`. The file is needed both when the form first loads and when the mail processing script runs, so the `include` command needs to come before the conditional statement that runs only if the form has been submitted. You also need to create variables for the public and private keys. Edit the code at the top of `contact.php` like this (using your own public and private keys):

```
<?php
include('./includes/title.inc.php');
require_once('./includes/recaptchalib.php');
$public_key = 'your_public_key';
$private_key = 'your_private_key';
$errors = array();
```

5. The code that checks the user's response must be run only when the form has been submitted. If you plan to use a reCAPTCHA widget on every site, you can put it in `processmail.inc.php` immediately after the code that validates the email address. However, it will trigger an error if you decide not to use reCAPTCHA, so I have put it in `contact.php` just before `processmail.inc.php` is included. The code looks like this:

```
$headers .= 'Content-Type: text/plain; charset=utf-8';
$response = recaptcha_check_answer($private_key, $_SERVER['REMOTE_ADDR'], ➥
  $_POST['recaptcha_challenge_field'], $_POST['recaptcha_response_field']);
if (!$response->is_valid) {
  $errors['recaptcha'] = true;
}
require('./includes/processmail.inc.php');
```

   The `recaptcha_get_answer()` function takes four arguments: your private key, a PHP superglobal variable that identifies the user's IP address, and two `$_POST` variables that contain the challenge and response. The result is stored in an object called `$response`.

The conditional statement checks the response by accessing the object's `is_valid` property. If the response is invalid, `$errors['recaptcha']` is added to the `$errors` array, preventing `processmail.inc.php` from sending the email.

6. To display the reCAPTCHA widget in the contact form, add the following code above the submit button:

```
<?php if (isset($errors['recaptcha'])) { ?>
  <p class="warning">The values didn't match. Try again.</p>
<?php }
echo recaptcha_get_html($public_key); ?>
<p>
  <input name="send" id="send" type="submit" value="Send message">
</p>
```

   This adds a paragraph that displays an error message if the user's response was invalid, followed by the code to display the reCAPTCHA widget.

7. Upload the revised version of `contact.php` and `recaptchalib.php` to your remote server, and load the contact form into a browser. A reCAPTCHA widget should appear above the submit button as shown in Figure 5-9.

   You can check your code against `contact_10.php` in the `ch05` folder.

The code generated by reCAPTCHA creates a `<div>` with the ID `recaptcha_widget_div`, which you can use to create a CSS style rule to align the widget with other form elements.

You can find instructions on how to customize the look of a reCAPTCHA widget at `http://code.google.com/apis/recaptcha/docs/customization.html`. At the time of this writing, you can choose from four themes or create your own. You can also change the language. There are built-in translations for several languages, including French, Spanish, and Russian. If your language isn't supported, you can define your own custom translations.

Handling multiple-choice form elements

The form in `contact.php` uses only text input fields and a text area. To work successfully with forms, you also need to know how to handle multiple-choice elements, namely:

- Radio buttons
- Check boxes
- Drop-down option menus
- Multiple-choice lists

The principle behind them is the same as the text input fields you have been working with: the `name` attribute of the form element is used as the key in the `$_POST` array. However, there are some important differences:

- Check box groups and multiple-choice lists store selected values as an array, so you need to add an empty pair of square brackets at the end of the `name` attribute for these types of input. For example, for a check box group called `interests`, the `name` attribute in each `<input>` tag

should be `name="interests[]"`. If you omit the square brackets, only the last item selected is transmitted through the `$_POST` array.

- The values of selected items in a check box group or multiple-choice list are transmitted as a subarray of the `$_POST` array. The code in PHP Solution 5-6 automatically converts these subarrays to comma-separated strings. However, when using a form for other purposes, you need to extract the values from the subarrays. You'll see how to do so in later chapters.
- Radio buttons, check boxes, and multiple-choice lists are *not* included in the `$_POST` array if no value is selected. So, it's vital to use `isset()` to check for their existence before attempting to access their values when processing the form.

Figure 5-10 shows `contact.php` with each type of input added to the original design.

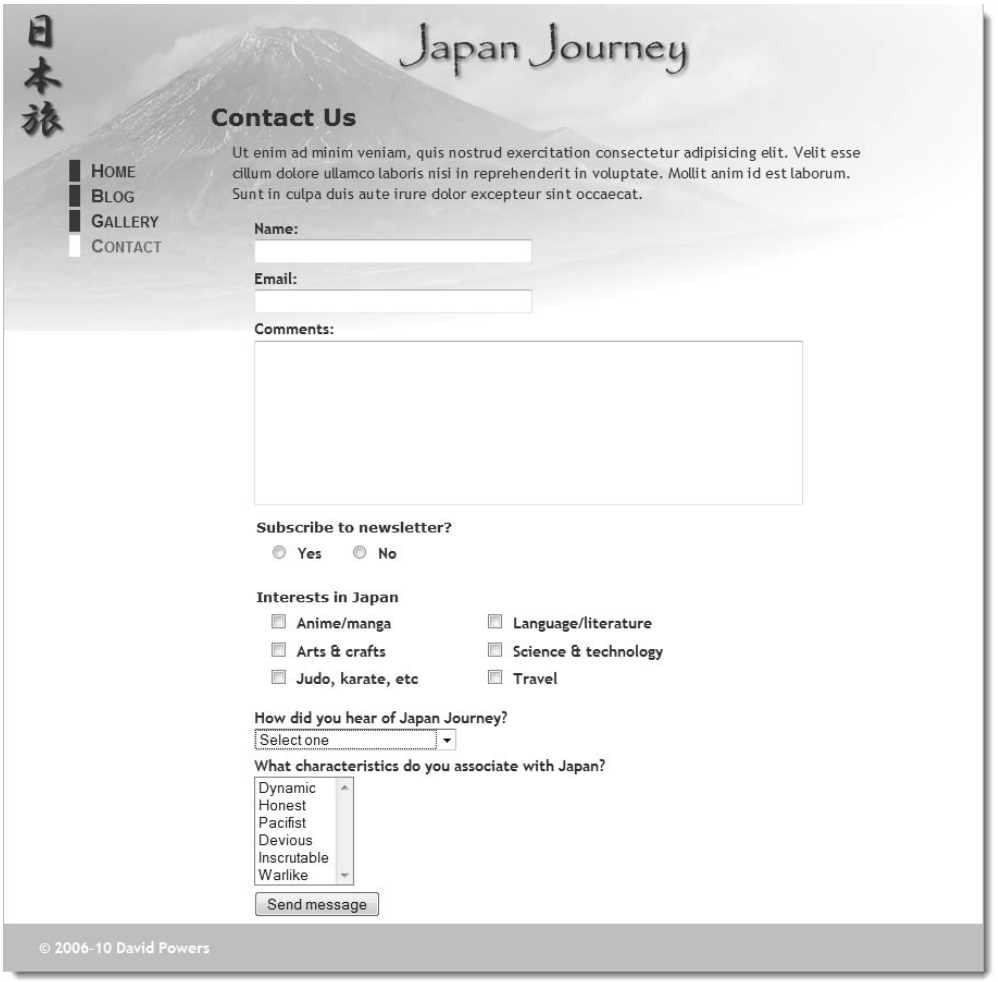


Figure 5-10. The feedback form with examples of multiple-choice form elements

The remaining PHP solutions in this chapter show how to handle multiple-choice form elements. Rather than go through each step in detail, I'll just highlight the important points. Bear the following points in mind when working through the rest of this chapter:

- Processing these elements relies on the code in `processmail.inc.php`.
- You must add the `name` attribute of each element to the `$expected` array for it to be added to the message body.
- To make a field required, add its `name` attribute to the `$required` array.
- If a field that's not required is left blank, the code in `processmail.inc.php` sets its value to "Not selected."

The completed code for the rest of the chapter is in `contact_11.php`. The reCAPTCHA widget has been omitted to simplify the page.

HTML5 adds many new types of form input. They all use the `name` attribute and send values as text or as a subarray of the `$_POST` array, so you should be able to adapt the code accordingly.

PHP Solution 5-8: Handling radio button groups

Radio button groups let you pick only one value. Although it's common to set a default value in the HTML markup, it's not obligatory. This PHP solution shows how to handle both scenarios.

1. The simple way to deal with radio buttons is to make one of them the default. The radio group is always included in the `$_POST` array, because a value is always selected.

   The code for a radio group with a default value looks like this (the `name` attributes and PHP code are highlighted in bold):

```
<fieldset id="subscribe">
  <h2>Subscribe to newsletter?</h2>
  <p>
  <input name="subscribe" type="radio" value="Yes" id="subscribe-yes"
  <?php
  if ($_POST && $_POST['subscribe'] == 'Yes') {
    echo 'checked';
  } ?>>
  <label for="subscribe-yes">Yes</label>
  <input name="subscribe" type="radio" value="No" id="subscribe-no"
  <?php
  if (!$_POST || $_POST['subscribe'] == 'No') {
    echo 'checked';
  } ?>>
  <label for="subscribe-no">No</label>
  </p>
</fieldset>
```

   All members of the radio group share the same `name` attribute. Because only one value can be selected, the `name` attribute does *not* end with a pair of empty brackets.

The conditional statement in the **Yes** button checks `$_POST` to see if the form has been submitted. If it has and the value of `$_POST['subscribe']` is "Yes," the `checked` attribute is added to the `<input>` tag.

In the **No** button, the conditional statement uses `||` (or). The first condition is `!$_POST`, which is `true` when the form hasn't been submitted. If `true`, the `checked` attribute is added as the default value when the page first loads. If `false`, it means the form has been submitted, so the value of `$_POST['subscribe']` is checked.

2. When a radio button doesn't have a default value, it's not included in the `$_POST` array, so it isn't detected by the loop in `processmail.inc.php` that builds the `$missing` array. To ensure that the radio button element is included in the `$_POST` array, you need to test for its existence after the form has been submitted. If it isn't included, you need to set its value to an empty string like this:

```
$required = array('name', 'comments', 'email', 'subscribe');
// set default values for variables that might not exist
if (!isset($_POST['subscribe'])) {
  $_POST['subscribe'] = '';
}
```

3. If the radio button group is required but not selected, you need to display an error message when the form reloads. You also need to change the conditional statements in the `<input>` tags to reflect the different behavior.

   The following listing shows the `subscribe` radio button group from `contact_11.php`, with all the PHP code highlighted in bold:

```
<fieldset id="subscribe">
  <h2>Subscribe to newsletter?
  <?php if ($missing && in_array('subscribe', $missing)) { ?>
    <span class="warning">Please make a selection</span>
  <?php } ?>
  </h2>
  <p>
  <input name="subscribe" type="radio" value="Yes" id="subscribe-yes"
  <?php
  if ($_POST && $_POST['subscribe'] == 'Yes') {
    echo 'checked';
  } ?>>
  <label for="subscribe-yes">Yes</label>
  <input name="subscribe" type="radio" value="No" id="subscribe-no"
  <?php
  if ($_POST && $_POST['subscribe'] == 'No') {
    echo 'checked';
  } ?>>
  <label for="subscribe-no">No</label>
  </p>
</fieldset>
```

The conditional statement that controls the warning message in the <h2> tag uses the same technique as for the text input fields. The message is displayed if the radio group is a required item and it's in the $missing array.

The conditional statement surrounding the checked attribute is the same in both radio buttons. It checks if the form has been submitted and displays the checked attribute only if the value in $_POST['subscribe'] matches.

PHP Solution 5-9: Handling check boxes and check box groups

Check boxes can be used in two ways:

- **Individually**: Each check box must have a unique name attribute. The value attribute is optional. If omitted, the default is "on."
- **As a group**: When used this way, all check boxes in the group share the same name attribute, which needs to end with an empty pair of square brackets for PHP to transmit the selected values as an array. To identify which check boxes have been selected, each one needs a unique value attribute.

This PHP solution shows how to deal with a check box group called interests. If no items are selected, the check box group is not included in the $_POST array. After the form has been submitted, you need to check the $_POST array to see if it contains a subarray for the check box group. If it doesn't, you need to create an empty subarray as the default value for the script in processmail.inc.php.

1. To save space, just the first two check boxes of the group are shown. The name attribute and PHP sections of code are highlighted in bold.

```
<fieldset id="interests">
<h2>Interests in Japan</h2>
<div>
  <p>
    <input type="checkbox" name="interests[]" value="Anime/manga" id="anime"
    <?php
    if ($_POST && in_array('Anime/manga', $_POST['interests'])) {
      echo 'checked';
    } ?>>
    <label for="anime">Anime/manga</label>
  </p>
  <p>
    <input type="checkbox" name="interests[]" value="Arts & crafts" id="art"
    <?php
    if ($_POST && in_array('Arts & crafts', $_POST['interests'])) {
      echo 'checked';
    } ?>>
    <label for="art">Arts &amp; crafts</label>
  </p>
. . .
</div>
</fieldset>
```

Each check box shares the same `name` attribute, which ends with an empty pair of square brackets, so the data is treated as an array. If you omit the brackets, `$_POST['interests']` contains the value of only the first check box selected.

Although the brackets must be added to the `name` attribute for multiple selections to be treated as an array, the subarray of selected values is in `$_POST['interests']`, not `$_POST['interests[]']`.

The PHP code inside each check box element performs the same role as in the radio button group, wrapping the `checked` attribute in a conditional statement. The first condition checks that the form has been submitted. The second condition uses the `in_array()` function to check whether the `value` associated with that check box is in the `$_POST['interests']` subarray. If it is, it means the check box was selected.

2. After the form has been submitted, you need to check for the existence of `$_POST['interests']`. If it hasn't been set, you must create an empty array as the default value for the rest of the script to process. The code follows the same pattern as for the radio group:

```
$required = array('name', 'comments', 'email', 'subscribe', 'interests');
// set default values for variables that might not exist
if (!isset($_POST['subscribe'])) {
  $_POST['subscribe'] = '';
}
if (!isset($_POST['interests'])) {
  $_POST['interests'] = array();
}
```

When dealing with a single check box, use an empty string instead of an empty array.

3. To set a minimum number of required check boxes, use the `count()` function to check the number of values transmitted from the form. If it's less than the minimum required, add the group to the `$errors` array like this:

```
if (!isset($_POST['interests'])) {
  $_POST['interests'] = array();
}
// minimum number of required check boxes
$minCheckboxes = 2;
if (count($_POST['interests']) < $minCheckboxes) {
  $errors['interests'] = true;
}
```

The `count()` function returns the number of elements in an array, so this creates `$errors['interests']` if fewer than two check boxes have been selected. You might be wondering why I have used a variable instead of the number like this:

```
if (count($_POST['interests']) < 2) {
```

This certainly works and it involves less typing, but `$minCheckboxes` can be reused in the error message. Storing the number in a variable means this condition and the error message always remain in sync.

4. The error message in the body of the form looks like this:

```
<h2>Interests in Japan
<?php if (isset($errors['interests'])) { ?>
  <span class="warning">Please select at least <?php echo $minCheckboxes; ?></span>
<?php } ?>
</h2>
```

PHP Solution 5-10: Using a drop-down option menu

Drop-down option menus created with the `<select>` tag are similar to radio button groups in that they normally allow the user to pick only one option from several. Where they differ is one item is always selected in a drop-down menu, even if it's only the first item inviting the user to select one of the others. As a result, this means that the `$_POST` array always contains an element referring to a `<select>` menu, whereas a radio button group is ignored unless a default value is preset.

1. The following code shows the first two items from the drop-down menu in `contact_11.php` with the PHP code highlighted in bold. As with all multiple-choice elements, the PHP code wraps the attribute that indicates which item has been chosen. Although this attribute is called `checked` in radio buttons and check boxes, it's called `selected` in `<select>` menus and lists. It's important to use the correct attribute to redisplay the selection if the form is submitted with required items missing. When the page first loads, the `$_POST` array contains no elements, so you can select the first `<option>` by testing for `!$_POST`. Once the form is submitted, the `$_POST` array always contains an element from a drop-down menu, so you don't need to test for its existence.

```
<p>
  <label for="select">How did you hear of Japan Journey?</label>
  <select name="howhear" id="howhear">
    <option value="No reply"
    <?php
    if (!$_POST || $_POST['howhear'] == 'No reply') {
      echo 'selected';
    } ?>>Select one</option>
    <option value="foED"
    <?php
    if (isset($_POST && $_POST['howhear'] == 'foED') {
      echo 'selected';
    } ?>>friends of ED</option>
. . .
    </select>
</p>
```

2. Even though an option is always selected in a drop-down menu, you might want to force users to make a selection other than the default. To do so, add the `name` attribute of the `<select>`

menu to the `$required` array, and set the `value` attribute and the `$_POST` array element for the default option to an empty string like this:

```
<option value=""
<?php
if (!$_POST || $_POST['howhear'] == '') {
  echo 'selected';
} ?>>Select one</option>
```

The `value` attribute is not required in the `<option>` tag, but if you leave it out, the form uses the text between the opening and closing tags as the selected value. So, it's necessary to set the `value` attribute explicitly to an empty string. Otherwise, "Select one" is transmitted as the selected value.

3. The code that displays a warning message if no selection has been made follows the familiar pattern:

```
<label for="select">How did you hear of Japan Journey?
<?php if ($missing && in_array('howhear', $missing)) { ?>
  <span class="warning">Please make a selection</span>
<?php } ?>
</label>
```

PHP Solution 5-11: Handling a multiple-choice list

Multiple-choice lists are similar to check boxes: they allow the user to choose zero or more items, so the result is stored in an array. If no items are selected, you need to add an empty subarray to the `$_POST` array in the same way as with a check box group.

1. The following code shows the first two items from the multiple-choice list in `contact_11.php` with the `name` attribute and PHP code highlighted in bold. Note that the `name` attribute needs a pair of square brackets on the end to store the results as an array. The code works in an identical way to the check boxes in PHP Solution 5-9.

```
<p>
  <label for="select">What characteristics do you associate with ➥
    Japan?</label>
  <select name="characteristics[]" size="6" multiple="multiple" ➥
    id="characteristics">
    <option value="Dynamic"
    <?php
    if ($_POST && in_array('Dynamic', $_POST['characteristics'])) {
      echo 'selected';
    } ?>>Dynamic</option>
    <option value="Honest"
    <?php
    if ($_POST && in_array('Honest', $_POST['characteristics'])) {
      echo 'selected';
    } ?>>Honest</option>
. . .
```

```
  </select>
</p>
```

2. In the code that processes the message, set a default value for a multiple-choice list in the same way as for an array of check boxes.

```
if (!isset($_POST['interests'])) {
  $_POST['interests'] = array();
}
if (!isset($_POST['characteristics'])) {
  $_POST['characteristics'] = array();
}
```

3. To make a multiple-choice list required and set a minimum number of choices, use the same technique as for a check box group in PHP Solution 5-9.

Chapter review

A lot of work has gone into building `processmail.inc.php`, but the beauty of this script is that it works with any form. The only parts that need changing are the `$expected` and `$required` arrays and details specific to the form, such as the destination address, headers, and default values for multiple-choice elements that won't be included in the `$_POST` array if no value is selected.

I've avoided talking about HTML email because the `mail()` function handles only plain text email. The PHP online manual at `www.php.net/manual/en/function.mail.php` shows a way of sending HTML mail by adding an additional header. However, it's not a good idea, as HTML mail should always contain an alternative text version for email programs that don't accept HTML. If you want to send HTML mail or attachments, try PHPM@iler (`http://phpmailer.worxware.com/`) or Zend_Mail (`http://zendframework.com/manual/en/zend.mail.html`).

As you'll see in later chapters, online forms lie at the heart of just about everything you do with PHP. They're the gateway between the browser and the web server. You'll come back time and again to the techniques that you have learned in this chapter.